

PDyTR

TP 3

GITHUB CON EL PROYECTO:

https://github.com/JCRigol/PDyTR_2024/tree/main/practica3

1) MANEJO DE ERRORES DE CONECTIVIDAD

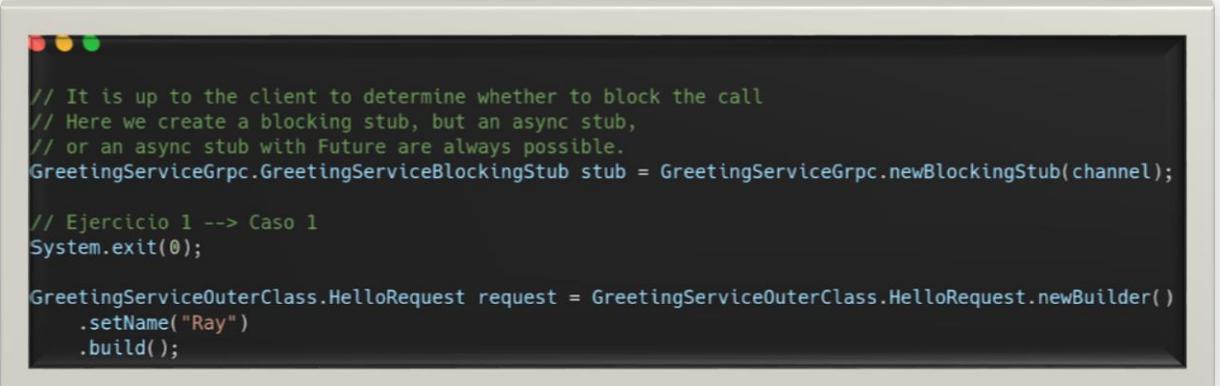
Usando como base el programa **ejemplo1** (1) de gRPC, realice los siguientes experimentos para simular y observar fallos de conectividad tanto del lado del cliente como del servidor:

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) <https://github.com/pdytr/pdytr-grpc-demo.git>

Se probaron cinco casos:

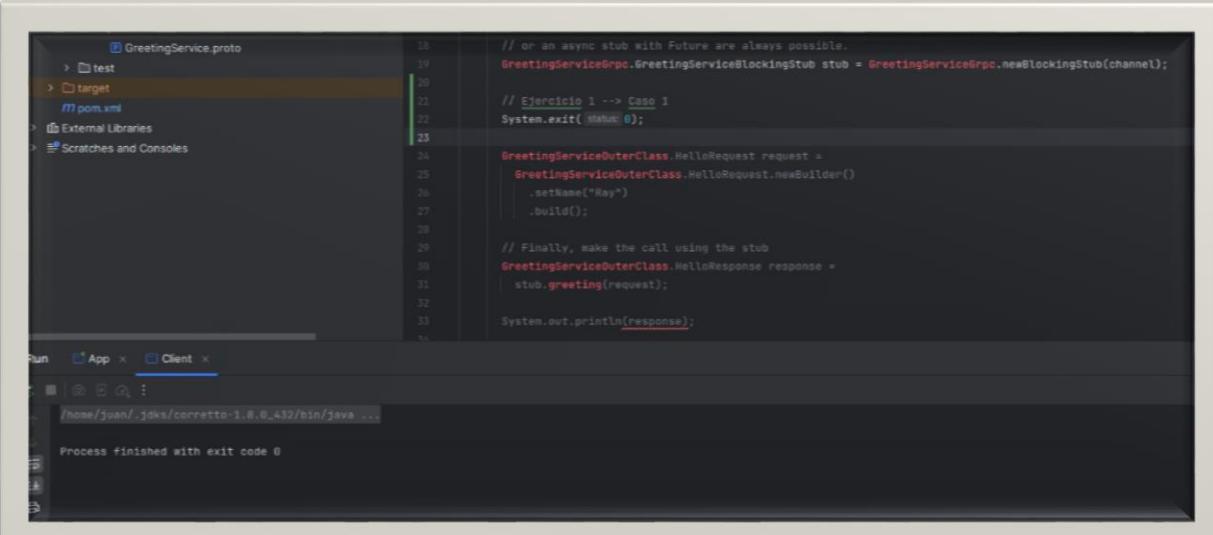
- El Cliente se desconecta luego de generar el stub



```
// It is up to the client to determine whether to block the call
// Here we create a blocking stub, but an async stub,
// or an async stub with Future are always possible.
GreetingServiceGrpc.GreetingServiceBlockingStub stub = GreetingServiceGrpc.newBlockingStub(channel);

// Ejercicio 1 --> Caso 1
System.exit(0);

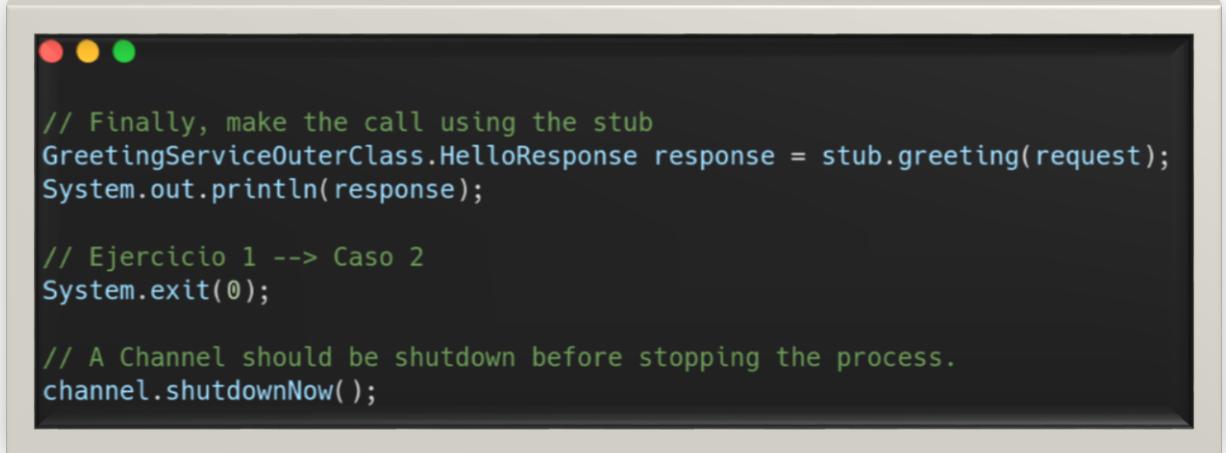
GreetingServiceOuterClass>HelloRequest request = GreetingServiceOuterClass>HelloRequest.newBuilder()
    .setName("Ray")
    .build();
```



The screenshot shows an IDE interface with the following details:

- Project Structure:** The left sidebar shows a project named "Client" with a "target" folder selected. Other visible items include "GreetingService.proto", "test", "pom.xml", "External Libraries", and "Scratches and Consoles".
- Code Editor:** The main area displays Java code. Lines 21 and 22 contain the line `System.exit(0);`. Lines 24 through 35 show the creation of a `GreetingServiceOuterClass>HelloRequest` object and its use with a stub.
- Run Tab:** At the bottom, the "Run" tab is active, showing the command `/home/juan/.jdks/corretto-1.8.0_432/bin/java ...`.
- Output Tab:** The "App" tab is active, showing the output: `Process finished with exit code 0`.

- El Cliente se desconecta antes de ejecutar el shutdown



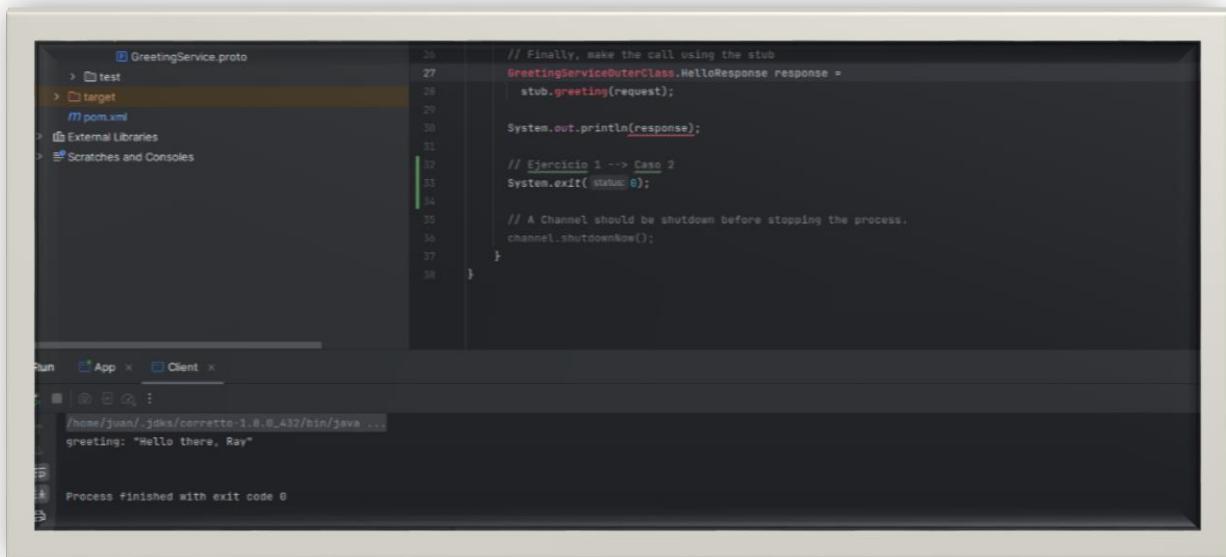
```

// Finally, make the call using the stub
GreetingServiceOuterClass>HelloResponse response = stub.greeting(request);
System.out.println(response);

// Ejercicio 1 --> Caso 2
System.exit(0);

// A Channel should be shutdown before stopping the process.
channel.shutdownNow();

```



The screenshot shows the IntelliJ IDEA interface with the GreetingService.proto file open in the left panel and the generated Client.java code in the right panel. The code is identical to the one shown in the terminal above. Below the editor, a terminal window shows the output of running the application, which prints "greeting: "Hello there, Ray"" and exits with code 0.

- El Service se desconecta al entrar



```

public void greeting(GreetingServiceOuterClass>HelloRequest request,
                      StreamObserver<GreetingServiceOuterClass>HelloResponse> responseObserver) {
    // HelloRequest has toString auto-generated.
    System.out.println(request);

    // Ejercicio 1 --> Caso 3
    System.exit(0);

    // You must use a builder to construct a new Protobuffer object
    GreetingServiceOuterClass>HelloResponse response =
        GreetingServiceOuterClass>HelloResponse.newBuilder()
            .setGreeting("Hello there, " + request.getName())
            .build();
}

```

```

@Override
public void greeting(GreetingServiceOuterClass.HelloRequest request,
                     StreamObserver<GreetingServiceOuterClass.HelloResponse> responseObserver) {
    // HelloRequest has toString auto-generated.
    System.out.println(request);

    // Ejercicio 1 --> Caso 3
    System.exit(0);

    // You must use a builder to construct a new Protobuf object
    GreetingServiceOuterClass.HelloResponse response = GreetingServiceOuterClass.HelloResponse.newBuilder()
        .setGreeting("Hello there, " + request.getName())
        .build();

    // Use responseObserver to send a single response back
    responseObserver.onNext(response);

    // When you are done, you must call onComplete.
    responseObserver.onCompleted();
}

```

- El Service se desconecta previo al onNext

```

// You must use a builder to construct a new Protobuf object
GreetingServiceOuterClass.HelloResponse response = GreetingServiceOuterClass.HelloResponse.newBuilder()
    .setGreeting("Hello there, " + request.getName())
    .build();

// Ejercicio 1 --> Caso 4
System.exit(0);

// Use responseObserver to send a single response back
responseObserver.onNext(response);

```

```

@Override
public void greeting(GreetingServiceOuterClass.HelloRequest request,
                     StreamObserver<GreetingServiceOuterClass.HelloResponse> responseObserver) {
    // HelloRequest has toString auto-generated.
    System.out.println(request);

    // Ejercicio 1 --> Caso 4
    System.exit(0);

    // Use responseObserver to send a single response back
    responseObserver.onNext(response);

    // When you are done, you must call onComplete.
    responseObserver.onCompleted();
}

```

- El Service se desconecta previo al onCompleted

```

// Use responseObserver to send a single response back
responseObserver.onNext(response);

// Ejercicio 1 --> Caso 5
System.exit(0);

// When you are done, you must call onCompleted.
responseObserver.onCompleted();

```

```

GreetingServiceImpl.GreetingServiceGrpc<-, -> void hello(GreetingServiceGrpc.GreetingRequest request) {
    GreetingServiceGrpc.GreetingResponse response = GreetingServiceGrpc.class.HelloResponse.newBuilder()
        .setGreeting("Hello there, " + request.getName())
        .build();

    // Use responseObserver to send a single response back
    responseObserver.onNext(response);

    // Ejercicio 1 --> Caso 5
    System.exit(0);

    // When you are done, you must call onCompleted.
    responseObserver.onCompleted();
}

```

Client

```

/home/juan/Downloads/NetBeans 8.0.2/bin/java ...
Server started
name: "Ray"

Process finished with exit code 0

```

```

Exception in thread "main" 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:103)
        at io.grpc.stub.ClientCalls.blockingUnaryCall(ClientCalls.java:124)
        at proto.example.grpc.GreetingServiceGrpc$GreeterBlockingStub.greeting(
                GreetingServiceImpl.java:16)
        at proto.example.grpc.Client.main(Client.java:29)

```

- El Service se desconecta luego del onCompleted

```

// When you are done, you must call onCompleted.
responseObserver.onCompleted();

// Ejercicio 1 --> Caso 6
System.exit(0);

```

Primera ejecución del Cliente:

```
java
└─ pdytr.example.grpc
    └─ App
        └─ Client
            └─ GreetingServiceImpl
    └─ proto
Run  App  X
/home/juan/.jdeps/corretto-1.8.0_432/bin/java ...
Server started
name: "Ray"
Process finished with exit code 0
```

```
Client  X
/home/juan/.jdeps/corretto-1.8.0_432/bin/java ...
greeting: "Hello there, Ray"
Process finished with exit code 0
```

Segunda ejecución del Cliente:

```
java
└─ pdytr.example.grpc
    └─ App
        └─ Client
            └─ GreetingServiceImpl
    └─ proto
Run  App  X
/home/juan/.jdeps/corretto-1.8.0_432/bin/java ...
Server started
name: "Ray"
Process finished with exit code 0
```

```
Client  X
/home/juan/.jdeps/corretto-1.8.0_432/bin/java ...
Exception in thread "main" io.grpc.StatusRuntimeException: UNAVAILABLE
at io.grpc.stub.ClientCalls.toStatusRuntimeException(ClientCalls.java:218)
at io.grpc.stub.ClientCalls.getUnchecked(ClientCalls.java:193)
at io.grpc.stub.ClientCalls.blockingUnaryCall(ClientCalls.java:124)
at pdytr.example.grpc.GreetingServiceBlockingStub.greeting(GreetingServiceBlockingStub.java:63)
at pdytr.example.grpc.Client.main(Client.java:22)
Caused by: io.netty.channel.AbstractChannel$AnnotatedConnectException: Conexión rehusada
localhost/[27.9.0.1:8080]
at sun.nio.ch.SocketChannelImpl.checkConnect(SocketChannelImpl)
at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:71)
at io.netty.channel.socket.nio.NioSocketChannel.doFinishConnect(NioSocketChannel.java:123)
at io.netty.channel.nio.AbstractNioChannel$AbstractNioUnsafe.finishConnect(AbstractNioChannel.java:140)
at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:58)
at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:50)
at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:49)
at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:85)
at io.netty.util.concurrent.DefaultThreadFactory$DefaultRunnableDecorator.run(DefaultThreadFactory.java:138)
at java.lang.Thread.run(Thread.java:750)
Caused by: java.net.ConnectException: Conexión rehusada
... 11 more
```

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.

Se configuró un **DEADLINE** de 5 segundos en el stub del cliente utilizando la función withDeadlineAfter(5, TimeUnit.SECONDS) del BlockingStub.

```
// It is up to the client to determine whether to block the call  
// Here we create a blocking stub, but an async stub,  
// or an async stub with Future are always possible.  
GreetingServiceGrpc.GreetingServiceBlockingStub stub =  
    GreetingServiceGrpc.newBlockingStub(channel)  
        .withDeadlineAfter(5, TimeUnit.SECONDS);
```

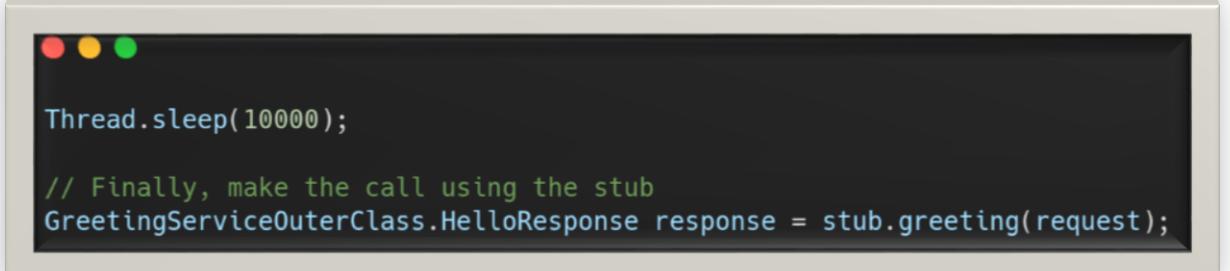
Luego, se hizo dormir al Service durante 10 segundos al levantarse.

```
//Demora del servidor al responder  
try  
{  
    Thread.sleep(10000);  
}  
  
catch (InterruptedException e)  
{  
    throw new RuntimeException(e);  
}
```

```
[home/juan/.java/corretto-1.8.0.432/bin/java -cp  
Server started  
name: "Ray"  
[home/juan/.java/corretto-1.8.0.432/bin/java -cp  
Exception in thread "main" io.grpc.StatusRuntimeException: DEADLINE_EXCEEDED: deadline  
exceeded after 407423915ns  
at io.grpc.stub.ClientCalls.toStatusRuntimeException(ClientCalls.java:210)  
at io.grpc.stub.ClientCalls.performanceCheckedException(ClientCalls.java:107)  
at io.grpc.stub.ClientCalls.blockingUnaryCall(ClientCalls.java:123)  
at pdatr.example.grpc.GreetingServiceBlockingStub$BlockingStub.greeting  
(GreetingServiceGrpc.java:163)  
at pdatr.example.grpc.Client.main(Client.java:17)
```

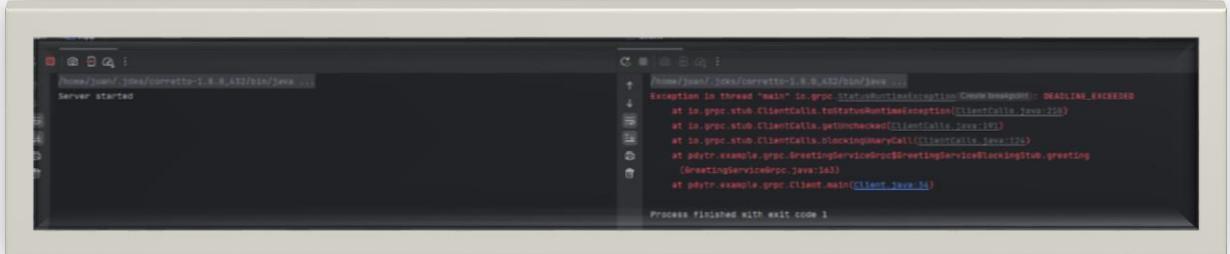
Finalmente, se hizo dormir al Cliente durante 10 segundos antes y después de llamar a la función greeting() del stub.

Caso 1:



```
Thread.sleep(10000);

// Finally, make the call using the stub
GreetingServiceOuterClass>HelloResponse response = stub.greeting(request);
```

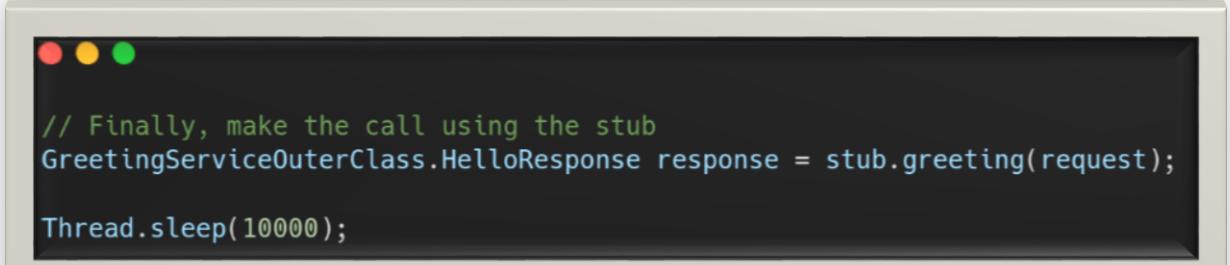


```
/home/juan/.jdkx/corretto-1.8.0_432/bin/java ...
Server started

Exception in thread "main" io.grpc.StatusRuntimeException: DEADLINE_EXCEEDED
at io.grpc.stub.ClientCalls.toStatusRuntimeException(ClientCalls.java:218)
at io.grpc.stub.ClientCalls.getUnchecked(ClientCalls.java:109)
at io.grpc.stub.ClientCalls.blockingUnaryCall(ClientCalls.java:114)
at pdatr.example.grpc.GreetingServiceGrpc$GreeterBlockingStub.greeting(
    GreetingServiceGrpc.java:363)
at pdatr.example.grpc.Client.main(Client.java:54)

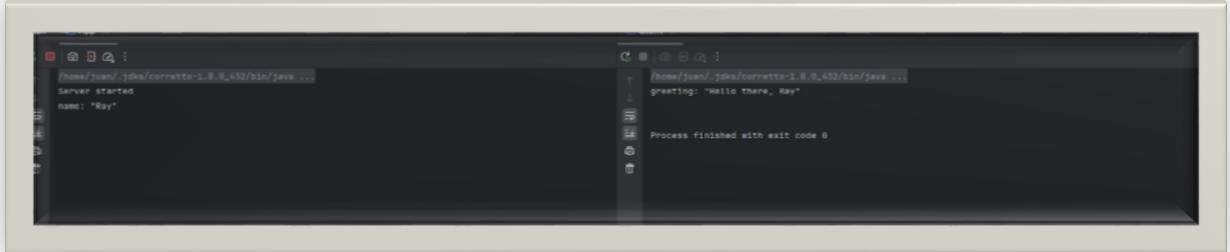
Process finished with exit code 1
```

Caso 2:



```
// Finally, make the call using the stub
GreetingServiceOuterClass>HelloResponse response = stub.greeting(request);

Thread.sleep(10000);
```



```
/home/juan/.jdkx/corretto-1.8.0_432/bin/java ...
Server started
name: "Ray"

greeting: "Hello there, Ray"

Process finished with exit code 0
```

2) ANÁLISIS DE APIs EN gRPC

Describa y analice los distintos tipos de APIs que ofrece gRPC. Con base en el análisis, elabore una conclusión sobre cuál sería la mejor opción para los siguientes escenarios:

a) Un sistema de pub/sub.

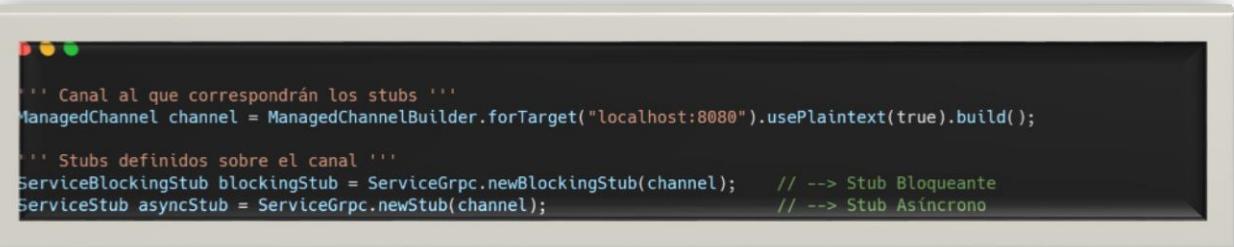
b) Un sistema de archivos FTP.

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?

<https://grpc.io/docs/what-is-grpc/core-concepts/> Documentación / Listado de las APIs

Para el desglose de las APIs encontradas, se utilizarán las siguientes variables:



```
``` Canal al que corresponderán los stubs ```
ManagedChannel channel = ManagedChannelBuilder.forTarget("localhost:8080").usePlaintext(true).build();

``` Stubs definidos sobre el canal ```
ServiceBlockingStub blockingStub = ServiceGrpc.newBlockingStub(channel); // --> Stub Bloqueante
ServiceStub asyncStub = ServiceGrpc.newStub(channel); // --> Stub Asincrono
```

A su vez, los métodos del servidor solo presentaran una implementación (en los casos donde haya dos servicios definidos para la misma API), debido a que la diferenciación en el manejo de las APIs al utilizar distintos tipos de stubs sólo se produce del lado del cliente.

```
''' Servicio '''
service Service {
    rpc unaryBlocking(UnaryMsg) returns (UnaryRes);
    rpc unaryAsync(UnaryMsg) returns (UnaryRes);

    rpc serverStrBlocking(ServerStrMsg) returns (stream ServerStrRes);
    rpc serverStrAsync(ServerStrMsg) returns (stream ServerStrRes);

    rpc clientStrAsync(stream ClientStrMsg) returns (ClientStrRes);

    rpc bidirectionalAsync(stream BidirMsg) returns (stream BidirRes);
}

''' Mensajes (solo para completitud del ejemplo) '''
message UnaryMsg {}
message UnaryRes {}
message ServerStrMsg {}
message ServerStrRes {}
message ClientStrMsg {}
message ClientStrRes {}
message BidirMsg {}
message BidirRes {}
```

TIPOS DE APIs ENCONTRADAS:

RPC Unario:

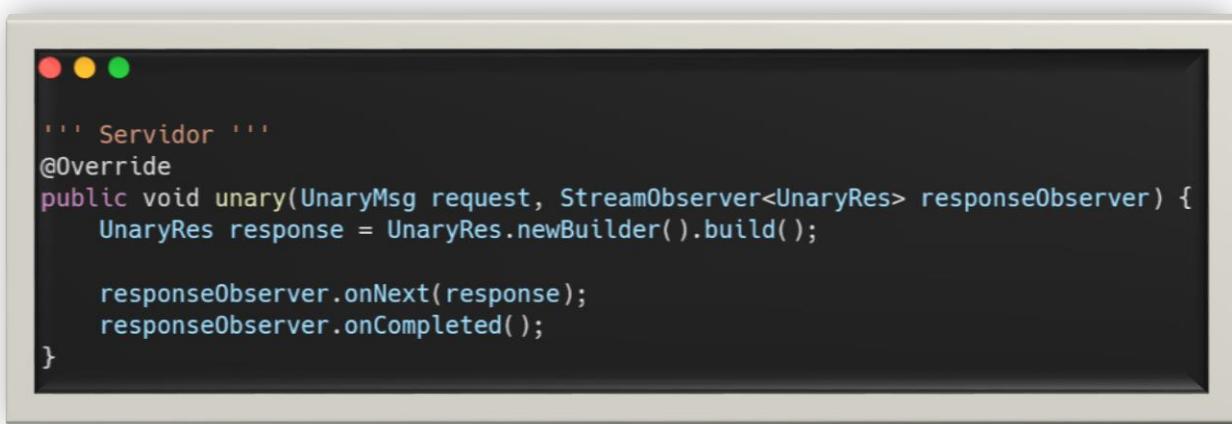
El cliente le hace una única solicitud al servidor, y el mismo le devuelve una única respuesta.

```
''' Cliente '''
UnaryMsg request = UnaryMsg.newBuilder().build();
UnaryRes blockingResponse = blockingStub.unaryBlocking(request);

asyncResponse.unaryAsync(request, new StreamObserver<UnaryRes>() {
    @Override
    public void onNext(UnaryRes response) {}

    @Override
    public void onError(Throwable t) {}

    @Override
    public void onCompleted() {}
});
```



```
''' Servidor '''
@Override
public void unary(UnaryMsg request, StreamObserver<UnaryRes> responseObserver) {
    UnaryRes response = UnaryRes.newBuilder().build();

    responseObserver.onNext(response);
    responseObserver.onCompleted();
}
```

Cuando el cliente llama a su método stub el servidor es notificado respecto a la invocación a RPC con la metadata del cliente, el método a ejecutar y el tiempo máximo de espera (en caso de haberse configurado uno).

Luego el servidor envía su metadata. Esto puede suceder inmediatamente, o esperar a recibir la solicitud del cliente. El comportamiento específico a seguir depende de cómo se haya escrito la aplicación.

Al tener la solicitud del cliente, el servidor opera sobre la misma y genera un objeto respuesta en base a los resultados. Esta respuesta es enviada al cliente, junto al estatus de la operación y, opcionalmente, metadata.

Si el estatus es OK, entonces el cliente recibe la respuesta, completando la llamada a la API del lado del cliente.

Finalmente, el cliente operará sobre la respuesta recibida. En el caso de haber utilizado un stub bloqueante, el cliente permanecerá bloqueado hasta recibir la respuesta del servidor, y luego podrá utilizar el objeto “blockingResponse” para operar sobre lo recibido.

En caso de haber realizado la llamada utilizando un stub asíncrono, el cliente deberá definir un objeto anónimo previo a realizar la llamada. El mismo será una implementación de la interfaz StreamObserver que definirá el comportamiento a realizar en base a la respuesta a la llamada.

El cliente podrá entonces continuar con su ejecución y, al finalizarse la llamada, se ejecutará el método correspondiente implementado en el objeto anónimo (“onNext” en caso exitoso, “onError” en caso de error y “onCompleted” como cierre de la comunicación, similar a un finally en un try/catch).

Se destaca que este comportamiento se ejecutará en un Thread aparte, paralelamente al comportamiento actual del cliente.

RPC Streaming del lado del servidor:

Similar a la unaria. Se diferencia en que el servidor responde a la solicitud del cliente con un conjunto de mensajes en vez de un único objeto respuesta.

```
''' Cliente '''
ServerStrMsg request = ServerStrMsg.newBuilder().build();
Iterator<ServerStrRes> blockingResponse = blockingStub.serverStrBlocking(request);

asyncResponse.serverStrAsync(request, new StreamObserver<ServerStrRes>() {
    @Override
    public void onNext(ServerStrRes response) {}

    @Override
    public void onError(Throwable t) {}

    @Override
    public void onCompleted() {}
});
```

```
''' Servidor '''
@Override
public void serverStr(ServerStrMsg request, StreamObserver<ServerStrRes> responseObserver) {
    for (int i = 0; i < 100; i++) {
        ServerStrRes response = ServerStrRes.newBuilder().build();
        responseObserver.onNext(response);
    }

    responseObserver.onCompleted();
}
```

Al finalizar de enviar todos los mensajes, el servidor envía el estatus y (opcionalmente) la metadata y termina su operación.

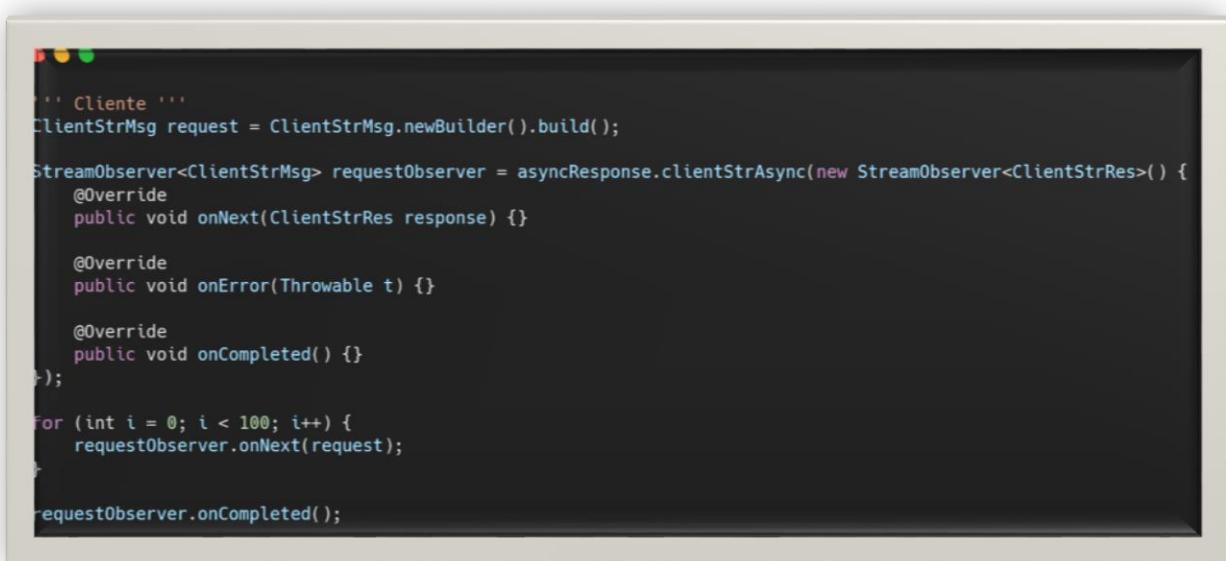
Del lado del cliente, en el caso de utilizar un stub bloqueante el mismo se bloquea hasta recibir todas las respuestas del servidor sobre el iterador. Luego utiliza el mismo para procesar las respuestas.

En el caso asíncrono, la operativa es casi idéntica al caso asíncrono de la API unaria. La única salvedad, relativa más bien al caso bloqueante de esta misma API, es que la operatoria sobre cada respuesta individual se da al instante de recibir la misma (se puede circumventar esto en la definición del “onNext” del objeto anónimo, pero no sería deseable).

Cabe destacar que ya que la recepción “queda corriendo” en el background del cliente y la llamada no será terminada hasta que el servidor así lo indique, sumado a que la ejecución del “onNext” sobre la cola de recepción se da de manera paralela a la ejecución del cliente, es posible utilizar esta metodología para implementar “pseudo-daemons” para las instancias de los clientes.

RPC Streaming del lado del cliente:

Similar a la unaria. En esta API, el cliente es el que envía un conjunto de mensajes al servidor en vez de una única solicitud.



```
/// Cliente ///
ClientStrMsg request = ClientStrMsg.newBuilder().build();

StreamObserver<ClientStrMsg> requestObserver = asyncResponse.clientStrAsync(new StreamObserver<ClientStrRes>() {
    @Override
    public void onNext(ClientStrRes response) {}

    @Override
    public void onError(Throwable t) {}

    @Override
    public void onCompleted() {}
});

for (int i = 0; i < 100; i++) {
    requestObserver.onNext(request);
}

requestObserver.onCompleted();
```



```
/// Servidor ///
@Override
public StreamObserver<ClientStrMsg> clientStrAsync(final StreamObserver<ClientStrRes> responseObserver) {
    return new StreamObserver<ClientStrMsg>() {
        @Override
        public void onNext(ClientStrMsg request) {}

        @Override
        public void onError(Throwable t) {}

        @Override
        public void onCompleted() {}
    };
}
```

El servidor responde con una única respuesta, junto al estatus y (opcionalmente) metadata, usualmente (pero no necesariamente) al terminar de recibir todos los mensajes del cliente.

Esta API sólo puede utilizarse mediante un stub asíncrono.

Puede apreciarse en los ejemplos que la funcionalidad respecto al streaming del lado del servidor es similar, con las diferencias siendo que el servidor responde al llamado del cliente con un objeto anónimo donde define su comportamiento ante cada mensaje, y el cliente define el comportamiento sobre la respuesta que dé el servidor en un objeto anónimo enviado durante la llamada.

También es el cliente quien maneja el “flujo” de la conversación, llamando a los métodos “onNext” y “onCompleted” del StreamObserver recibido.

RPC Streaming bidireccional:

El cliente inicia la operación invocando al stub y enviando su metadata, el método a ejecutar y en caso de haberlo el tiempo máximo de espera. El servidor puede elegir responder con su metadata, o esperar a que el cliente empiece a enviar mensajes.

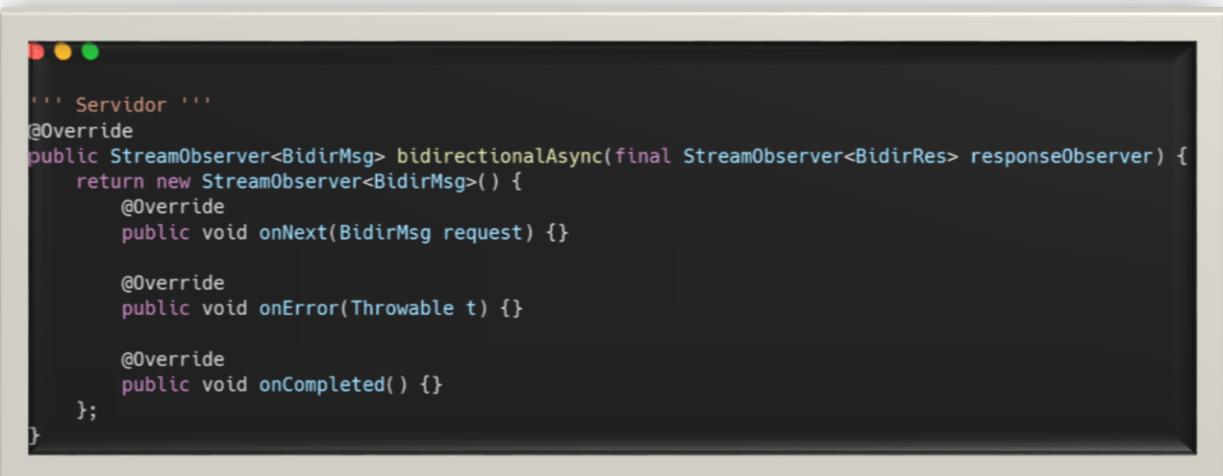


```
''' Cliente '''
BidirMsg request = BidirMsg.newBuilder().build();

StreamObserver<BidirMsg> requestObserver = asyncResponse.bidirectionalAsync(new StreamObserver<BidirRes>() {
    @Override
    public void onNext(BidirRes response) {}

    @Override
    public void onError(Throwable t) {}

    @Override
    public void onCompleted() {}
});
```



```
''' Servidor '''
@Override
public StreamObserver<BidirMsg> bidirectionalAsync(final StreamObserver<BidirRes> responseObserver) {
    return new StreamObserver<BidirMsg>() {
        @Override
        public void onNext(BidirMsg request) {}

        @Override
        public void onError(Throwable t) {}

        @Override
        public void onCompleted() {}
    };
}
```

Como los flujos de mensajes son independientes, tanto el cliente como el servidor pueden enviar o recibir mensajes del otro sin ningún orden específico (los mensajes siempre mantienen el orden en el que fueron enviados).

Por ejemplo, el servidor puede esperar a recibir todos los mensajes del cliente y luego enviarle los propios, o bien puede dar respuesta inmediata a cualquier mensaje que reciba.

El comportamiento específico de las interacciones se basa en la implementación dada a los métodos de la interfaz StreamObserver de ambas partes involucradas. A su vez, tanto el servidor como el cliente pueden manejar el flujo de la comunicación.

a) Sistema de pub/sub

El patrón publicadores-subscriptores es un patrón de mensajería asíncrono que cuenta de tres componentes:

- **los publicadores:** no conocen la cantidad ni la identidad de los subscriptores, solo publican mensajes en canales específicos.
- **los subscriptores:** reciben los mensajes publicados. Para ello, se suscriben a los canales para recibir los mensajes que sean publicados en los mismos.
- **el bróker:** informa a los subscriptores relevantes cuando un canal recibe una nueva publicación. Para implementar el patrón utilizando gRPC, se debe considerar que el bróker no es necesario, ya que su funcionalidad es suplida por los **StreamObservers** que formarán parte de los llamados al servidor.

Metodología:

- ✓ Los publicadores se conectan al servidor mediante una llamada Unaria bloqueante a un canal específico, en la cual envían el contenido de su publicación y reciben el estatus de la misma (en caso de no poder ser publicada, reintentar la operación).

Puede darse el caso que un publicador quisiera enviar contenido más grande que el máximo permitido por una llamada Unaria. En esos casos, lo correspondiente sería que la llamada fuese Client Streaming, para evitar que la información sea publicada de manera fragmentada e incompleta a los suscriptores del canal.

También puede ser deseable que los publicadores queden “suscritos” a su canal de publicación, para no deber re establecer la conexión en cada envío. En esos casos, la llamada debería ser del tipo Client Streaming y quedar guardada en el cliente una referencia al requestObserver recibido para realizar publicaciones en el canal.

Esta última solución implica que el servidor “sepa” cuando empieza y termina el stream relativo a una publicación en específico del cliente. Esto puede lograrse mediante lógica adicional del lado del cliente, definiendo mensajes con un descriptor que indique a qué publicación pertenecen y haciendo que el cliente mantenga el “hilo” de las mismas.

- ✓ Del lado de los subscriptores, la implementación deseable es mediante una llamada Server Streaming asíncrona al canal a suscribirse, definiendo la lógica necesaria en el onNext del StreamObserver enviado al servidor.

El servidor debería mantener colecciones con los responseObservers de los subscriptores, sobre las cuales broadcastearia las publicaciones realizadas al canal. Se debe garantizar que el uso de los mismos cumpla las condiciones de exclusión mutua.

El servidor también debe ser capaz de manejar casos en los que un suscriptor se desconectase inesperadamente. Esto puede resolverse utilizando una clase abstracta “ServerCallStreamObserver” como responseObserver, ya que la misma presenta el hook “setOnCancelHandler” el cual permite definir el comportamiento a ejecutar del lado del servidor en caso que el cliente asociado a dicho responseObserver se desconecte.

b) Sistema de archivos FTP

En un Sistema de archivos FTP se pueden dar multiplicidad de interacciones entre el cliente y el servidor, cada una de ellas siendo una posibilidad de accionar para el cliente.

A su vez, los archivos a subir o bajar pueden exceder el tamaño que permita enviar la API Unaria en un único mensaje.

- ✓ Por lo tanto, la API que mejor se ajusta al Sistema es la de Streaming Bidireccional, permitiendo esta enviar y/o recibir cualquier archivo al servidor, y a su vez mantener la funcionalidad esperada del mismo.

Sin embargo, la API Bidireccional presenta problemas de escalabilidad, al necesitar de mayores recursos físicos para cada cliente conectado. Si el Servidor fuese accedido por numerosos usuarios a la vez, entonces el mismo podría quedarse sin recursos para servir el tráfico e incurrir en la generación de tiempo de inactividad.

Bajo la misma lógica, el Servidor sería más vulnerable a los ataques DDoS.

- ✓ Otra alternativa a considerar sería la utilización de la API Unaria para la interacción con la interfaz del servicio, y luego la utilización de APIs de Streaming del cliente/servidor o Unaria para ejecutar las interacciones, según la operación elegida. (Por ejemplo, carga de archivos → Streaming Cliente; bajada de archivos → Streaming Servidor; renombrar archivo, listar archivos → Unaria).

Esta metodología tiene como ventaja la capacidad de utilizar llamadas asíncronas para las operaciones de streaming, dejándolas efectivamente en el “background” para que ambos cliente y servidor puedan continuar su interacción.

La desventaja es que requiere de un mucho mayor esfuerzo para manejar la concurrencia del lado del servidor para el manejo de actualizaciones de su contenido, de forma que múltiples clientes puedan ver reflejadas sus acciones al mismo tiempo.

Suponer el caso que dos clientes interactúen con el servidor, uno subiendo un archivo y el otro pidiendo el listado de los mismos. Si no se manejase bien, el segundo cliente no podría ver reflejado el “cambio” en proceso.

Incluso si se manejase bien, el segundo cliente podría elaborar un pedido de descarga/borrado/sobreescritura del archivo mientras el mismo está siendo subido.

Una solución a estos problemas es que ciertas operaciones del servidor operen sincrónicamente, pero esta solución podría incurrir en la reducción de la concurrencia, y a su vez hacer que el Servidor sea vulnerable a ataques DoS.

Otra solución posible sería que el servidor utilice un “marcador” para distinguir accesos a archivos en secciones críticas de clientes, y que el mismo maneje la exclusión mutua a ellos. Esta alternativa complejiza la lógica del servidor e introduce otras variables a tener en cuenta, como el orden de operaciones distribuidas (un ejemplo sería no permitir listar archivos con órdenes de borrado encoladas).

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 (nombre 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.

Se establece la precondición que todas las instancias de clientes definidas, tanto en el planteo teórico de la solución como en el testeo práctico, compartirán un mismo ManagedChannel para establecer la conexión con el Servidor.

Luego, se procede a desglosar la problemática a resolver en las siguientes partes:

CONEXION

El cliente realiza una solicitud al servidor, enviándole como mensaje su nombre de usuario y recibiendo el estatus de la operación. Dicha solicitud debería idealmente subscribir al cliente al chat. Por motivos de eficiencia y escalabilidad, se dividió la operativa de esta solicitud en dos distintas, llamadas “Probe” y “Join” respectivamente.

CONEXIÓN - JOIN

En el caso del “Join”, utilizar una llamada Bidireccional por cada cliente pondría estrés sobre los recursos del servidor, causando una posible degradación de la performance al escalar la cantidad de clientes conectados concurrentemente.

Por su parte, utilizar llamadas Unarias Asíncronas del lado del Servidor para transmitir las publicaciones a los clientes va en contra del modelo Cliente/Servidor, ya que el Servidor no debería ser el detonante de ninguna comunicación.

Finalmente se decide utilizar una llamada Server Streaming Asíncrona para lidiar con la suscripción.

El Servidor recibirá de cada cliente un mensaje con el StreamObserver que defina el comportamiento de su conexión al chat, y guardará el mismo en una estructura dinámica con capacidad de acceso concurrente (se eligió utilizar un ConcurrentHashMap, utilizando como llave el nombre de usuario recibido en el mensaje).

A su vez, se encontró durante el desarrollo la necesidad de asegurar exclusión mutua sobre cada Observer guardado.

Esto se debió a que al recibir múltiples publicaciones en simultáneo, el Servidor trata de recorrer el CHMap para notificar a los Observers utilizando distintos Threads. Se puede dar el caso que un Observer particular vea llamado su “onNext” varias veces al mismo tiempo, causando un error en gRPC.

Al recibir el Observer, el Servidor genera un ReentrantLock asignado al mismo, el cual es guardado sobre una segunda CHMap (utilizando el Observer como clave). Esto asegura la exclusión mutua sobre cada Observer, escalable a la cantidad de clientes conectados (se entrará en un superior detalle al respecto al describir el envío de mensajes).

Cabe destacar que uno de los requisitos explicitados es que el Servidor cuente con mecanismos de contención para las desconexiones inesperadas que pudiesen darse del lado de los clientes.

Para ello, los StreamObservers recibidos en el “Join” son casteados a ServerCallStreamObservers. Al hacer esto se pueden establecer CancelHandlers

sobre los mismos, donde se explicita el manejo en caso de suceder una desconexión espontánea.

CONEXIÓN - PROBE

Suponiendo que los nombres de usuario a utilizar sean relativamente razonables (no excedan el tamaño máximo de mensajes gRPC) y que el cliente deba esperar a recibir el estatus para continuar, se plantea modelar el “Probe” utilizando una llamada Unaria Bloqueante.

El Servidor debe recibir el nombre de usuario selecto del cliente, y retornar un estado correspondiente a la subscripción del mismo. Si el nombre es una clave del CHMap de Observers entonces retorna un estado exitoso representado mediante un valor true, caso contrario retorna el valor false.

Finalmente envía un mensaje de recibida al nuevo cliente conectado.

CONEXIÓN

Del lado del **cliente**, éste hace un llamado al “Join” con un objeto JoinRequest que contiene la implementación del comportamiento al subscribirse al chat (imprimir los mensajes entrantes) y el nombre de usuario deseado.

Luego itera realizando N llamados al “Probe” con su nombre de usuario, hasta recibir un true como respuesta. Cabe destacar que esta solución incurre en el uso de Busy Waiting y puede actuar como una parte distribuida de un DDoS en casos de extrema concurrencia de clientes.

DESCONEXION - LEAVE

Para el caso que **un cliente** desee desconectarse del chat se plantea la solicitud “Leave”, en la cual el cliente deberá enviar al servidor su nombre de usuario y esperar a recibir una confirmación del Servidor. Se utilizará de nuevo un estatus booleano para la confirmación, y la operativa del lado del cliente es idéntica al “Probe” ya detallado.

Del lado **del Servidor** se tomará el Lock del Observer correspondiente al usuario recibido y se removerá dicho Observer de la CHMap, guardándose la referencia al mismo.

Efectuado esto, el servidor utilizará la referencia para enviar un mensaje de desconexión al usuario, a su vez devolviendo y eliminando el Lock asociado. Finalmente el Servidor realiza un broadcast informando a los clientes conectados de la desconexión.

En caso de poder realizar estas operaciones el Servidor responderá con un estatus true al cliente, caso contrario responderá con un false.

El manejo definido para una desconexión espontánea es idéntico al ya descrito, exceptuando el mensaje de desconexión al usuario. Tampoco se toma ni devuelve el Lock del Observer, ya que según lo investigado el CancelHandler definido tiene una mayor “velocidad” o prioridad de ejecución.

ENVÍO DE MENSAJES

Para la publicación de mensajes se plantea la solicitud “SendMessage”.

Del ***Lado del Cliente***, el manejo de la misma es idéntico al ya descrito como solución al problema de los publicadores en el ejercicio previo, utilizándose la opción Unaria Bloqueante.

Esta decisión se basa en asumir que los clientes quieren esperar a que sus mensajes sean reflejados en el chat antes de enviar nuevos, y que los mensajes enviados serán de tamaño razonable. Si la primer suposición fuese falsa se utilizaría una llamada Asíncrona, mientras que si la segunda fuese falsa se utilizaría una llamada del tipo Client Streaming.

Del ***Lado del Servidor*** se deberá retransmitir el contenido del mensaje a todos los Observers almacenados. Para la confirmación de la operación se utilizará el sistema de estatus booleano ya descrito.

Al iterar sobre la CHMap de los Observers, el Servidor obtiene el Lock asociado a cada uno, envía el mensaje, y lo devuelve. Esto acota la sección crítica al objeto dentro de la colección, en vez de la colección misma. Se realizará un análisis sobre el impacto de esto sobre la concurrencia del programa en el Ejercicio 4.

HSTORIAL

Finalmente se plantea “GetHistory” como solicitud para que un cliente solicite un archivo de texto con el historial de mensajes.

Del lado ***del Servidor*** es necesario contar con un archivo actualizado que describa el historial a enviar. La escritura a dicho archivo debe ser tratada como sección crítica. Esto causa un posible cuello de botella en situaciones de tráfico elevado. Soluciones posibles a este problema se detallarán en el Ejercicio 4.

Se implementará una clase privada que actuará como servicio de escritura al historial al hacer un broadcast.

Al recibir la solicitud se deberá enviar por chunks el contenido del archivo solicitado. Para realizar esta operación se debería garantizar exclusión mutua sobre el archivo historial. Con el objetivo de minimizar la incidencia de la solución planteada sobre la concurrencia, se crea y envía una copia temporal del archivo a enviar, la cual es descartada al finalizar la solicitud.

Un problema que puede surgir de esta solución es el caso en que dos o más clientes soliciten el historial al mismo tiempo, ya que el archivo temporal en curso de ser enviado sería sobrescrito por las llamadas subsecuentes.

Esto se debe a que el nombre del archivo se encuentra definido de manera estática en el programa, y es resoluble generando dinámicamente un nuevo nombre de archivo para cada copia temporal que sea necesario crear.

No se incluye la implementación de esta solución en el código entregado, ya que probar su funcionamiento correcto complejizaba el desarrollo. Sin embargo, la implementación de la solución puede ser tan simple como que el servidor lleve un contador numérico de la cantidad de veces que el historial fue solicitado, actualizado en cada solicitud, y añada el mismo como prefijo al nombre del archivo temporal a crear.

Para el cliente se utiliza una llamada Server Streaming Asíncrona, ya que se asume que el mismo quiere seguir enviando mensajes al chat mientras se realiza la “descarga”. No se contempla el caso que un cliente se desconecte voluntariamente antes que finalice la operación.

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.

Por la forma en la que fue elaborado, el Servidor debería ser capaz de atender peticiones concurrentes de múltiples clientes. Para demostrar esto, se ajusta el manejo de la solicitud “SendMessage” de la siguiente forma:

```

1 private class ChatServerImpl extends ChatServiceGrpc.ChatServiceImplBase {
2
3     private final CountDownLatch concurrencyTestingLatch = new CountDownLatch(4);
4
5     ...
6
7     @Override
8     public void sendMessage(MessageRequest request, StreamObserver<ServerEvent> responseObserver) {
9
10        ...
11
12        /*
13         * First 4 messages from clients should be concurrent, since threads release
14         * from latch at around the same time, and go into race conditions
15         */
16        if (concurrencyTestingLatch.getCount() > 0) {
17            try {
18                concurrencyTestingLatch.countDown();
19                concurrencyTestingLatch.await();
20
21                Instant timestamp = Instant.now();
22
23                broadcastEvent(
24                    chatEventBuilder(request, timestamp),
25                    request.getUsername()
26                );
27            } catch (InterruptedException e) {
28                responseObserver.onNext(ServerEvent.newBuilder().setStatus(false).build());
29            }
30        } else {
31            broadcastEvent(
32                chatEventBuilder(request),
33                request.getUsername()
34            );
35        }
36
37        responseObserver.onNext(ServerEvent.newBuilder().setStatus(true).build());
38        responseObserver.onCompleted();
39    }
40
41    ...
42
43 }

```

Agregar este CountDownLatch sirve las veces de barrera, haciendo que el servidor procese el primer mensaje enviado por cada cliente al mismo tiempo, con el objetivo de simular concurrencia en el envío.

Al comparar los logs del Servidor con las salidas de cada cliente, puede verificarse que la presunción inicial es correcta.

LOGS DEL SERVIDOR:

Salida del chat del usuario Johnny:

Salida del chat del usuario Bárbara:

```
juan@...:~/Documentos/ej4$ cat Barbara_logs
server: Welcome!
2024-12-05T19:06:47.423Z] Barbara: Hello from Barbara
2024-12-05T19:06:47.423Z] Johnny: Hello from Johnny
2024-12-05T19:06:47.423Z] CGPT: Hello from CGPT
2024-12-05T19:06:47.423Z] Dave: Hello from Dave
2024-12-05T19:06:47.477Z] Barbara: Another message(1) from Barbara
2024-12-05T19:06:47.539Z] Dave: Another message(1) from Dave
2024-12-05T19:06:47.578Z] Johnny: Another message(1) from Johnny
2024-12-05T19:06:47.583Z] CGPT: Another message(1) from CGPT
2024-12-05T19:06:47.629Z] Johnny: Another message(2) from Johnny
2024-12-05T19:06:47.662Z] Barbara: Another message(2) from Barbara
2024-12-05T19:06:47.693Z] Dave: Another message(2) from Dave
2024-12-05T19:06:47.812Z] Barbara: Another message(3) from Barbara
2024-12-05T19:06:47.979Z] Johnny: Another message(3) from Johnny
2024-12-05T19:06:48.089Z] Dave: Another message(3) from Dave
2024-12-05T19:06:48.282Z] Barbara: Another message(4) from Barbara
2024-12-05T19:06:48.311Z] Server: CGPT disconnected
2024-12-05T19:06:48.317Z] Barbara: Another message(5) from Barbara
ISTORY DOWNLOAD DONE
2024-12-05T19:06:48.376Z] Dave: Another message(4) from Dave
2024-12-05T19:06:48.379Z] Barbara: Another message(6) from Barbara
2024-12-05T19:06:48.400Z] Barbara: Another message(7) from Barbara
2024-12-05T19:06:48.478Z] Johnny: Another message(4) from Johnny
2024-12-05T19:06:48.553Z] Johnny: Another message(5) from Johnny
2024-12-05T19:06:48.649Z] Dave: Another message(5) from Dave
2024-12-05T19:06:48.699Z] Johnny: Another message(6) from Johnny
2024-12-05T19:06:48.872Z] Barbara: Another message(8) from Barbara
2024-12-05T19:06:49.106Z] Dave: Another message(6) from Dave
2024-12-05T19:06:49.192Z] Barbara: Another message(9) from Barbara
2024-12-05T19:06:49.199Z] Johnny: Another message(7) from Johnny
2024-12-05T19:06:49.314Z] Johnny: Another message(8) from Johnny
2024-12-05T19:06:49.408Z] Dave: Another message(7) from Dave
2024-12-05T19:06:49.591Z] Johnny: Another message(9) from Johnny
2024-12-05T19:06:49.619Z] Barbara: Another message(10) from Barbara
2024-12-05T19:06:49.757Z] Dave: Another message(8) from Dave
2024-12-05T19:06:49.892Z] Dave: Another message(9) from Dave
2024-12-05T19:06:50.075Z] Johnny: Another message(10) from Johnny
2024-12-05T19:06:50.154Z] Dave: Another message(10) from Dave
2024-12-05T19:06:50.557Z] Server: Johnny disconnected
server: Goodbye!
```

Salida del chat del usuario Dave:

```
juan:~/Documentos/ej4$ cat Dave_logs
server: Welcome!
2024-12-05T19:06:47.423Z] Barbara: Hello from Barbara
2024-12-05T19:06:47.423Z] Johnny: Hello from Johnny
2024-12-05T19:06:47.423Z] CGPT: Hello from CGPT
2024-12-05T19:06:47.423Z] Dave: Hello from Dave
2024-12-05T19:06:47.477Z] Barbara: Another message(1) from Barbara
2024-12-05T19:06:47.539Z] Dave: Another message(1) from Dave
2024-12-05T19:06:47.578Z] Johnny: Another message(1) from Johnny
2024-12-05T19:06:47.583Z] CGPT: Another message(1) from CGPT
2024-12-05T19:06:47.629Z] Johnny: Another message(2) from Johnny
2024-12-05T19:06:47.662Z] Barbara: Another message(2) from Barbara
2024-12-05T19:06:47.693Z] Dave: Another message(2) from Dave
2024-12-05T19:06:47.812Z] Barbara: Another message(3) from Barbara
2024-12-05T19:06:47.979Z] Johnny: Another message(3) from Johnny
2024-12-05T19:06:48.089Z] Dave: Another message(3) from Dave
2024-12-05T19:06:48.282Z] Barbara: Another message(4) from Barbara
2024-12-05T19:06:48.311Z] Server: CGPT disconnected
2024-12-05T19:06:48.317Z] Barbara: Another message(5) from Barbara
2024-12-05T19:06:48.376Z] Dave: Another message(4) from Dave
2024-12-05T19:06:48.379Z] Barbara: Another message(6) from Barbara
2024-12-05T19:06:48.400Z] Barbara: Another message(7) from Barbara
2024-12-05T19:06:48.478Z] Johnny: Another message(4) from Johnny
2024-12-05T19:06:48.553Z] Johnny: Another message(5) from Johnny
2024-12-05T19:06:48.649Z] Dave: Another message(5) from Dave
2024-12-05T19:06:48.699Z] Johnny: Another message(6) from Johnny
2024-12-05T19:06:48.872Z] Barbara: Another message(8) from Barbara
2024-12-05T19:06:49.106Z] Dave: Another message(6) from Dave
2024-12-05T19:06:49.192Z] Barbara: Another message(9) from Barbara
2024-12-05T19:06:49.199Z] Johnny: Another message(7) from Johnny
2024-12-05T19:06:49.314Z] Johnny: Another message(8) from Johnny
2024-12-05T19:06:49.408Z] Dave: Another message(7) from Dave
2024-12-05T19:06:49.591Z] Johnny: Another message(9) from Johnny
2024-12-05T19:06:49.619Z] Barbara: Another message(10) from Barbara
2024-12-05T19:06:49.757Z] Dave: Another message(8) from Dave
2024-12-05T19:06:49.892Z] Dave: Another message(9) from Dave
2024-12-05T19:06:50.075Z] Johnny: Another message(10) from Johnny
2024-12-05T19:06:50.154Z] Dave: Another message(10) from Dave
2024-12-05T19:06:50.557Z] Server: Johnny disconnected
2024-12-05T19:06:50.618Z] Server: Barbara disconnected
server: Goodbye!
```

Salida del chat del usuario CGPT:

```
juan:~/Documentos/ej4$ cat CGPT_logs
server: Welcome!
2024-12-05T19:06:47.423Z] Barbara: Hello from Barbara
2024-12-05T19:06:47.423Z] Johnny: Hello from Johnny
2024-12-05T19:06:47.423Z] CGPT: Hello from CGPT
2024-12-05T19:06:47.423Z] Dave: Hello from Dave
2024-12-05T19:06:47.477Z] Barbara: Another message(1) from Barbara
2024-12-05T19:06:47.539Z] Dave: Another message(1) from Dave
2024-12-05T19:06:47.578Z] Johnny: Another message(1) from Johnny
2024-12-05T19:06:47.583Z] CGPT: Another message(1) from CGPT
2024-12-05T19:06:47.629Z] Johnny: Another message(2) from Johnny
2024-12-05T19:06:47.662Z] Barbara: Another message(2) from Barbara
2024-12-05T19:06:47.693Z] Dave: Another message(2) from Dave
2024-12-05T19:06:47.812Z] Barbara: Another message(3) from Barbara
ESCONEXIÓN ESPONTÁNEA
```

Si bien es cierto que el orden en el que los mensajes fueron enviados inicialmente no se mantiene, se obtiene un caso concurrente válido.

Del lado de la eficiencia, puede observarse en los logs del Servidor que la transmisión del conjunto inicial de mensajes a cada cliente fue completada en un margen de 0.004 segundos.

En parte este retraso se debe al retardo generado por llamar operaciones de entrada/salida para llevar los logs, sin embargo también se le debe atribuir parte de la culpa a las colisiones obtenidas sobre las secciones críticas generadas para cada Observer.

Como la concurrencia sólo se degrada al colisionar intentos de acceso a una sección crítica determinada, reducir la cantidad de colisiones mejoraría la performance del programa.

Debido a que la exclusión mutua está dada sobre cada Observer en particular, a medida que la cantidad de usuarios en el sistema fuese incrementando, las colisiones dadas sobre los mismos se vuelven más y más infrecuentes.

Por ello se concluye que esta parte del programa escala en grados de eficiencia y concurrencia de forma inversamente proporcional a la cantidad de usuarios del sistema.

A pesar de lo dicho se identifica un cuello de botella en la concurrencia del programa, ya que el Servidor maneja las escrituras al archivo historial de manera atómica, y escribe al mismo al instanciar cada mensaje a retransmitir.

Una forma de resolver esto es que el Servidor dedique parte de sus recursos a llevar el historial en una estructura concurrente, como también una variable que indique si el archivo historial se encuentra al día o no.

Al ser llamado “GetHistory” el Servidor primero volcaría el contenido de su variable historial en el archivo según sea necesario, y luego enviaría la copia del mismo. A su vez el Servidor debería volcar el contenido automáticamente antes de apagarse.

Esta solución abre las puertas a otros problemas posibles, tanto del lado de la escalabilidad (grandes cantidades de mensajes sin llamadas regulares a “GetHistory” por parte de los clientes) como de la consistencia (situación en la que el Servidor se apague inesperadamente y se pierda el contenido no guardado).

Se puede disminuir el impacto de ambas situaciones implementando lógica de detección de tiempo ocioso en el Servidor, para que se realicen volcados automáticos del contenido. A su vez se podría implementar lógica que determine el estrés de recursos, para contener los casos donde el Servidor no pueda encontrarse ocioso lo suficiente.

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.

El experimento diseñado consta que un cliente envíe un mensaje al servidor repetidas veces, midiendo los tiempos al enviar el mensaje y recibir la respuesta. El tiempo total de cada comunicación es dado por el valor absoluto de la diferencia entre ambos tiempos, sobre la cantidad de comunicaciones por llamada (2).

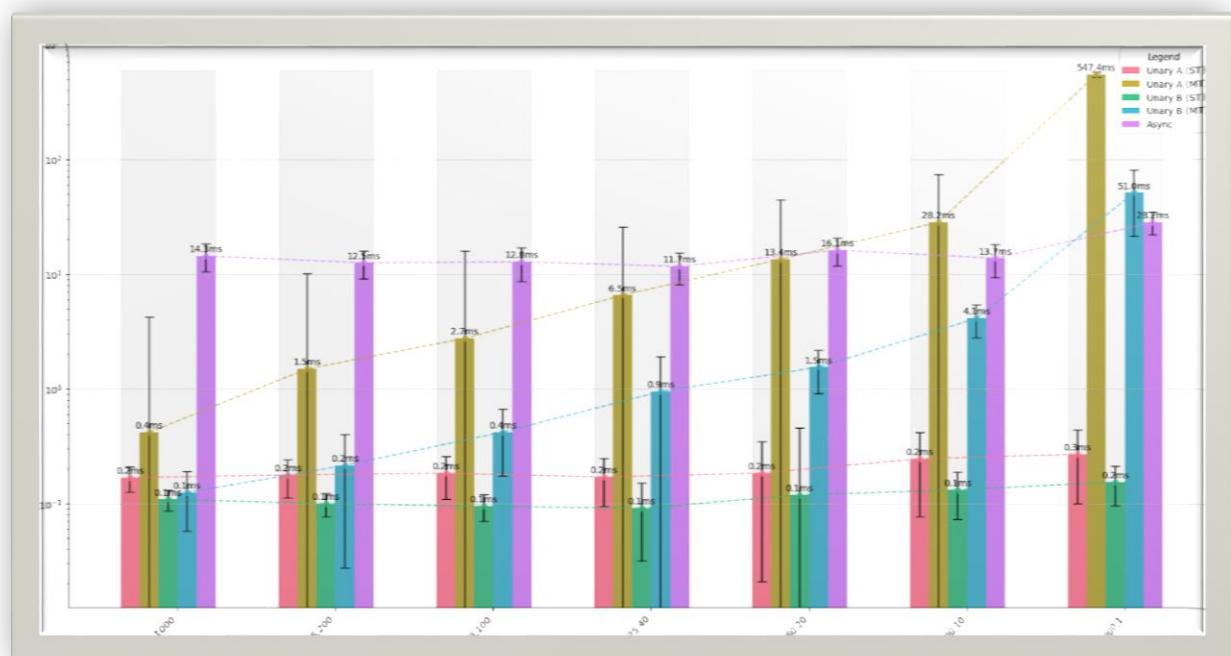
Es efectuado sobre una llamada Bidireccional Asíncrona (diseñada para que se comporte como una Unaria) y sobre otra Unaria Bloqueante. En el caso Unario el cliente evalúa dos maneras distintas para medir el tiempo inicial: tomarlo antes de hacer la llamada tomarlo dentro de la llamada (en la construcción del objeto mensaje).

Se evalúan tanto la ejecución secuencial de las llamadas como la ejecución paralela mediante Threads que se reparten la carga.

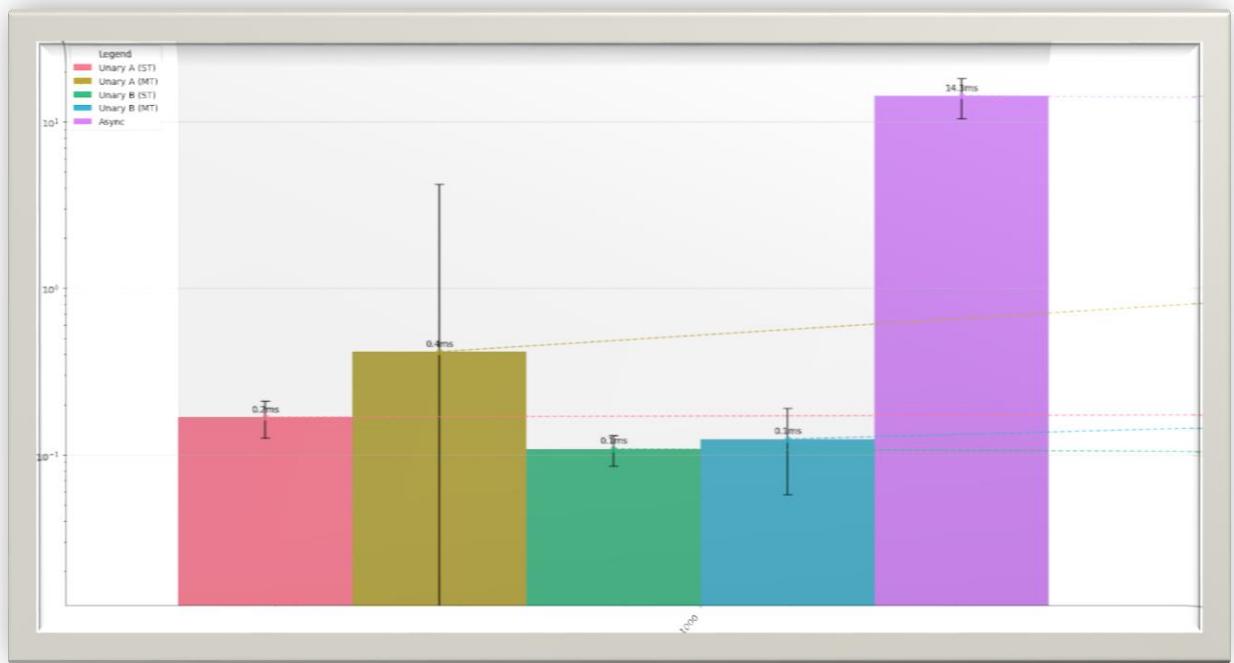
Como parámetros se evalúan 1, 5, 10, 25, 50, 100 y 1000 Threads ejecutando un total de mil llamadas (2000 mensajes) entre ellos, de la forma (cant_threads, llamadas_por_thread).

La ejecución secuencial siempre será constante y ejecutará dichas mil llamadas. Los resultados secuenciales obtenidos para cada ejecución se ven plasmados en los gráficos como consecuencia de simplificar su registro, sin embargo sirven para probar consistencia en los resultados obtenidos.

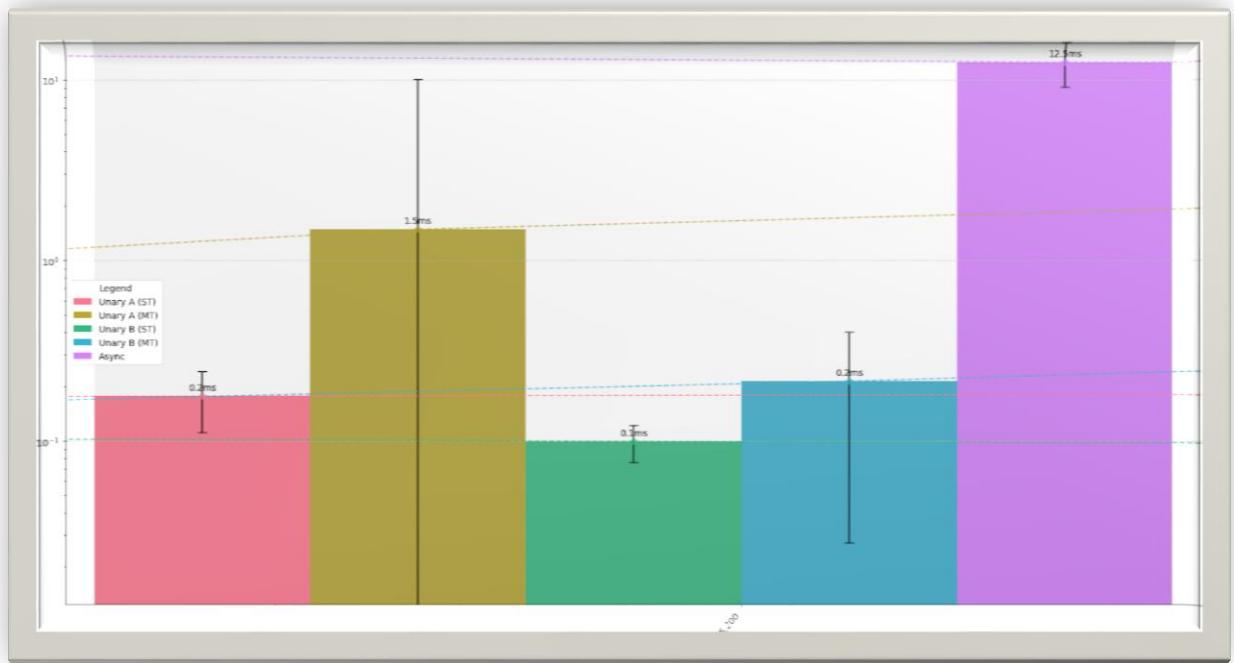
Gráficos de los resultados:



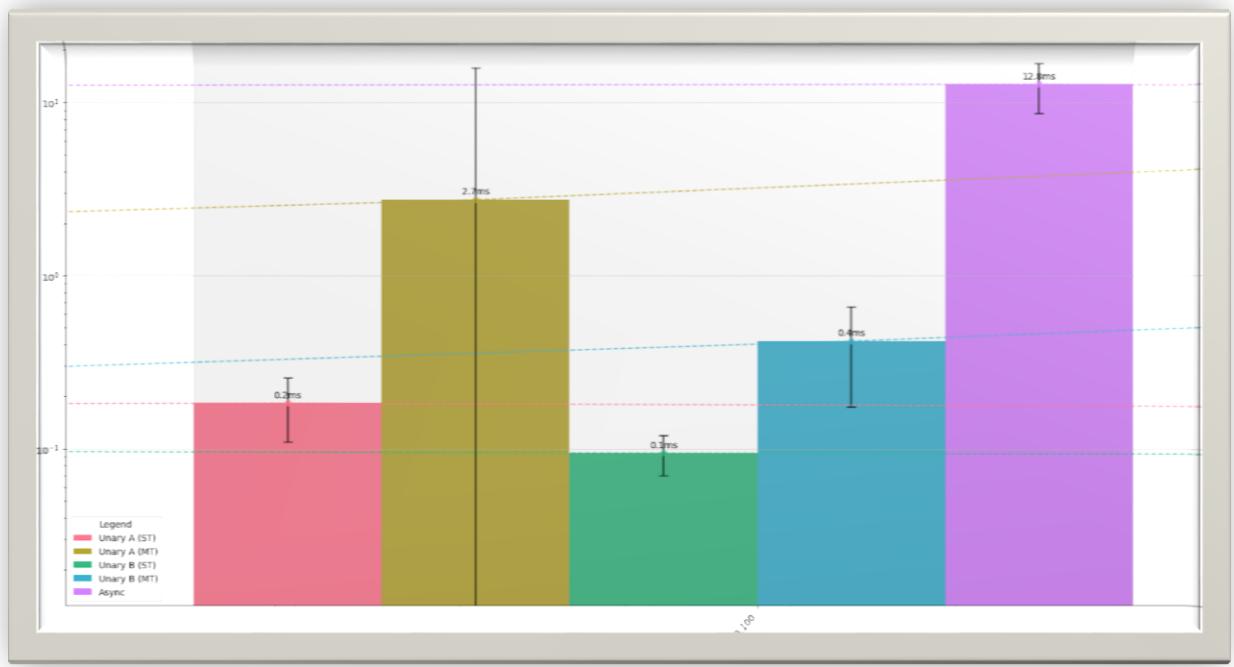
Resultados generales



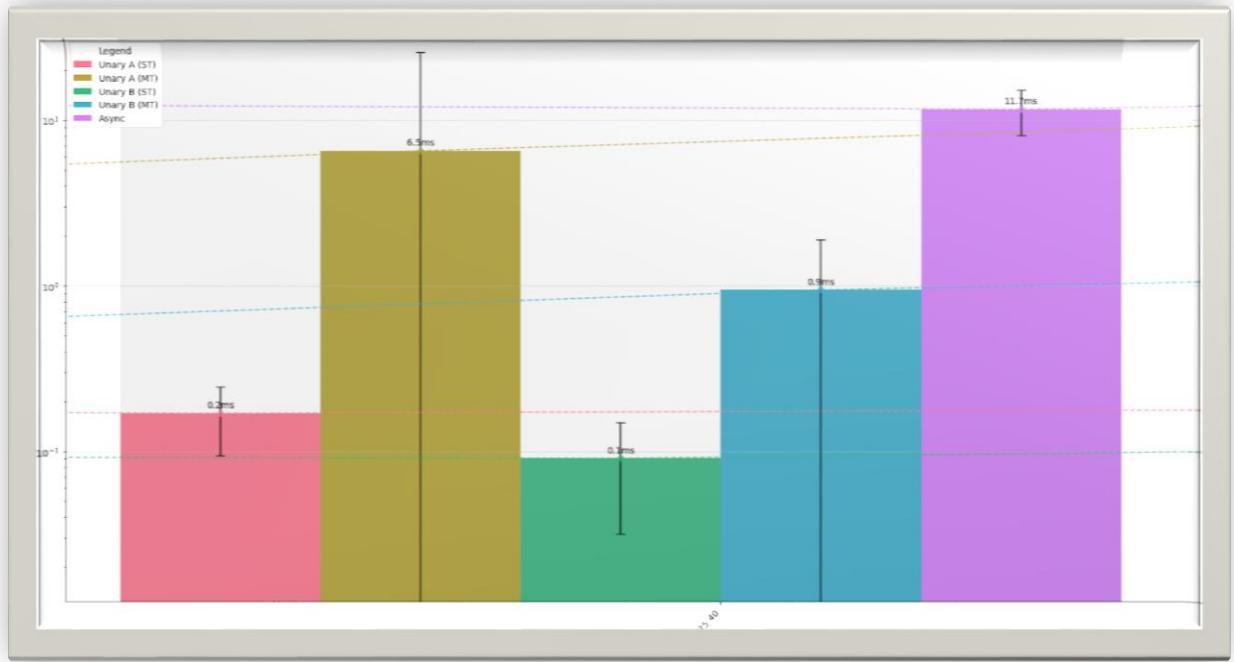
Acercamiento a los resultados de 1 Thread y 1000 llamadas por Thread



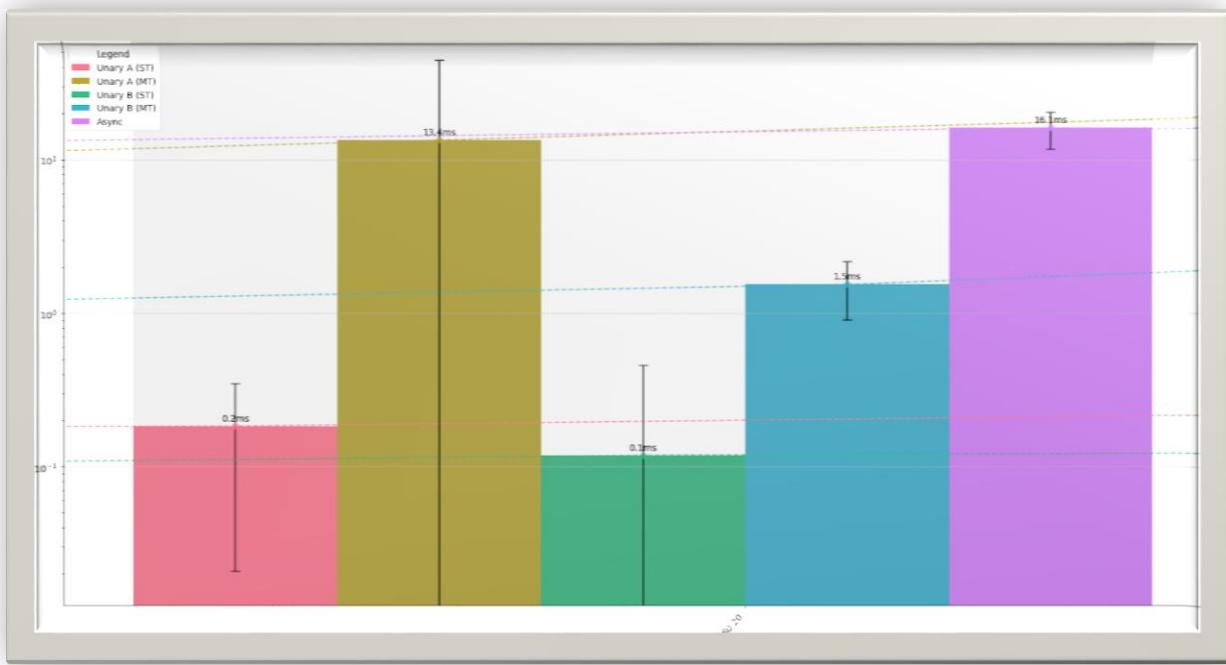
Acercamiento a los resultados de 5 Thread y 200 llamadas por Thread



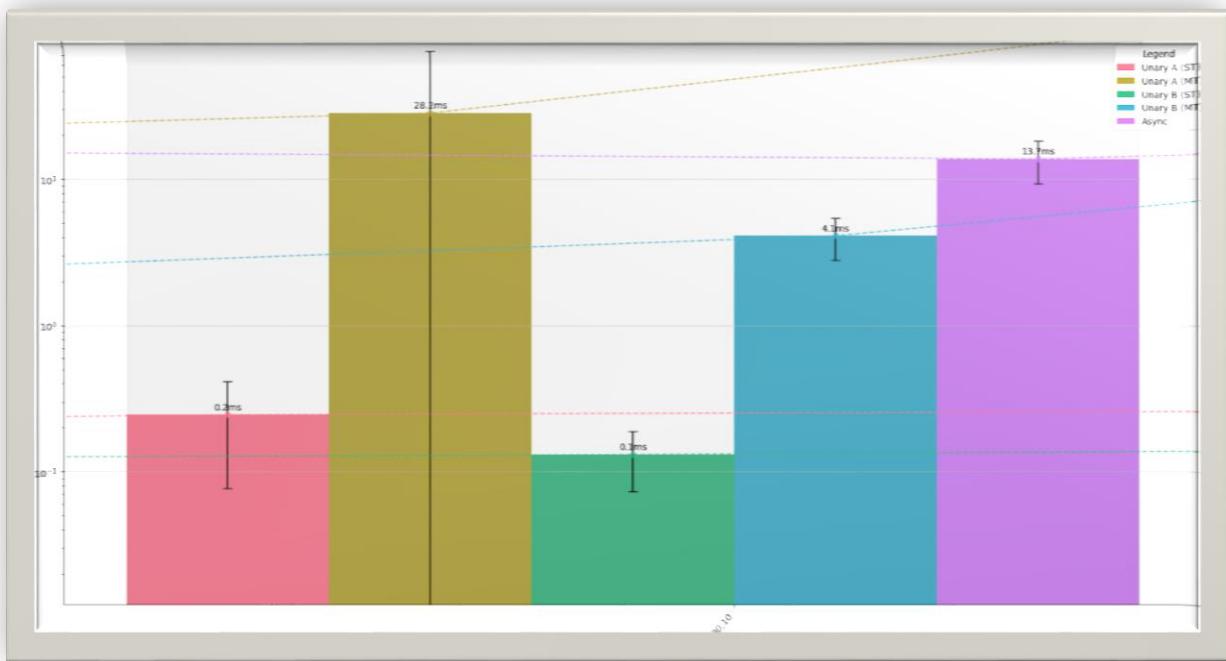
Acercamiento a los resultados de 10 Thread y 100 llamadas por Thread



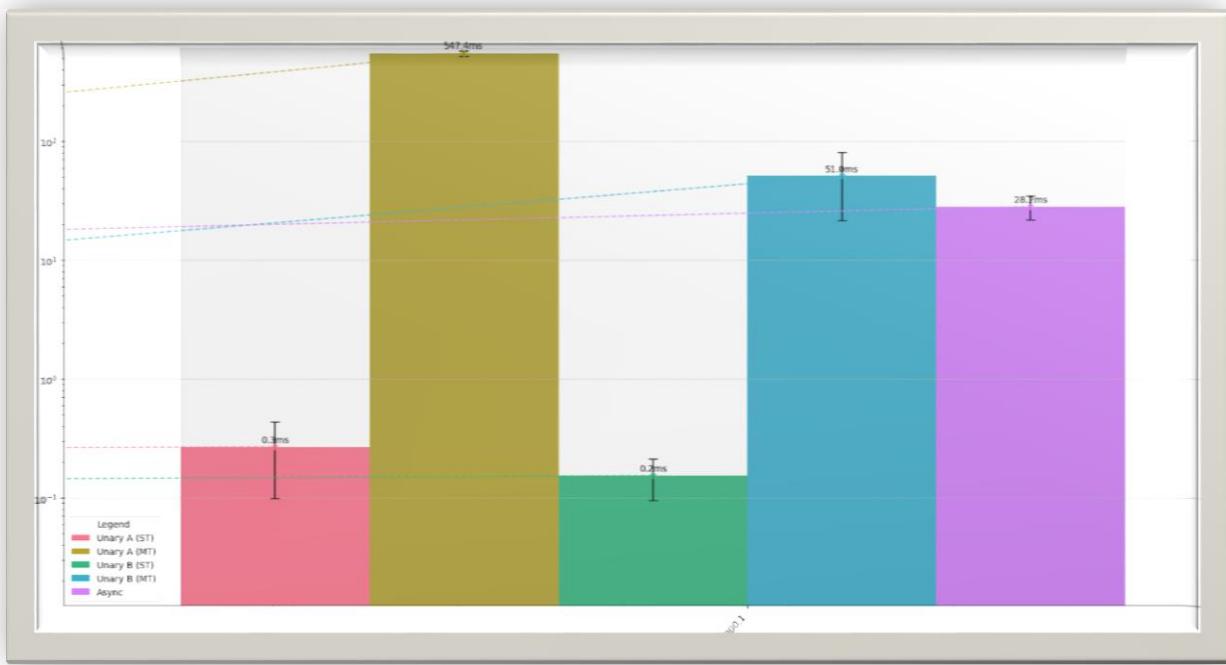
Acercamiento a los resultados de 25 Thread y 40 llamadas por Thread



Acercamiento a los resultados de 50 Thread y 20 llamadas por Thread



Acercamiento a los resultados de 100 Thread y 10 llamadas por Thread



Acercamiento a los resultados de 1000 Thread y 1 llamada por Thread

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.

Se parte de comparar el tamaño de los mensajes enviados en ambos experimentos, con la intención de delimitar un marco de referencia que permita determinar resultados conclusivos y aplicables a ambos paradigmas.

Debido a que los mensajes TimeStamp del experimento sólo envían un int64, y el tamaño de los mismos es de 8 bytes, se comparará solo con los resultados obtenidos para 10 bytes en la Práctica 1, asumiendo que los resultados obtenidos para 8b ≈ 10b.

Cabe destacar que los valores obtenidos en el desarrollo de la Práctica 1 fueron obtenidos sobre sockets en C.

A su vez, sólo se tendrá en cuenta las mediciones de los tipos UnarioA ST y Asíncronas obtenidas. Esto se debe a dos cosas:

Por una parte se descartan las mediciones MT, ya que claramente se observa que la performance del paradigma sufre proporcionalmente a la cantidad de Threads involucrados, y el trabajo de determinar si esto es causado por una sobrecarga de tráfico sobre el canal utilizado o (lo que intuiría verdadero causante) un mal manejo de los Threads por parte de la JDK excede al alcance de este informe.

Además se descarta el tipo UnarioB, ya que se asume que las mejoras obtenidas en las mediciones respecto al UnarioA se basan en que, probablemente, la JVM instancia

el mensaje a enviar mucho más “próximo” temporalmente al envío real del mismo sobre el canal. También podría deberse a retardos inducidos en la definición de una nueva variable.

De cualquier forma, el tipo UnarioA es una representación más cercana a la metodología utilizada para el muestreado hecho sobre Sockets.

Para la comunicación mediante Sockets se obtuvo un valor promedio de 0,000021 segundos, con una desviación estándar de 0,000013 segundos.

Para la comunicación mediante RPC se obtienen las siguientes mediciones:

- La UnariaA presenta un valor promedio $\approx 0,0002$ segundos, con una desviación estándar $\approx 0,00005$ segundos.
- La Asíncrona presenta un valor promedio $\approx 0,014$ segundos, con una desviación estándar $\approx 0,002$ segundos.

Claramente se puede observar una significativa reducción en la performance obtenida por los distintos paradigmas, como a su vez entre las distintas APIs del paradigma RPC.

La UnariaA presenta un incremento en el tiempo de retardo de un orden de magnitud, mientras que la Asíncrona (Bidireccional trabajada como si fuese Unaria) presenta un incremento de dos órdenes de magnitud.

Si bien es cierto que una parte considerable de este retardo puede darse a causa de las diferencias intrínsecas entre los lenguajes utilizados (comparar la performance de casi cualquier otro lenguaje con C, exceptuando casos como C++ y Rust, es plena y llanamente injusto) la diferencia es lo suficientemente amplia como para determinar que el paradigma RPC introduce un grado significativo de ineficiencia sobre la transmisión de las comunicaciones respecto Sockets. En especial al complejizarse la API utilizada.

Sin embargo, también es fácil señalar la ventaja que presenta utilizar RPC en un desarrollo, decrementando en gran medida costes de análisis y diseño. Si se utilizasen Sockets, toda la “infraestructura” necesaria ya implementada internamente en gRPC debiese ser definida y codificada.

También puede darse el caso en que al “adaptar” Sockets para este hipotético desarrollo, se reintroduzcan los mismos elementos internos que causan la diferencia en las mediciones obtenidas. O quizás incluso peor (no hay que olvidarse que gRPC fue desarrollado por Google).

De todo lo dicho, se concluye que no hay un paradigma idóneo entre ambos, y que la elección entre uno u otro depende tanto de la complejidad del problema a resolver, el grado de importancia de la eficiencia en la solución esperada, y la capacidad de los desarrolladores encargados de la elaboración.