

Estudio de gRPC como tecnología para implementar llamadas a procedimientos remotos

Pelayo Iglesias Manzano – UO266600

César Llano Rodríguez – UO289969

Pelayo Calleja Villafañe – UO292393

Martín Braña Peralvo – UO276268

Índice

1.Introducción	3
2.Fundamentos del RPC	3
2.1 Ejemplo básico de RPC	4
2.2 Características clave de RPC	5
2.3 Tecnologías RPC	6
3. ¿Qué es gRPC?	6
3.1 HTTP/2	7
4.Arquitectura de gRPC.....	7
4.1 Componentes principales.....	8
4.2 PROTOCOL BUFFERS	9
4.3 Flujo de una llamada gRPC	11
4.4 Tipos de llamadas	12
5.Ventajas y Desventajas de gRPC.....	13
5.1 Ventajas.....	13
5.2 Desventajas	14
6.Casos de Uso	14
6.1 Microservicios y Cloud-Native	14
6.2 Comunicación eficiente entre servicios internos	15
6.3 Aplicaciones en tiempo real.....	15
6.4 Empresas y tecnologías que usan gRPC	15
6.5 Interoperabilidad entre varios lenguajes	16
7.Comparativa con otras tecnologías.....	16
7.1 gRPC frente a REST	16
7.2 gRPC frente a GraphQL	17
7.3 gRPC frente a SOAP.....	17
7.4 Resumen en tabla	18
8.Conclusiones.....	18
Bibliografía	19

1.Introducción

En muchos sistemas actuales, las aplicaciones están formadas por varios servicios que trabajan juntos desde diferentes ordenadores. Para que estos servicios se comuniquen, se utilizan las llamadas a procedimientos remotos, conocidas como **RPC (Remote Procedure Call)**. Esto permite que un programa invoque funciones que se encuentran en otro equipo, como si fueran locales.

A lo largo del tiempo han existido varias tecnologías para implementar RPC, como CORBA o JAVA RMI, pero muchas de ellas resultan complejas o poco eficientes. Por eso, han surgido nuevas soluciones como gRPC, desarrollada por Google, que destaca por ser más rápida, sencilla y compatible con distintos lenguajes de programación.

Conocer gRPC es útil porque se trata de una herramienta actual, utilizada en muchos proyectos reales, que permite conectar servicios de forma eficaz y con buen rendimiento. Entender cómo funciona ayuda a diseñar mejores sistemas distribuidos y a elegir la tecnología adecuada en entornos modernos.

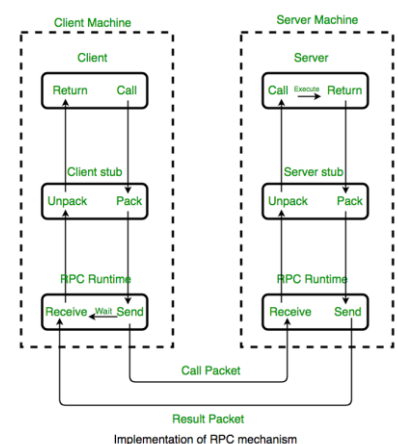
Este trabajo tiene como objetivo explicar qué es gRPC, cómo funciona y por qué es una buena opción para los sistemas distribuidos. Para ello, se plantean los siguientes objetivos específicos:

- Explicar qué es una llamada a procedimiento remoto (RPC) y por qué es útil en la comunicación entre servicios.
- Conocer las ventajas y desventajas de gRPC frente a otras tecnologías de RPC tradicionales.
- Describir cómo funciona gRPC, incluyendo sus componentes básicos y tecnologías asociadas, como HTTP/2 y Protocol Buffers.
- Incluir ejemplos prácticos para una mayor comprensión.

2.Fundamentos del RPC

RPC es un protocolo que permite a un programa informático ejecutar un procedimiento o función en otro ordenador o servidor. Con RPC, es posible llamar a funciones de ordenadores remotos como si fueran locales, esto facilita el desarrollo de aplicaciones distribuidas.

RPC funciona mediante un modelo **cliente-servidor** (como el esquema de la figura). El cliente inicia una petición al servidor, indicando específicamente el procedimiento que desea ejecutar y los parámetros necesarios para ello. La petición se envía a través de la red y el servidor la recibe, finalmente este localiza el



procedimiento solicitado, lo ejecuta y devuelve los resultados al cliente que los ha solicitado.

2.1 Ejemplo básico de RPC

Para comprender de mejor forma esta tecnología, hemos decidido incluir un ejemplo muy básico de uso de RPC con los comentarios pertinentes. El código se dividirá en:

- **Parte servidor**

```
// sumar_server.c
#include "sumar.h"

/* Esta función se ejecuta en el servidor.
   El cliente la llama remotamente pasando dos enteros. */
int *sumar_1_svc(int a, int b, struct svc_req *req) {
    static int resultado;
    resultado = a + b;
    return &resultado; // Se devuelve el resultado al cliente
}
```

- **Parte cliente**

```
// sumar_client.c
#include <stdio.h>
#include "sumar.h"

int main(int argc, char *argv[]) {
    CLIENT *cliente;

    int *resultado;

    int a = 4, b = 6;

    // Verifica que se introdujo la IP o nombre del servidor
    if (argc != 2) {
        printf("Uso: %s <servidor>\n", argv[0]);
        return 1;
    }

    // Establece conexión con el servidor RPC
    cliente = clnt_create(argv[1], SUMAR_PROG, SUMAR_VERS, "udp");

    if (cliente == NULL) {
```

```

        clnt_pcreateerror(argv[1]);

        return 1;
    }

    // Llama a la función remota 'SUMAR' enviando los dos enteros
    resultado = sumar_1(a, b, cliente);

    if (resultado == NULL) {
        clnt_perror(cliente, "Error al llamar al servidor");
        return 1;
    }

    // Imprime el resultado recibido del servidor
    printf("Resultado de %d + %d = %d\n", a, b, *resultado);

    clnt_destroy(cliente);

    return 0;
}

```

Antes de compilar cualquier código RPC, se debe generar el código “auxiliar” que permitirá la comunicación entre ambas partes, realizando la serialización y deserialización. Esto se hace utilizando `rpcgen`, y en este caso se aplicaría de esta forma:
rpcgen sumar.x

En el código anterior podemos ver el uso de diferentes comandos esenciales para el correcto funcionamiento de la tecnología RPC:

- **clnt_create**: Establece la conexión RPC con el servidor remoto.
- **resultado = sumar_1(a, b, clnt)**: Esta es la llamada remota al procedimiento RPC. Aunque parece una función local, en realidad envía los datos al servidor, espera la respuesta y la devuelve.
- **clnt_destroy**: Libera los recursos de la conexión RPC.
- **sumar_1_svc**: Es la implementación real de la función RPC en el servidor.

2.2 Características clave de RPC

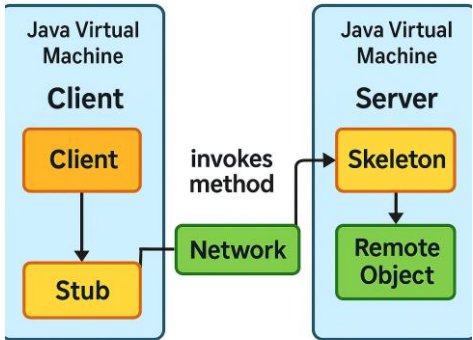
La tecnología RPC mantiene **transparencia de localización**, es decir, RPC oculta la ubicación del servidor al cliente. El cliente no necesita conocer la dirección física o de red del servidor, puede simplemente invocar una función en el servidor usando una

llamada a procedimiento remoto. A su vez hay que tener en cuenta la **transparencia de migración**, que implica que el cliente no necesita saber en qué máquina está realmente ejecutándose el procedimiento remoto que ha invocado. Siempre llama a la función de la misma forma, como si fuera local.

Gracias a herramientas como **rpcgen**, como hemos visto en el apartado anterior, se genera automáticamente el código de comunicación entre cliente y servidor, evitando que el programador tenga que manejar **sockets** o estructuras de red manualmente.

2.3 Tecnologías RPC

En este apartado vamos a hablar brevemente sobre algunas de las tecnologías RPC más importantes y destacadas. Estas son CORBA, JAVA RMI y .NET Remoting.

- **CORBA:** este estándar permitía a diferentes componentes software, escritos en diferentes lenguajes y corriendo sobre diferentes operativos y arquitecturas trabajar conjuntamente. Trajo varios problemas de incompatibilidad con lenguajes como C y C++. Ha caído en desuso actualmente.
- **JAVA RMI:** permite a los programas Java invocar métodos de objetos ubicados en otras máquinas dentro de una red como si fueran métodos locales. A diferencia de otras soluciones RPC que trabajan con funciones o procedimientos, Java RMI trabaja con objetos, permitiendo enviar referencias a objetos remotos y aprovechar las características de la programación orientada a objetos.
- **.NET Remoting:** es una tecnología desarrollada por Microsoft que permite la comunicación entre objetos en diferentes procesos o máquinas, dentro del entorno .NET. Aunque no se llama explícitamente “RPC”, funciona bajo el mismo concepto: invocar métodos remotos como si fueran locales, por lo que se considera una forma de RPC orientado a objetos (como JAVA RMI). Al igual que Java RMI, .NET Remoting fue pensado para funcionar exclusivamente dentro de su propio entorno, lo que facilitaba su integración, pero limitaba la interoperabilidad con otros entornos.

3. ¿Qué es gRPC?

gRPC es una implementación de llamadas a procedimiento remoto (RPC) diseñado originalmente por Google. Se emplea en comunicaciones cliente-servidor distribuidas por su eficiencia gracias a la ingeniería de procesos basada en RPC, pero surgió como una necesidad de modernizar las comunicaciones entre servicios distribuidos,

especialmente en entornos con alto rendimiento, escalabilidad y diversidad de lenguajes de programación.

3.1 HTTP/2

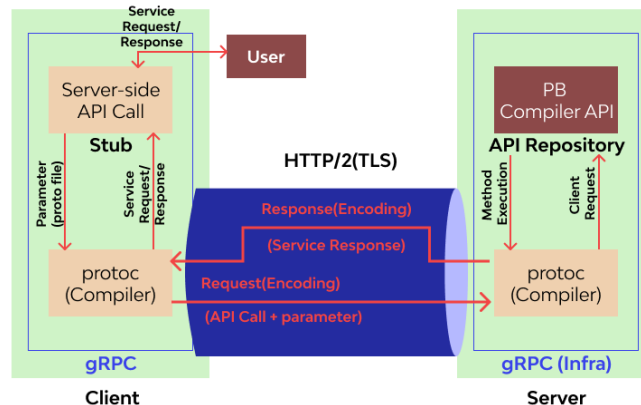
HTTP (Hypertext Transfer Protocol) es un protocolo de comunicación que permite la transferencia de información en internet. La versión más conocida es la HTTP/1.1, pero tiene algunas limitaciones. La más importante es que necesita una conexión TCP/IP por cada elemento que quiere obtener. Para superar esta limitación surge **HTTP/2**, que, aunque conserva la misma semántica que la versión HTTP/1.1, tiene numerosas mejoras. Las más importantes son:

- **Multiplexación:** con HTTP/1.1 para cargar cualquier elemento se requieren múltiples conexiones TCP, pero con HTTP/2 se utiliza **una única conexión**, a través de la cual se pueden enviar varias peticiones y recibir varias respuestas de forma simultánea, ya que la comunicación es **bidireccional**.
- **Compresión de cabeceras:** en HTTP/1.1 cada petición que se envía tiene un bloque formado por las cabeceras para indicar cierto comportamiento entre el cliente y el servidor. En HTTP/2 todas esas cabeceras se empaquetan en un único bloque comprimido de forma que se obtienen mejores tiempos de respuesta. Además, la compresión se realiza mediante el algoritmo HPACK, el cual se encarga de eliminar campos redundantes y prevenir posibles vulnerabilidades.
- **Formato binario:** Una de las claves del buen rendimiento de gRPC es que usa un formato binario para transmitir los datos, **en lugar de texto** como hacen otras tecnologías (por ejemplo, REST con JSON). Esto es posible gracias a que gRPC utiliza **Protocol Buffers (protobuf)** como sistema de serialización, de lo cual hablaremos más adelante. Cuando un cliente realiza una llamada a un servicio remoto, los datos enviados y recibidos se codifican en un formato binario muy compacto, permitiendo así una mayor velocidad, mensajes que ocupan menos espacio y menor uso del ancho de banda.

4.Arquitectura de gRPC

En gRPC, cada servicio cliente incluye un stub (archivos generados automáticamente), similar a una interfaz que contiene los procedimientos remotos actuales. El cliente gRPC realiza la llamada al procedimiento local al stub con los parámetros que se enviarán al servidor. El stub del cliente serializa los parámetros mediante el proceso de serialización mediante Protobuf y reenvía la solicitud a la biblioteca local del cliente en la máquina local.

El sistema operativo realiza una llamada al servidor remoto mediante el protocolo HTTP/2. El sistema operativo del servidor recibe los paquetes y llama al procedimiento stub del servidor, que decodifica los parámetros recibidos y ejecuta la invocación correspondiente mediante Protobuf. El stub del servidor envía la respuesta codificada a la capa de transporte del cliente. El stub del cliente recibe el mensaje de resultado y descomprime los parámetros devueltos, y la ejecución regresa al invocador.



4.1 Componentes principales

En este apartado, hablaremos sobre la arquitectura de gRPC, hablaremos sobre los clientes, los servidores, los stub y el service definition.

CLIENTE: El cliente gRPC es un componente de software que puede enviar solicitudes y recibir respuestas de un servidor gRPC. Una llamada gRPC es una llamada de procedimiento remoto (RPC) que utiliza buffers de protocolo como formato de mensaje.

gRPC cliente se puede escribir en cualquiera de los idiomas compatibles, como C#, Java, Python, Ruby, Go, etc. También puede utilizar bibliotecas clientes de gRPC para simplificar el proceso de desarrollo y aprovechar las características proporcionadas por gRPC, tales como autenticación, cifrado, balanceo de carga, etc.

SERVIDOR: es el componente responsable de recibir las llamadas remotas del cliente, ejecutar la lógica del servicio y enviar la respuesta de vuelta. Implementa los métodos definidos en el archivo .proto, se mantiene escuchando solicitudes entrantes desde clientes a través de una conexión HTTP/2, recibe los datos serializados en Protocol Buffers, los deserializa, ejecuta el código correspondiente y devuelve una respuesta.

STUB: es un código cliente o servidor generado automáticamente que actúa como intermediario entre el programa de aplicación y la red. Permite que el cliente llame a funciones del servidor como si fueran funciones locales, ocultando toda la complejidad de la comunicación remota.

En el cliente, el stub:

- Recibe las llamadas a métodos locales (hechas por el programa cliente).

- Empaqueta (serializa) los parámetros usando Protocol Buffers.
- Envía la solicitud al servidor a través de HTTP/2.
- Recibe la respuesta del servidor, la deserializa y devuelve el resultado al programa cliente.

En el **servidor**, el stub:

- Recibe la llamada del cliente.
- Desempaqueta (deserializa) los parámetros.
- Llama a la función real implementada en el servidor.
- Empaqueta la respuesta y se la envía al cliente.

SERVICE DEFINITION: es el contrato entre el cliente y el servidor. Es el lugar donde se especifica:

- Qué servicios existen.
- Qué métodos RPC pueden llamarse.
- Qué tipos de datos se intercambian.

Esta definición se escribe en un archivo especial de texto llamado `.proto` (Protocol Buffers)

Un service definition lo que contiene es:

1. El servicio
2. Los métodos RPC
3. Los mensajes

4.2 PROTOCOL BUFFERS

Protocol Buffers es un formato de serialización de datos desarrollado por Google para estructurar y serializar datos de manera eficiente. Se utiliza principalmente para la comunicación entre sistemas distribuidos, como en el caso de gRPC, pero también es utilizado para almacenar datos o cualquier otra tarea que implique la transferencia o almacenamiento eficiente de datos estructurados.

En `protobuf`, los datos se definen utilizando un lenguaje de esquema simple y legible llamado “Protocol Buffers Language” o “`proto`”. En este esquema, se definen las estructuras de datos que se van a serializar, incluyendo los tipos de datos y sus nombres. A partir de este esquema, se genera automáticamente código fuente en diferentes lenguajes de programación que permite serializar y deserializar los datos de manera eficiente.

Por ejemplo, supongamos que queremos definir un mensaje de usuario con un nombre y una dirección de correo electrónico utilizando *protobuf*.

El archivo de definición de esquema *proto* sería algo así:

```
syntax = "proto3";

message User {
    string name = 1;
    string email = 2;
}
```

Este esquema lo compilaremos usando el compilador de *ProtoBuf*: *protoc*. Para que nos devuelva un archivo para usar en Python ejecutamos lo siguiente:

```
protoc --python_out=. user.proto
```

Esto nos generaría un archivo, con un nombre similar a `user_pb2`, que podremos importar y usar en nuestro código:

```
import user_pb2

# Create user
user = user_pb2.User()
user.name = "John Doe"
user.email = "john.doe@example.com"

# Serialize user
serialized_user = user.SerializeToString()

# Deserialize user
deserialized_user = user_pb2.User()
deserialized_user.ParseFromString(serialized_user)

# Print name and email
print("Name:", deserialized_user.name)
print("Email:", deserialized_user.email)
```

Con *protobuf* no solo definimos los modelos de datos, sino que también definimos las interfaces de los servicios y los mensajes que se utilizan para comunicarse entre ellos.

Vamos a implementar un sencillo servicio que saluda a nuestros usuarios:

- Un mensaje `HelloRequest`, que recibe como parámetro un objeto de tipo `User`
- Un mensaje `HelloResponse`
- Un mensaje `User`
- Un servicio `Greeter` con un método RPC `SayHello`, que acepta como entrada un mensaje de tipo `HelloRequest` y devuelve otro de tipo `HelloResponse`

```

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloResponse) {}
}

message HelloRequest {
  User user = 1;
}

message HelloResponse {
  string message = 1;
}

message User {
  string name = 1;
  string email = 2;
}

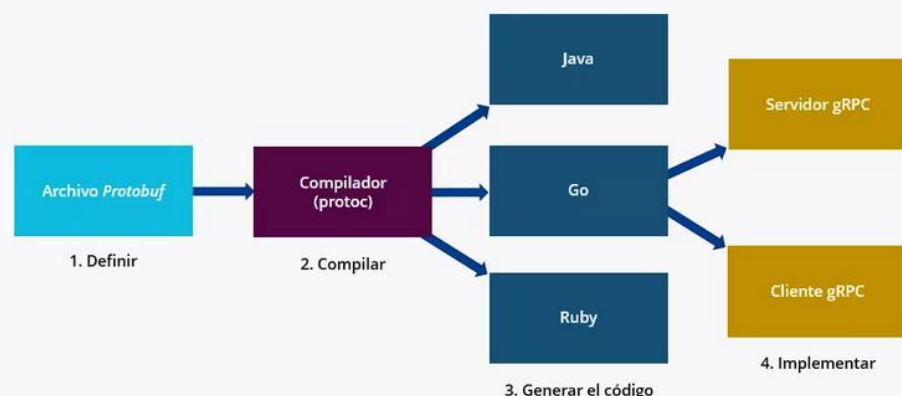
```

4.3 Flujo de una llamada gRPC

El flujo de trabajo gRPC se divide en cuatro pasos:

1. **Definición del contrato de servicio** para la comunicación entre procesos: se determinan los servicios que se deben aplicar y los parámetros y tipos de respuesta básicos a los que se puede acceder de manera remota.
2. **Generación del código gRPC** del archivo *.proto*: unos compiladores especiales (herramientas de líneas de comandos denominadas “protoc”) generan el código operativo con las clases correspondientes para un lenguaje de destino deseado (p. ej., C++, Java) a partir de los archivos *.proto* guardados.
3. **Implementación del servidor** en el lenguaje deseado.
4. **Creación del *stub* del cliente** que llama al servicio; a continuación, entre el servidor y el cliente se procesan las consultas.

Flujo de trabajo gRPC



4.4 Tipos de llamadas

En gRPC hay cuatro tipos de llamadas o modelos de comunicación según cómo se envían y reciben los mensajes entre el cliente y el servidor:

- **Unary RPC:** es el tipo más simple de RPC, donde el cliente envía una única solicitud y recibe una única respuesta
 1. Una vez que el cliente llama a un método del stub, el servidor es notificado de que se ha invocado la RPC, junto con los metadatos del cliente para esta llamada, el nombre del método y el plazo (deadline) especificado si aplica.
 2. El servidor entonces puede enviar inmediatamente sus propios metadatos iniciales (los cuales deben ser enviados antes de cualquier respuesta), o puede esperar a recibir el mensaje de solicitud del cliente. Qué sucede primero depende de la aplicación
 3. Una vez que el servidor tiene el mensaje de solicitud del cliente, realiza el trabajo necesario para crear y preparar una respuesta. Luego, la respuesta es enviada (si todo va bien) al cliente, junto con detalles del estado de la operación (código de estado y mensaje de estado opcional) y metadatos de cierre opcionales
 4. Si el estado de la respuesta es OK, entonces el cliente recibe la respuesta, lo que completa la llamada en el lado del cliente.
- **Server Streaming RPC:** es similar al Unary RPC, excepto que el servidor devuelve un flujo de mensajes en respuesta a la solicitud del cliente. Después de enviar todos sus mensajes, el servidor envía al cliente los detalles de estado (código de estado y mensaje de estado opcional) y metadatos de cierre opcionales. Esto completa el procesamiento en el lado del servidor. El cliente finaliza una vez que ha recibido todos los mensajes del servidor.
- **Client Streaming RPC:** también es similar al Unary RPC, excepto que el cliente envía un flujo de mensajes al servidor en lugar de un único mensaje. El servidor responde con un único mensaje (junto con los detalles de estado y los metadatos de cierre opcionales), generalmente, aunque no necesariamente, después de haber recibido todos los mensajes del cliente
- **Bidirectional Steaming RPC:** la llamada es iniciada por el cliente al invocar el método, y el servidor recibe los metadatos del cliente, el nombre del método y el plazo (deadline). El servidor puede elegir entre enviar inmediatamente sus metadatos iniciales o esperar a que el cliente comience a enviar mensajes en flujo. El procesamiento de los flujos, tanto en el lado del cliente como en el del servidor, depende de la aplicación. Dado que los dos flujos son independientes, el cliente y el servidor pueden leer y escribir mensajes en cualquier orden. Por ejemplo, el servidor puede esperar hasta recibir todos los mensajes del cliente

antes de enviar sus propios mensajes, o bien el servidor y el cliente pueden actuar en un modo tipo "ping-pong": el servidor recibe una solicitud, luego envía una respuesta, el cliente responde nuevamente basándose en la respuesta, y así sucesivamente

5.Ventajas y Desventajas de gRPC

5.1 Ventajas

La creación de gRPC y su posterior uso por parte de Google y de otras compañías importantes y usuarios se debe a que tiene una serie de ventajas:

- **Rendimiento:** Puede que sea el principal motivo por el que los usuarios puedan pensar en cambiar a gRPC, ya que proporciona velocidad y baja latencia a la hora de comunicar servicios cliente-servidor, por lo que mejora muchísimo los tiempos de respuesta y la experiencia de usuario. Además, la información viaja serializada y en formato binario, ocupando un espacio mucho mas pequeño que si fuera a través de un servicio REST mediante formato JSON. Esto beneficia mucho para uso de aplicaciones móviles donde la velocidad de conexión es menor y se puede incrementar mucho el tiempo de respuesta
- **Mejora en el streaming:** En este caso, gRPC permite todas las alternativas posibles para el streaming de información:
 - Sin streaming
 - Streaming unidireccional de servidor a cliente
 - Streaming unidireccional de cliente a servidor
 - Streaming bidireccional
- **Estrategia API First:** En la creación de API's es fundamental la creación de un API robusta y que permita la perfecta comunicación entre cliente y servidor, de forma que cada uno de ellos, disponga de toda la información que se debe mandar y que va a recibir. Aunque la estrategia API FIRST es algo opcional que ya puede ser utilizado en otras API's como puede ser REST, en el caso de gRPC es algo intrínseco y obligatorio desde el principio. Para la creación de servicios y mensajes, se define el API a través de un fichero proto (.proto) y es después cuando gRPC autogenera las clases bases de servicios y mensajes a partir de esa definición. De esta forma, se crea desde el primer momento un contrato entre cliente y servidor que ambos deberán cumplir para no romper el API.
- **Interoperabilidad:** gRPC es compatible con una amplia variedad de lenguajes de programación con generación de código incorporada, como C++, Java, Python, PHP, Go, Ruby, C#, Node.js, etc.
- **Seguridad:** gRPC proporciona autenticación conectable, seguimiento, equilibrio de carga y controles de estado para mejorar la seguridad y la resiliencia.

5.2 Desventajas

A pesar de que gRPC nos proporciona muchas facilidades y beneficios para su uso, también tiene una serie de desventajas que hacen que usarlo sea más complejo para algunos usuarios:

- **Legibilidad por parte del usuario:** A diferencia del API REST que utilizan el formato JSON para la transferencia de información mediante texto plano, gRPC usa ficheros binarios serializados por lo que por un lado es beneficioso, pero por otros es un inconveniente porque si se quiere utilizar esa información para hacer debug o detectar posibles problemas o errores en la comunicación entre aplicaciones
- **Limitación con navegadores:** Actualmente no es posible la comunicación directa entre los navegadores del mercado y gRPC, ya que gRPC utiliza el protocolo HTTP/2 y los exploradores actuales no proporcionan un nivel de control suficiente sobre este protocolo para admitir un cliente gRPC. Aun con todo esto, existen varios métodos para realizar la comunicación del navegador con el servicio gRPC:
 - **gRPC-Web:** tecnología suplementaria a gRPC que soluciona el problema de compatibilidad
 - **Crear API REST automáticamente desde servicios gRPC:** esto es posible mediante la inclusión de anotaciones de los ficheros de definición proto.

6.Casos de Uso

6.1 Microservicios y Cloud-Native

Uno de los principales casos de uso de gRPC es la comunicación entre microservicios, especialmente en arquitecturas cloud-native desplegadas en plataformas como puede ser Kubernetes, Docker.

Las arquitecturas cloud-native están diseñadas para mejorar aplicaciones tradicionales o hacer nuevos desarrollos con valores muy simples: más rápido, de mejor calidad y más baratas.

El uso de gRPC es una gran opción para la comunicación entre microservicios haciendo uso de esta arquitectura debido a su rendimiento. Con microservicios, una aplicación es desarrollada en pequeños servicios de manera totalmente independiente, permitiendo el uso de diferentes lenguajes de programación o plataformas.

Los servicios desarrollados mediante gRPC se caracterizan a diferencia de otros protocolos como REST por tener una menor latencia gracias al uso de HTTP/2, mayor eficiencia en la serialización de datos por el uso de Protocol Buffers y generación automática de código cliente y servidor.

6.2 Comunicación eficiente entre servicios internos

Muchas empresas utilizan gRPC para la comunicación interna entre componentes de sus sistemas, donde el rendimiento es un factor clave. A diferencia de REST, gRPC minimiza el tamaño de los mensajes y reduce la sobrecarga de red, lo que es útil para grandes volúmenes de tráfico.

Por ejemplo, Netflix implementó gRPC para gestionar el tráfico interno. Entre sus servicios de streaming, análisis de comportamiento de usuarios y sistemas de recomendaciones, necesitan miles de millones de llamadas diarias entre servicios.

Usar REST con JSON era demasiado pesado. Con gRPC, se redujo la latencia y el uso de CPU en los servicios internos.

6.3 Aplicaciones en tiempo real

Las aplicaciones en tiempo real que usan gRPC se benefician de la comunicación eficiente y de baja latencia que ofrece esta tecnología. Además de su soporte para streaming unidireccional y bidireccional. Algunos ejemplos serían:

- Aplicaciones de mensajería instantánea
- Juegos multijugador online
- Streaming de vídeo y audio
- Monitorización de sensores o métricas en tiempo real

Este modelo permite que tanto servidor como cliente mantengan un canal abierto para enviar y recibir datos de forma continua sin necesidad de realizar múltiples conexiones HTTP.

6.4 Empresas y tecnologías que usan gRPC

Diversas compañías tecnológicas han implementado gRPC para sus sistemas distribuidos.

Algunos ejemplos son:

- Google: gRPC fue desarrollado por Google y se utiliza internamente en la compañía además de en Google Cloud Platform (GCP) y en sus APIs públicas.
- Netflix: Gran parte de la comunicación interna de servicio a servicio en Netflix se ejecuta con gRPC.
- Dropbox, Square o Cisco, entre otras: han migrado de REST a gRPC para mejorar el rendimiento de sus aplicaciones.

6.5 Interoperabilidad entre varios lenguajes

Otro caso de uso relevante de gRPC es que permite la comunicación de servicios escritos en diferentes lenguajes de programación. Tiene soporte nativo para lenguajes como Java, Python, Go, C#, C++, etc. Al aportar un formato de serialización mediante Protocol Buffers, y herramientas para generar código en diferentes lenguajes, permite que servicios programados en distintos lenguajes puedan comunicarse entre sí.

7.Comparativa con otras tecnologías

La evolución de los sistemas distribuidos ha impulsado la creación de numerosas tecnologías orientadas a facilitar las llamadas a procedimientos remotos (RPC). Entre todas ellas, gRPC destaca por su eficiencia y su diseño adaptado a las necesidades actuales. No obstante, para comprender mejor su papel en el ecosistema tecnológico, resulta interesante compararlo con otras soluciones ampliamente adoptadas, como REST, GraphQL y SOAP.

7.1 gRPC frente a REST

gRPC es un framework de código abierto impulsado por Google, que combina HTTP/2 como protocolo de transporte con Protocol Buffers (Protobuf) para la serialización de datos. Gracias a esta combinación, permite una comunicación notablemente más rápida y eficiente que las implementaciones tradicionales basadas en REST.

REST, por su parte, se apoya en HTTP/1.1 y utiliza formatos como JSON o XML para el intercambio de información. Aunque su sencillez de implementación y su compatibilidad con navegadores han favorecido su enorme popularidad, el hecho de trabajar con formatos de texto introduce una sobrecarga que puede afectar al rendimiento frente a soluciones más modernas como gRPC.

Principales diferencias:

- **Protocolo:** HTTP/2 (gRPC) frente a HTTP/1.1 (REST).
- **Formato de datos:** Binario en gRPC (Protobuf) y textual en REST (JSON/XML).
- **Rendimiento:** gRPC logra una transmisión más rápida y eficiente.
- **Compatibilidad:** REST resulta más accesible para clientes ligeros, como aplicaciones web

7.2 gRPC frente a GraphQL

GraphQL, desarrollado por Facebook, es una tecnología que permite a los clientes definir de manera precisa los datos que requieren en cada solicitud. Es ideal para aplicaciones móviles y frontends complejos, donde se desea minimizar el número de llamadas al servidor.

Sin embargo, frente a gRPC, GraphQL muestra ciertas limitaciones en términos de rendimiento. Mientras que gRPC transmite datos en binario y aprovecha las capacidades de multiplexación de HTTP/2, GraphQL utiliza HTTP/1.1 y JSON, lo cual genera mayor sobrecarga de datos y latencias más elevadas.

Principales diferencias:

- **Consultas:** GraphQL permite consultas personalizadas; gRPC define interfaces más rígidas.
- **Transporte:** gRPC usa HTTP/2, GraphQL suele usar HTTP/1.1.
- **Enfoque:** GraphQL está pensado para frontends dinámicos; gRPC se orienta al backend y sistemas distribuidos.

7.3 gRPC frente a SOAP

SOAP (Simple Object Access Protocol) fue durante mucho tiempo el estándar para comunicaciones empresariales complejas. Se basa en el intercambio de mensajes XML a través de HTTP (u otros protocolos como SMTP) y ofrece características robustas de seguridad y transacciones distribuidas.

No obstante, su complejidad y su dependencia de XML suponen una carga importante para las comunicaciones modernas. Frente a esto, gRPC apuesta por un modelo más ligero y eficiente, ideal para arquitecturas basadas en microservicios y comunicaciones en tiempo real.

Principales diferencias:

- **Formato:** XML (SOAP) vs binario (gRPC).
- **Protocolos soportados:** SOAP es más flexible, mientras que gRPC está optimizado para HTTP/2.
- **Seguridad:** SOAP tiene soporte nativo para WS-Security; gRPC depende de TLS.
- **Caso de uso:** SOAP para entornos empresariales complejos; gRPC para sistemas distribuidos ágiles.

7.4 Resumen en tabla

Característica	gRPC	REST	GraphQL	SOAP
Protocolo de transporte	HTTP/2	HTTP/1.1	HTTP/1.1	HTTP, SMTP
Formato de serialización	Protobuf (binario)	JSON/XML (texto)	JSON (texto)	XML (texto)
Rendimiento	Alto	Medio	Medio	Bajo
Flexibilidad de consultas	Baja	Media	Alta	Baja
Complejidad de implementación	Media	Baja	Alta	Alta
Compatibilidad con navegadores	Limitada	Amplia	Amplia	Amplia
Uso principal	Microservicios, streaming	APIs públicas, Web Services	Frontends dinámicos	Entornos empresariales

8.Conclusiones

gRPC se ha convertido en una herramienta muy interesante para la comunicación entre servicios distribuidos. Su eficiencia, gracias al uso de HTTP/2 y Protocol Buffers, lo hace especialmente adecuado para aplicaciones que requieren bajo consumo de recursos, velocidad y buena escalabilidad, como en arquitecturas de microservicios o sistemas en tiempo real.

Aunque tiene algunas limitaciones, como la dificultad para trabajar directamente desde navegadores o la menor legibilidad de los mensajes, existen soluciones como gRPC-Web que permiten sortear esos obstáculos. En cualquier caso, es importante valorar cuándo tiene sentido usar gRPC frente a otras opciones como REST o GraphQL, ya que no siempre es la elección más sencilla ni la más adecuada para todos los contextos.

En definitiva, conocer gRPC y entender cómo funciona ayuda a tomar mejores decisiones a la hora de diseñar sistemas distribuidos, y aporta una alternativa potente y actual para los desarrolladores que buscan optimizar el rendimiento y la comunicación entre servicios.

Bibliografía

- https://www.lenovo.com/es/es/glossary/what-is-rpc/?orgRef=https%253A%252F%252Fwww.google.com%252F&srsId=AfmBOootAulqyXZ2hAg75FjYJC0_YQFvpn3uqsi5vFHh3cmReVWR4Hlk
- Apuntes Tema 2 de Sistemas Distribuidos
- <https://www.vpnunlimited.com/es/help/cybersecurity/remote-procedure-call>
- <https://es.wikipedia.org/wiki/CORBA>
- <https://electrocucaracha.com/2007/04/19/conceptos-basicos-acerca-de-net-remoting/>
- <https://www.paradigmadigital.com/dev/grpc-que-es-como-funciona/>
- <https://www.geeksforgeeks.org/graphql-vs-rest-vs-soap-vs-grpc/> Accedido el 27-4-2025
- <https://opensistemas.com/grpc-vs-rest-en-la-transmision-de-datos-eficiente/> Accedido el 27-4-2025
- <https://aws.amazon.com/es/compare/the-difference-between-grpc-and-rest/> Accedido el 27-4-2025
- <https://blog.postman.com/grpc-vs-graphql/> Accedido el 27-4-2025
- <https://www.wallarm.com/what/the-concept-of-grpc> Accedido el 29-4-2025
- <https://apidog.com/blog/grpc-client/> Accedido el 29-4-2025
- <https://grpc.io/docs/what-is-grpc/core-concepts/> Accedido el 29-4-2025
- <https://protobuf.dev/> Accedido el 29-4-2025
- <https://secture.com/que-necesitas-saber-sobre-grpc/> Accedido el 29-4-2025
- <https://www.ionos.es/digitalguide/servidores/know-how/que-es-grpc/> Accedido el 29-4-2025
- <https://grpc.io/docs/what-is-grpc/core-concepts/#rpc-life-cycle> Accedido el 29-4-2025
- <https://www.viewnext.com/que-es-el-grpc-ventajas-e-inconvenientes/> Accedido el 30-4-2025
- https://www.f5.com/es_es/glossary/grpc Accedido el 30-4-2025
- <https://grpc.io/docs/>
- <https://grpc.io/blog/>
- <https://adictosaltrabajo.com/2023/01/09/grpc-explicado-con-ejemplos-servidor-y-cliente/>
- <https://learn.microsoft.com/es-es/dotnet/architecture/cloud-native/grpc>
- <https://www.deloitte.com/es/es/services/consulting/blogs/todo-tecnologia/grpc-vs-rest-api.html>
- <https://auth0.com/blog/beating-json-performance-with-protobuf/>