



gRPC - Client - Server

Sistemas Distribuidos



Sistemas Distribuidos

Modelo de programación distribuido.



Sistema Distribuido - Origen

Tanto el **modelo multiproceso y el modelo de programación concurrente** son dos modelos que pretenden **paralelizar tareas dentro de un sistema.**

Estos modelos más allá de sus beneficios y problemas ya vistos tienen una cierta cantidad de limitantes dados por los recursos del sistema.

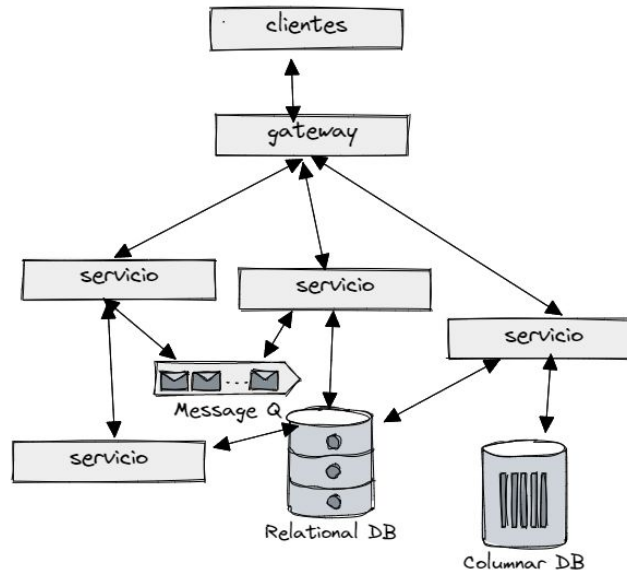
- **No se pueden correr concurrentemente más procesos de los que permite el procesador.**
- El número de **Threads** está limitado.
- La **memoria** también.
- Los procesos pueden llegar a competir por otros recursos compartidos como inodos, puertos, etc.

En algunos casos cuando se necesitan más recursos, potencia y/o eficiencia se puede utilizar el modelo distribuido.

Sistema Distribuido

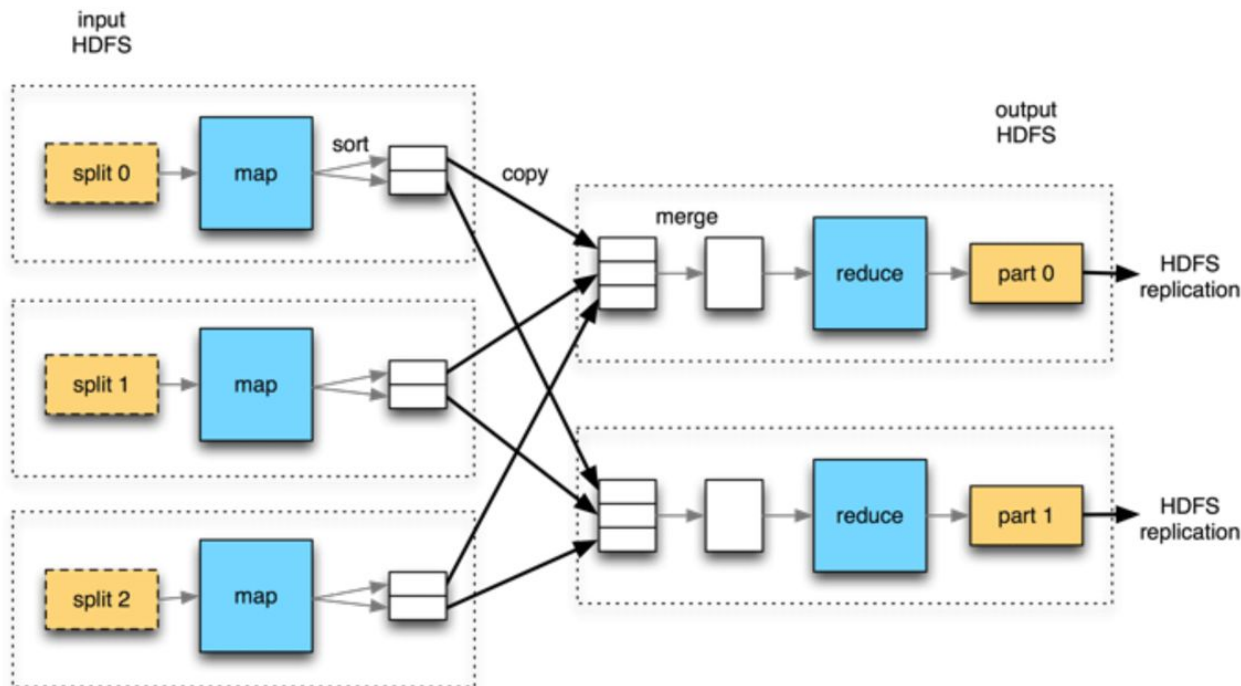
Es un sistema computacional cuyos **componentes** están **distribuidos en diferentes computadoras** (o elementos computacionales), que **se conectan y comunican por medio de mensajes de una red.**



Este sistema se presenta ante sus usuarios como un único sistema coherente.



Sistema Distribuido - Objetivo

Las componentes del sistema **dividen una tarea** coordinando sus recursos **para completarla de manera más eficiente** que si se ejecutara en una sola computadora.





**Servicios:
Cliente/Servidor**

El modelo cliente/servidor

El modelo cliente servidor se puede considerar como la **unidad atómica en la comunicación entre los nodos de un sistema distribuido.**

A pesar de que pueden ser **cientos o miles de máquinas** toda comunicación será entre **dos máquinas** una actuando como proveedor de un servicio y la otra como cliente del mismo.

La conexión es siempre uno a uno, sin importar la cantidad de nodos que tenga el servidor

La implementación más básica del modelo cliente/servidor es que el servidor es un socket en donde escucha, y el cliente escribe a ese socket

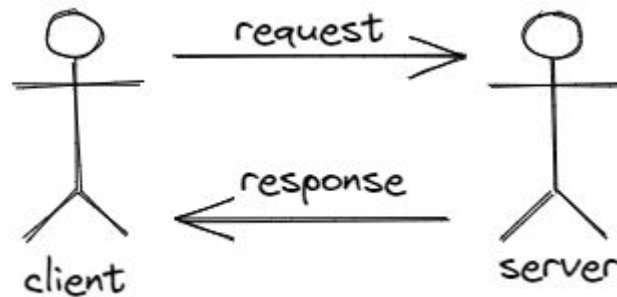
Servicio Ofrecido



Servicio requerido

¿Qué es un servicio?

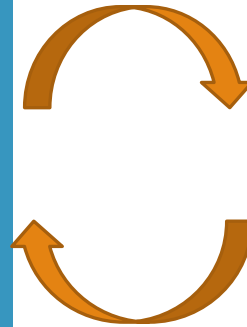
Parte de un sistema que **maneja recursos** y **presenta su funcionalidad a los usuarios** (personas o aplicaciones) **restringiendo sus posibilidades a un conjunto de OPERACIONES** definidas en una API.



Servicios - Roles

Servidor

Proceso en una red de computadoras que ejecuta las operaciones ofrecidas por la API del Servicio.



Cliente

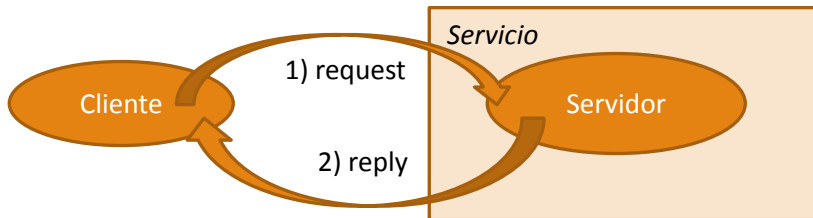
Proceso en una red de computadoras que solicita la ejecución de la operación de un servicio.

El término **cliente** y el término **servidor** sólo refiere al **ROL** que juega en un determinado momento de la existencia del servicio.

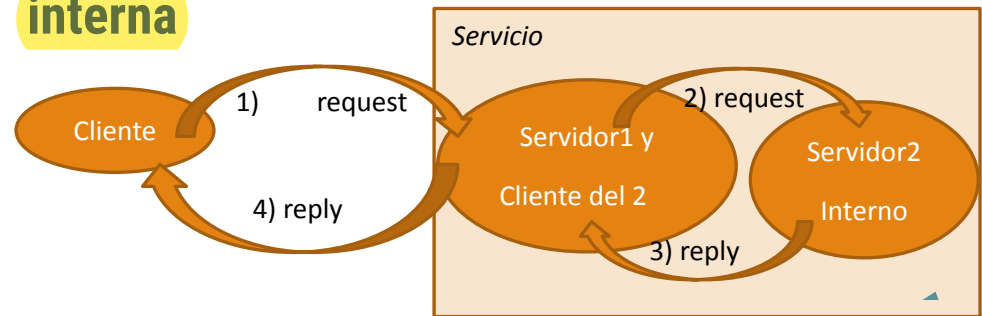
Cuando un cliente llama a un servicio que provee el servidor 1, este puede llamar por detras a un servicio en el servidor 2, convirtiendose en cliente

Servicios - Arquitecturas posibles

» Request/Reply a un único punto

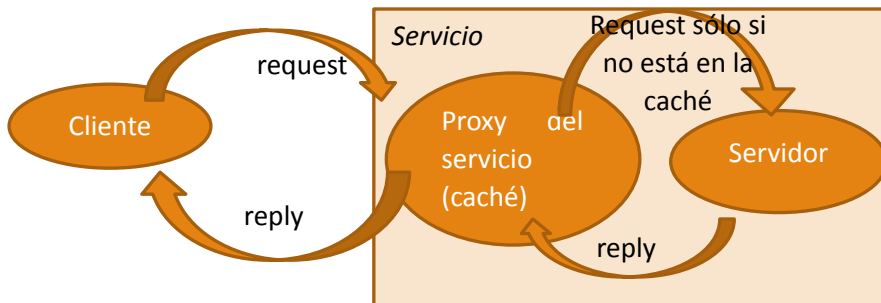


» Modelo Request/Reply con colaboración interna

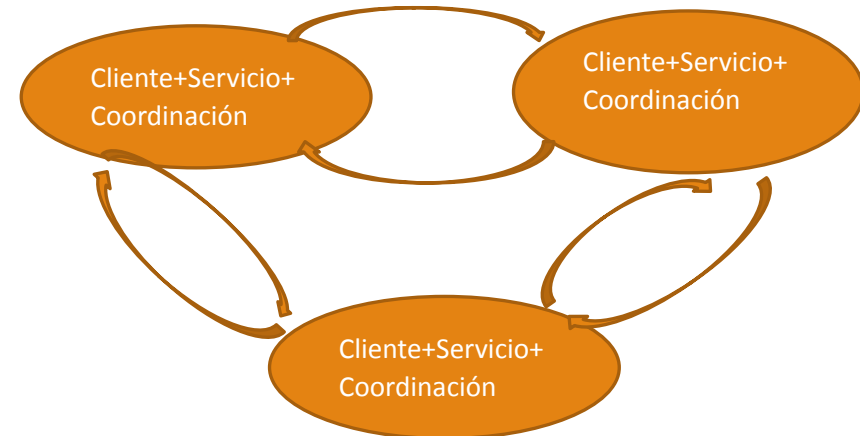


» Proxy y localidad de cache

El proxy podría ser un API gateway



» Procesos Peer: cuando la coordinación y comunicación se realiza simétricamente.



Y tantas otras topologías.



Servicios - Patrón de comunicación

La forma de implementar la comunicación entre cliente y servidor generalmente corresponden a dos estilos:

Request/Response

el estilo más común donde el cliente envía un pedido (request) y el servidor lo procesa sincrónicamente y una vez determinada la respuesta la envía hacia el cliente

Basado en Eventos

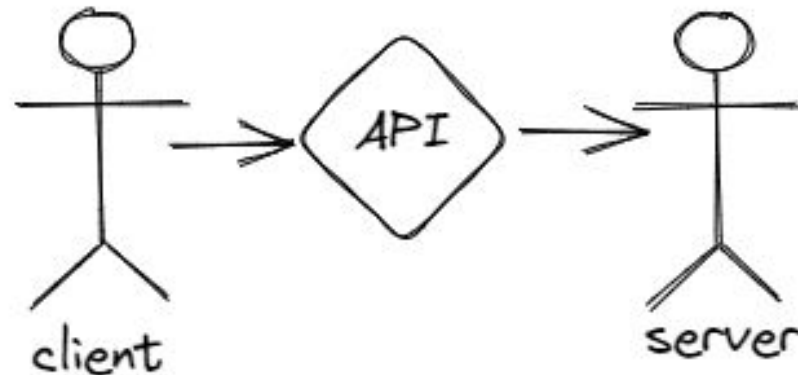
Es un tipo de comunicación asincrónica donde el cliente se registra para la recepción de cierto tipo de eventos y cuando estos ocurren el servidor se los envía.

La selección del estilo a usar dependerá mucho del tipo de servicio, cliente y tecnología.



Servicios - Interfaz/API

Una interfaz define un conjunto de operaciones que el servicio provee.






Servicios - Interfaz

La interfaz es el elemento más importante de un servicio ya que es el elemento que **le permite al cliente saber cómo invocar la operación que requiere.**

De cada operación su api/interfaz indica:

- modos de llamado
- nombres
- parámetros
- valores de respuesta
- errores.


Una vez que ya esta definida la interfaz, no importa en que lenguaje de programacion esta implementado el cliente o el servidor





Servicios - Interfaz

Una interfaz remota puede definirse usando:

- **Herramientas específicas del lenguaje/framework:** por ejemplo interfaces remotas de Java para RMI. RMI (remote method invocation) es el predecesor de gRPC
 - **IDL** (lenguaje de definición de interfaces): por ejemplo protobuf utilizado en grpc y otros.
 - **Documentación:** por ejemplo Apis Rest.
- 



IPC

Inter process communication





IPC - Comunicación entre procesos

¿Cómo programamos la comunicación entre 2 nodos?

Aplicaciones y Servicios

Sistemas Operativos

Computadoras y la Red

El **acceso a estos recursos** a nivel sistemas operativos y/o lenguajes de programación es **por medio de sockets**.






IPC - Mediante sockets

Aplicaciones y Servicios

Sistemas Operativos

Computadoras y la Red

- Para la comunicación a través de la red utilizamos TCP/IP y UDP/IP (punto a punto o multicast si es necesario). Entonces el **servicio** **está en** un nodo ubicado por su **dirección de IP** y el servicio en particular **escucha en un puerto** de dicho nodo.
 - **Abriendo un socket** entre 2 nodos se envían los mensajes de request y reply de forma binaria.
- 

La clase `ServerSocket` es para armar un socket para servidores

La clase `Socket` se usa para los clientes

`PrintWriter` es para mandar bytes

`BufferedReader` es para leer bytes. El metodo `.readLine` de `BufferedReader` es bloqueante, pero nos deja ir leyendo de a partes el mensaje a medida que van llegando

En el ejemplo, se usa el `'.'` como poison pill para matar al servidor

Ejercicio 1

- Bajar **socket-server.zip**.
- Correr `GenericSocketServer` y `GenericSocketClient` para ver como se establece la comunicación.

Servicios - IPC mediante a sockets

El servidor tiene que abrir sockets y escuchar los mensajes.

```
public void start(int port) throws IOException {
    logger.info("starting server on port {}", port);
    boolean loop = true;

    serverSocket = new ServerSocket(port);
    clientSocket = serverSocket.accept();
    out = new PrintWriter(clientSocket.getOutputStream(), true);
    in = new BufferedReader(new
        InputStreamReader(clientSocket.getInputStream()));
    String inputLine;
    while (loop && (inputLine = in.readLine()) != null) {
        logger.debug("received message {}", inputLine);
        if ((".").equals(inputLine)) {
            loop = false;
        } else if ("1".equals(inputLine)) {
            ++visitCount;
        }
    }
    out.println(visitCount);
}
```

Servicios - IPC mediante a sockets

El cliente se comunica y espera la respuesta.

```
public void startConnection(String ip, int port) throws
IOException {
    clientSocket = new Socket(ip, port);
    out = new PrintWriter(clientSocket.getOutputStream(),
true);
    in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
}

public String sendMessage(String msg) throws IOException {
    out.println(msg);
    String resp = in.readLine();
    return resp;
}
```

Ejercicio 2

- Modificar el server para utilizar el servicio armado en las clases anteriores. (el servicio del visitCount)



Middleware



Servicios - Capas y Middleware

Una de las ideas principales al **hacer** un servicio es que el **desarrollo** se haga lo **más transparente posible**.

El código para levantar Sockets se vuelve confuso y error prone, lo cual hace el código



Servicios - Capas y Middleware

Capa de comunicacion: SO, red

Por lo cual se intenta separar la capa de aplicación de la de comunicación mediante **una capa intermedia llamada middleware** que se encarga de abstraer a la aplicación de todo lo requerido para realizar la comunicación por medio de una red.

Estos middlewares generalmente vienen provistos por medio de frameworks estandarizados.






RPC



RPC

Remote procedure call: es un protocolo que especifica el llamado de ejecución de funciones remotas sin tener que preocuparse por la comunicación entre el cliente y el servidor.

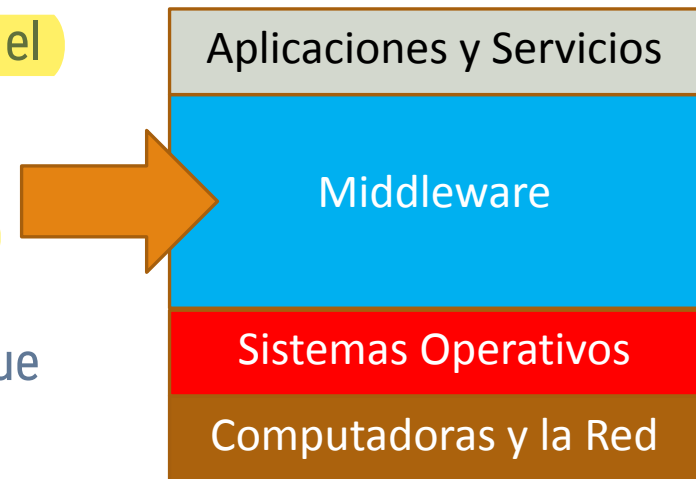


RPC

La idea es usar un RPC tanto en los clientes como en los servidores

RPC define que habrá un **middleware** que se encarga de:

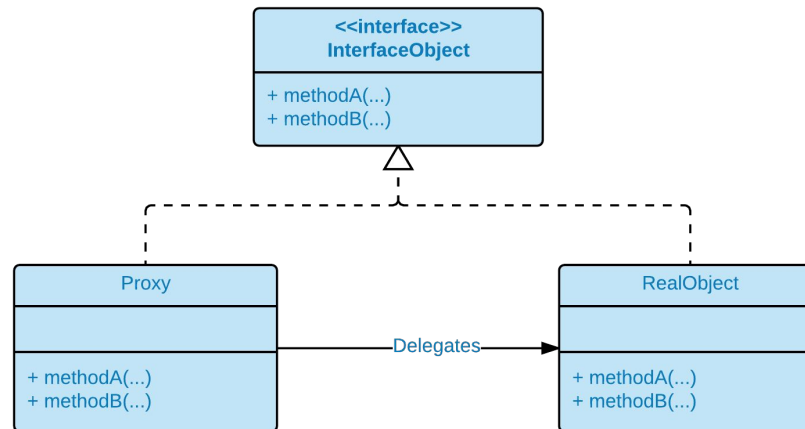
- En el cliente: **establecer la comunicación con el servidor, y traducir el llamado al método y parámetros a una invocación por la red.**
- En el servidor: **recibir la invocación, llamar al método correspondiente y transformar la respuesta** a una comunicación por red para que la reciba el cliente.
- **Además** de encargarse de todo lo relativo a errores y comunicación.



La intención de un framework como RPC es que las llamadas al servicio sean lo más “transparentes” posibles (o sea no se diferencien de llamados locales).

RPC - Middleware

Entonces estos elementos se encargan de la comunicación entre cliente y servidor aislando de los "problemas de comunicación" al cliente.



Efectivamente trabajando como un Proxy del servicio remoto.

Estas definiciones aparecen en los finales

Middleware definiciones

Stub:

Funciona de **proxy** a un objeto remoto **en el cliente**. Implementa los mismos métodos que la interfaz remota. Su **objetivo es recibir las invocaciones del cliente y pasarlas por la red** (el método invocado y los parámetros) y esperar los resultados para leerlos y transmitirlos al cliente.

El cliente cuando se quiere comunicar con el servidor, en realidad se está comunicando con el Stub.
El Stub se encarga de recibir estas invocaciones del cliente y redirigirlas al servidor, actuando como un proxy.

Middleware definiciones

Skeleton:

Se crea **en el servidor, se encarga de recibir el pedido del cliente**, interpretarlo, obtener los parámetros **y llamar al objeto remoto con los mismos**. Una vez que recibe la respuesta la transmite al cliente. Aislado de esta manera al objeto remoto de la forma de comunicación.

Recibe el pedido del cliente (que vino del stub), e interpreta el método que se quiere llamar. Finalmente, llama a ese método y le pasa los parámetros que recibió.

Tanto el stub como el skeleton se encargan de manejar las cosas engorrosas de la red (como errores y pasaje de parámetros).

Tanto el stub como el skeleton se autogeneran a partir de la interfaz que provea mi servidor.



gRPC

gRPC

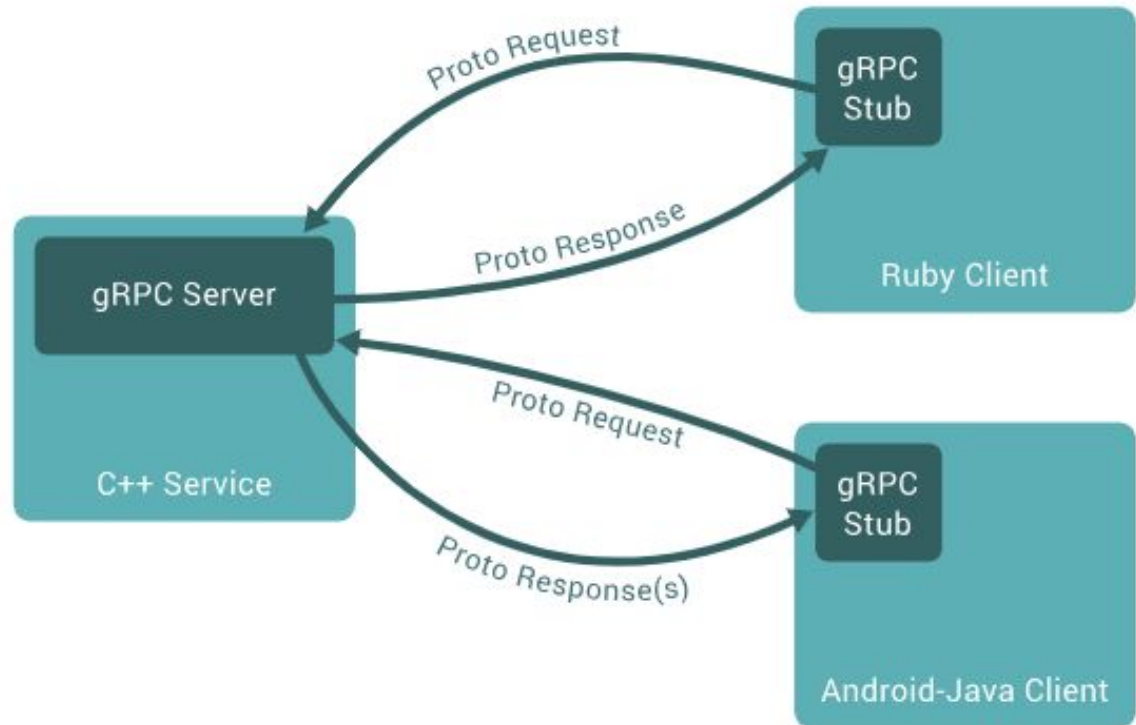


Una de las diferencias de gRPC y REST, es que no hace falta que gRPC sea human-readable (como los JSON de REST)

gRPC

gRPC es un sistema de comunicación vía RPC open source desarrollado inicialmente por Google que apunta a ser:

- **Multiplataforma**
- **Multilenguaje**
- **De alto rendimiento**
- **De uso general**
- **Open Source**





Protocol Buffers

Protocol Buffer es un mecanismo para serialización de datos que es independiente de lenguaje y plataforma

entonces al ser independiente del lenguaje es útil como IDL (lenguaje de definición de interfaces)

o sea definir servicios multilenguaje

Además por supuesto se puede utilizar como formato de serialización de la información que viaja entre el servicio y el cliente

(como sería JSON en REST por ejemplo)






gRPC

gRPC está basado en el uso de **Protocol Buffer** como IDL y formato de **serialización**, y **HTTP 2** para el transporte.

Esto permite que se pueda implementar features como:

- autenticación.
 - streaming bidireccional.
 - control de flujo.
 - timeouts.
 - cancelaciones.
- 

Protocol Buffers

Protocol Buffers es un formato de datos que permite serializar información estructurada:

Este es un ejemplo de archivo .proto

```
message Person {  
    int32 id = 2;  
    string name = 1;  
    repeated string email = 3;  
}
```

- Las **estructuras** pueden ser **message** ó **service**
- Ejemplos de tipos pueden ser **double, float, int32, int64, bool, string.**
- Existen también **soporte para tipos complejos**
- Los **modificadores** pueden ser **required, optional o repeated** (lista).
- El **número** es un identificador del campo dentro de la estructura.

El ID es obligatorio y tiene que ser unico

Protocol Buffers

La definición de mensajes y servicios se pasa por un compilador para generar el código en el lenguaje requerido

```
Person john =  
    Person.newBuilder()  
        .setId(1234)  
        .setName("John Doe")  
        .setEmail("jdoe@example.com")  
        .build();
```

Esta clase Person se genera automáticamente por el archivo .proto, y sigue el patron Builder

Particularmente en Java se genera la clase equivalente a cada mensaje y un Builder asociado para poder construir la instancia del mismo.

Ejercicio 3 - Proyecto grpc

1. Bajar desde campus *grpc-class-example.zip*.
2. Descomprimir el proyecto incorporar a la ide
3. Analizar la estructura y la definición del servicio en el proyecto API

Siempre que hacemos un servicio, hay que respetar la estructura:

- Cliente
- Servidor
- API




gRPC - Definición de Servicios

La definición de servicios se realiza en un archivo “nombre.proto”.
Antes del servicio se pueden customizar algunas características generales

```
syntax = "proto3";  
  
option java_multiple_files = true;  
option java_package = "ar.edu.itba.pod.grpc";
```

Estos son modificadores opcionales, en los cuales podemos pedir que los nombres de clase no sean tan largos, y que se generen en multiples lugares



gRPC - Definición de Servicios

El servicio se define a partir de la palabra reservada “service” y se listan sus métodos.

```
// The greeter service definition.  
service Greeter {  
    // Sends a greeting  
    rpc SayHello (HelloRequest) returns (HelloReply);  
}
```

Todos los métodos se prefijan con rpc tienen un mensaje de entrada y un mensaje de salida

Tengo el metodo SayHello que recibe un HelloRequest y devuelve un HelloReply. Un requisito es que todos los metodos reciben un parametro y devuelven un parametro. Si no quiero recibir nada hago un message vacio. Si quiero recibir muchas cosas me hago un message con multiples messages.

Tambien se pueden importar messages definidos por Google, como el message vacio

gRPC - Definición de Servicios

Para terminar de definir el servicio se agregan los mensajes de entrada y salida

```
// The request message containing the user's name.  
message HelloRequest {  
    string name = 1;  
}  
  
// The response message containing the greetings  
message HelloReply {  
    string message = 1;  
}
```

El mensaje debe existir aun si está vacío.



Ejercicio 4 - Definición del servicio

- 1. Compilar usando maven.**
 - 2. Revisar las clases generadas en la carpeta target del proyecto API.**
- 




gRPC - Java generación de código

En Java (como para todos los lenguajes) existe un compilador que recibe al compilar el archivo *.proto con la definición del servicio se obtiene una clase llamada:

<NombreDeServicioGrpc> (GreeterGrpc)

La misma funciona como middleware y provee unas subclases que utilizan el servidor y el cliente para realizar la comunicación entre ellos.


Además se generarán clases por cada mensaje de entrada y salida.





gRPC - Client

Para uso en el cliente el **Middleware** provee 3 versiones de stub para realizar llamados a los servicios del server.


- **<Servicio>BlockingStub:** Stub **sincrónico**
 - **<Servicio>Stub:** Stub **asincrónico** con observers para procesar
 - **<Servicio>FutureStub:** Stubs asincrónico que **retorna Futures**
- 



gRPC - Client

Cada uno de ellos se instancia a partir de un builder y tiene implementado los métodos del servicio variando sus parámetros y respuesta de acuerdo al tipo de llamado que realizan.

También estos stubs proveen de métodos para proveer a cada llamado:

- credenciales de autenticación
 - timeouts
 - control de flujo
 - Executor e interceptors
- 

gRPC - Client

Para consultar el servicio desde el cliente se instancia el stub a utilizar (pasando un channel).

```
public class HelloWorldClient {  
    private final GreeterGrpc.GreeterBlockingStub blockingStub;  
  
    public HelloWorldClient(Channel channel) {  
        blockingStub = GreeterGrpc.newBlockingStub(channel);  
    }  
  
    public void greet(String name) {  
        HelloRequest request =  
            HelloRequest.newBuilder().setName(name).build();  
        HelloReply response;  
        try {  
            response = blockingStub.sayHello(request);  
        } catch (StatusRuntimeException e) {}  
    }  
}
```

El channel recibe la IP y el puerto, que es el unico parametro necesario para acceder a la red. Es una abstraccion sobre un socket.

gRPC - Client

Para **inicializar el cliente** se genera el channel con los datos de conexión y crean los objetos client que wrapean los llamados al servicio

```
public static void main(String[] args) throws Exception{
    String user="world";
    String target="localhost";
    Integer port=50051;

    ManagedChannel
    channel=ManagedChannelBuilder.forAddress(target,port)
        .usePlaintext().build();

    try{
        HelloWorldClient client=new HelloWorldClient(channel);
        client.greet(user);
    }finally{
        channel.shutdownNow().awaitTermination(5, TimeUnit.SECONDS);
    }
}
```

Si estoy en un stub bloqueante, meto un timeout



Ejercicio 5 - Implementar Client

- 1. Agregar las clases clientes y correr contra el servicio indicado.**
- 

gRPC-Servant

Para poder implementar el servicio se utiliza la clase

Si no overrideamos esto, tira unimplemented exception

<NombreDeServicioImplBase> (**GenericServiceImplBase**)

Se debe extender la misma y overridear los métodos definidos por la api.

```
class GreeterImpl extends GreeterGrpc.GreeterImplBase {  
  
    @Override  
    public void sayHello(HelloRequest req, StreamObserver<HelloReply>  
        responseObserver) {  
        HelloReply reply = HelloReply.newBuilder().setMessage("Hello " +  
            req.getName()).build();  
        responseObserver.onNext(reply);  
        responseObserver.onCompleted();  
    }  
}
```

El **StreamObserver** recibido como parámetro se utiliza para enviar respuestas vía **onNext** y al terminar se utiliza el **onCompleted**.

Yo puedo dar multiples respuestas o respuestas parciales si llamo a **.onNext()** sin llamar a **.onCompleted**

gRPC - Server

Para inicializar el servicio se crea una instancia de **Server** y se le agregan los servicios implementados

```
private Server server;

private void start() throws IOException {
    server = ServerBuilder.forPort(50051)
        .addService(new GreeterImpl())
        .build().start();
}
```

Yo puedo hacer todos los addService que yo quiera. Entonces, el mismo servidor puede estar escuchando N servicios distintos.

```
Runtime.getRuntime().addShutdownHook(() -> {
    @Override
    public void run() {servidor cuando se esta apagando la JVM
        try {
            HelloWorldServer.this.stop();
        } catch (InterruptedException e) {
        }
    });
}
```



Ejercicio 6 - Implementar Servicio

1. Implementar el servidor.

