UNIVERSIDAD DE ANTIOQUIA
ARQUITECTURA DE SOFTWARE
Laboratorio Nro 3. BROKERS Introducción a gRPC y Protocol Buffers.
Profesor Ph.D Diego José Luis Botia Valderrama

Introducción

Hace mucho tiempo, estamos utilizando los servicios REST para compartir datos entre el cliente y el servidor. Además, utilizamos el protocolo HTTP/1.0 para la transmisión de datos en formato textual con encriptación.

HTTP necesita varios metadatos para especificar el tipo de datos, el tipo de compresión y muchas más cabeceras. Necesitamos enviar datos adicionales como encabezado, estado, tipo de compresión y tipo de datos. Debido a que HTTP/1.1 transmite datos textuales, siempre estamos utilizando algún método de serialización (JSON/XML) para hacer los datos parseables y legible para el ser humano.

Veamos las normas que existen hoy en día. El primero con el que probablemente nos encontremos es REST + HTTP/1.1. Por supuesto, existen diferentes estándares, pero alrededor de tres cuartas partes de la comunicación cliente-servidor está cubierta por ellas. Una mirada más de cerca nos dirá que REST se está convirtiendo en CRUD en el 95% de los casos.

Así que, tenemos lo siguiente:

- Protocolo HTTP/1.1 ineficiente: cabeceras sin comprimir, ausencia de una conexión bidireccional adecuada, uso ineficiente de los recursos del sistema operativo, tráfico adicional, y retrasos no deseados.
- La necesidad de ajustar nuestro modelo de datos y eventos a REST + CRUD, que a menudo es demasiado difícil. Además, muchos desarrolladores no aplican correctamente las operaciones REST o no entienden el funcionamiento del protocolo HTTP 1.1. Se recomienda leer el RFC 2616.

Por todo lo anterior se creó el bróker gRPC, una tecnología que facilita la comunicación asíncrona de datos generalmente a través del protocolo HTTP/2, y usando mecanismos de serialización eficientes.

Entre las principales características de gRPC están:

- Utiliza protobuf o proto3 como mecanismo de serialización de datos estructurados. Esta herramienta ha demostrado ser útil y eficaz. Todos los que necesitan productividad lo han utilizado y han pensado en formas de proporcionar el tráfico necesario. Pero ahora todo se reúne en un solo paquete.
- HTTP/2 como capa de transporte, que es una opción muy poderosa. Permite la compresión completa de datos, el control del tráfico, la llamada de eventos en el servidor, y el uso excesivo de un socket para varias peticiones paralelas.
- Rutas estáticas, lo que significa que ya no hay más "**service/collection/resource/request? parameter=value**". Ahora sólo hay "**servicio**", y depende de usted decidir qué modelo utilizar para describir lo que hay dentro de su servicio.
- Ya no es necesario conectar sus métodos a los métodos HTTP, ni devolver los valores a los estados HTTP.
- SSL/TLS, OAuth 2.0, autenticación a través de los servicios de Google, y la posibilidad de añadir su propia autenticación (una de dos factores, por ejemplo).
- Soporte para 9 lenguajes: C, C++, Java, Go, Node.js, Python, Ruby, Objective-C, PHP, C#.

El soporte de gRPC para la API pública de Google, ya está funcionando para algunos servicios. Por supuesto, las versiones REST permanecerán. Pero ¿elegiría una versión REST de una aplicación móvil o una versión gRPC, dado que son iguales en términos de gastos de desarrollo, pero la versión gRPC funciona el doble de rápido?

Protobuf / proto3

Es un nuevo método de serialización desarrollado por Google que actúa como un método de serialización en un medio de transmisión entre la capa de transporte HTTP. Lo más novedoso en la serialización de protobuf es su uso en el protocolo HTTP/2.

Google está desarrollando esta tecnología para hacer la web más rápida. Protobuf / proto3 funcionará como un protocolo para transmitir datos sólo en búferes.

La documentación oficial de protobuf/ proto3 se encuentra acá:

https://developers.google.com/protocol-buffers/docs/proto3

Por otro lado, Google ha diseñado el lenguaje proto3 que permite hacer fácil la serialización. Este lenguaje funciona como un IDL (Interface Definition Language) .

La documentación oficial la puede encontrar acá:

https://developers.google.com/protocol-buffers/docs/proto3?source=post_page

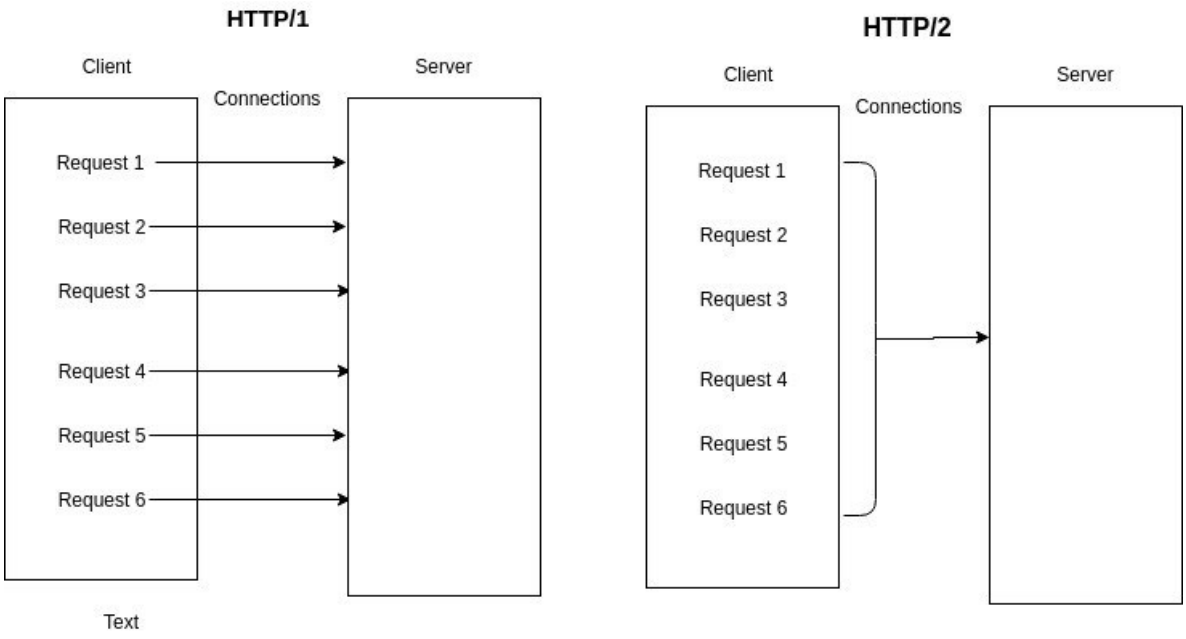
Que es HTTP/2

Es un nuevo y rápido protocolo más robusto que el protocolo HTTP/1.1. Transmite los datos de forma binario, no en formato textual y es la principal razón detrás de la transmisión libre de errores. Ac continuación se presentan sus principales características:

- La transferencia de datos entre el servidor y el cliente es altamente fiable.

- Una sola conexión para varios archivos: Puede transferir múltiples archivos en una unica solicitud.

- Streaming de solicitudes y respuestas: La característica más poderosa de HTTP/2 es la transmisión por secuencias de solicitudes y respuestas, que puede enviar múltiples solicitudes y recibir múltiples respuestas en una sola conexión, como un flujo. La siguiente figura muestra la diferencia entre la transmisión de HTTP/1.1 y HTTP/2.

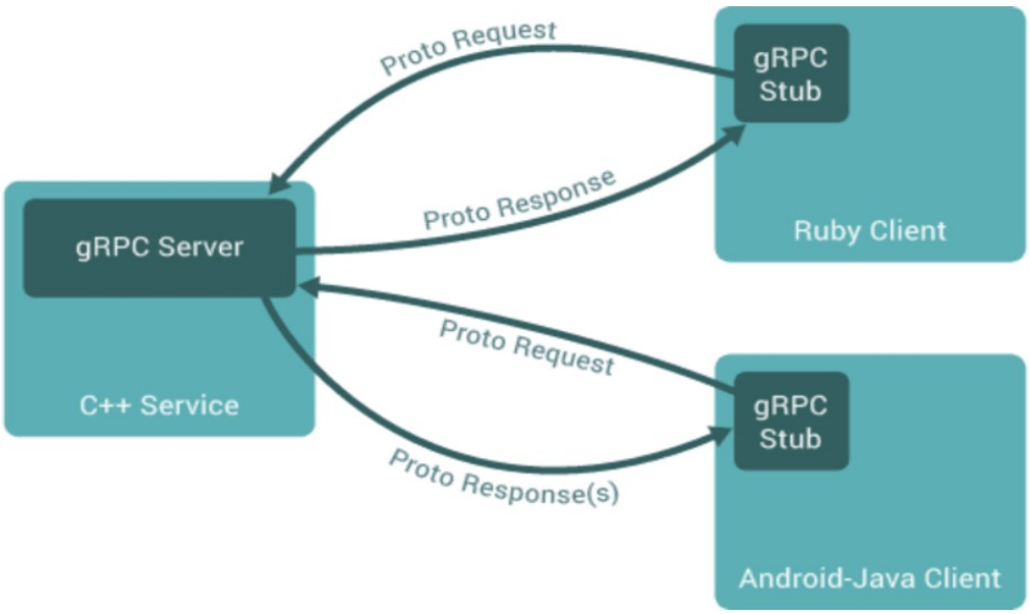


gRPC

El mundo real se está moviendo hacia la Arquitectura Orientada a Servicios (SOA) o a Microservicios para hacer la infraestructura más escalable. Vamos a utilizar la tecnología gRPC para hacer la conexión remota porque es más rápida que cualquier otro servicio como REST, SOAP y socket.

La documentación oficial la puede encontrar acá: https://grpc.io/docs/

A continuación, se presenta la arquitectura básica de gRPC



Para usar esta tecnología, emplearemos un modelo basado en el lenguaje Proto3.

A continuación, se muestra un ejemplo que tiene una simple petición la cual contiene un nombre de usuario asociado a un tipo de dato **string.**

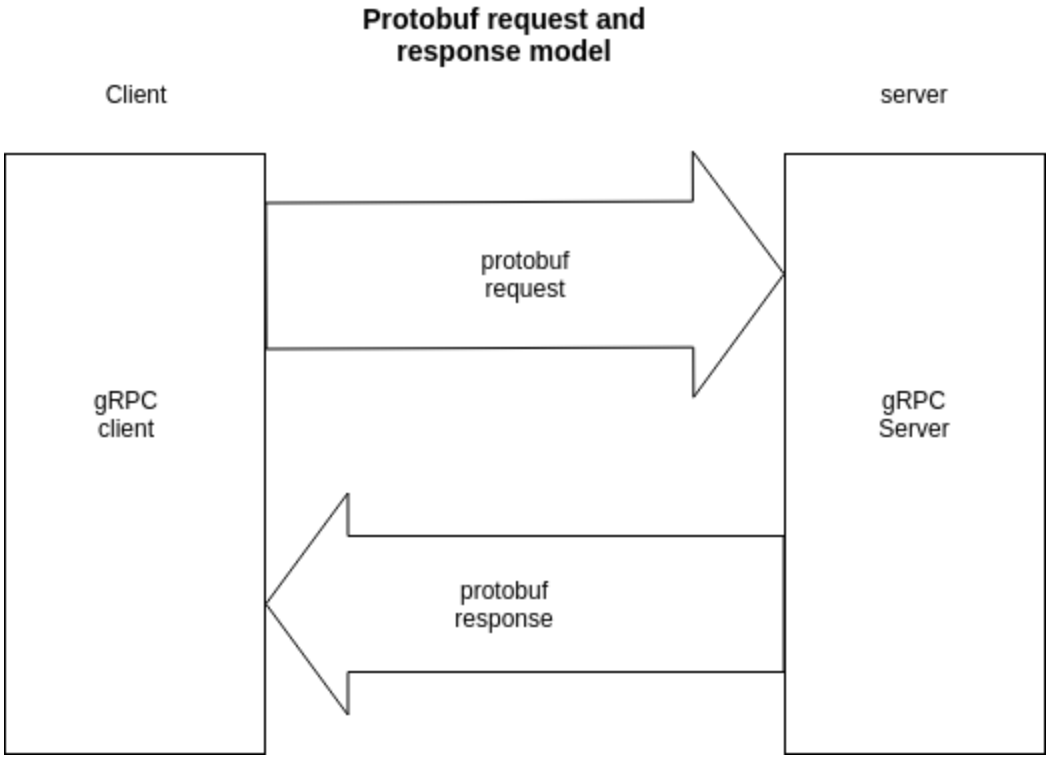
```
syntax = "proto3"; // this is the declaration of the proto language syntax.

// this model is for request

message WelcomeRequest {

  string name = 1; // Here we are declaring the fields need to be transmitted in buffer.

}

// This model is for response.

message WelcomeResponse {

 string message = 1;  // Here we are declaring the fields need to be transmitted in
buffer.

}
```

El anterior código se debe guardar en un formato de extensión **.proto**. Con el ejemplo anterior ya se tiene un modelo de petición proto listo para compilarse.

En un archivo proto podremos definir cuatro tipos de servicios diferentes:

- Servicios tradicionales: Servicios sin streaming. El cliente envía una petición y servidor responde.
- Servicios con streaming desde servidor: El cliente envía una petición al servidor y se abre un canal de streaming que permite al servidor mandar datos por el canal creado conforme se vayan generando.
- Streaming desde el cliente: El cliente puede abrir un canal de streaming hacia el servidor.
- Streaming bidireccional: Es el ejemplo que nos ocupa. El canal se establece en ambos sentidos permitiendo al cliente y al servidor mandar mensajes en el momento que consideren oportuno

Para declarar uno u otro tipo de mensajes basta con añadir la palabra reservada *stream* delante del tipo de mensaje de cliente a servidor y/o del tipo de mensaje de servidor a cliente.



Herramientas Necesarias

- NodeJS 10.X o superior
- NPM
- Editor Visual Studio Code o similares.

Implementación del Laboratorio

En esta aplicación, vamos a enviar sólo el nombre del usuario en la solicitud y recibiremos la respuesta como 'Hola <Usuario> Bienvenido a la UdeA'.

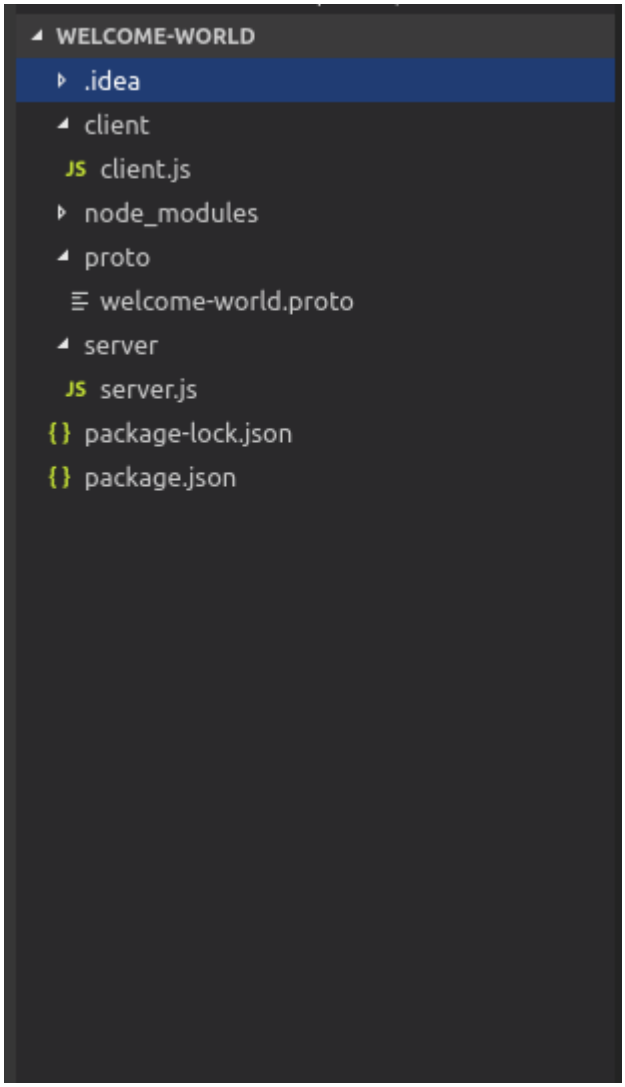
- Cree un directorio llamado **'welcome-world'**, y vaya al directorio en la terminal (Puede usar CMD de Windows o una nueva terminal en LINUX).

- Coloque el comando '**npm init**' y siga las instrucciones para inicializar el proyecto.

- Cree un directorio llamado **proto** donde se almacenarán los archivos proto.

- Cree un directorio llamado **server** donde se almacenarán los archivos del servidor y cree un directorio de **client** donde se almacenarán los archivos de cliente.

Asegúrese que su directorio de trabajo tenga la siguiente estructura:



Ahora crearemos un servicio en el archivo **.proto** para manejar las peticiones y respuestas. Inserte el siguiente código en el archivo

```
syntax = "proto3";

package welcome;

message WelcomeRequest {
string name = 1;
}

message WelcomeResponse {
string message = 1;
}

service WelcomeService {
// this service method needs to be implemented.
```

```
rpc greetUser(WelcomeRequest) returns (WelcomeResponse) {};
}
```

Se ha creado un servicio llamado **WelcomeService** que contendrá nuestros métodos RPC.
Como puede observar se ha creado un método RPC simple llamado **greetUser** que acepta la petición como un tipo de mensaje "**WelcomeRequest**" y devuelve el tipo de mensaje "**welcome response**", que es la declaración del método RPC que necesitamos para implementarlo en nuestro servidor. Vamos a implementar el servicio en nuestra aplicación nodeJS. Escriba el código abajo en su archivo **server.js.**

```
let grpc = require("grpc");
let protoLoader = require("@grpc/proto-loader");

const server = new grpc.Server();
const URL = "0.0.0.0:2019";

let proto = grpc.loadPackageDefinition(
protoLoader.loadSync("../proto/welcome-world.proto",{
keepCase: true,
longs: String,
enums: String,
defaults: true,
oneofs: true
})
);

function greetUser(call, callBack) {
callBack(null, {message: `Hello ${call.request.name} welcome to the UdeA`});
}

server.addService(proto.welcome.WelcomeService.service, { greetUser: greetUser });

server.bind(URL, grpc.ServerCredentials.createInsecure());
server.start();
```

Aquí hemos creado un servidor gRPC en el que hemos implementado el método RPC **greetUser** que devuelve la respuesta dentro de la llamada de retorno o callback.

El siguiente paso es crear un cliente que enlace el servicio a través del archivo proto.

El código del Cliente es (**client.js**):

```
let grpc = require("grpc");
let protoLoader = require("@grpc/proto-loader");
let readline = require("readline");
```

```
let reader = readline.createInterface({
input: process.stdin,
output: process.stdout
});

var proto = grpc.loadPackageDefinition(
protoLoader.loadSync("../proto/welcome-world.proto", {
keepCase: true,
longs: String,
enums: String,
defaults: true,
oneofs: true
})
);

const REMOTE_URL = "0.0.0.0:2019";
let client = new proto.welcome.WelcomeService(REMOTE_URL,
grpc.credentials.createInsecure());
reader.question("Please enter your name: ", answer => {
client.greetUser({name: answer}, (err, res) =>{ console.log(res.message); } );
});
```

Ahora ejecute en una terminal el archivo server.js así:

```
$ node server.js
```

En una nueva terminar ejecute el archivo client.js

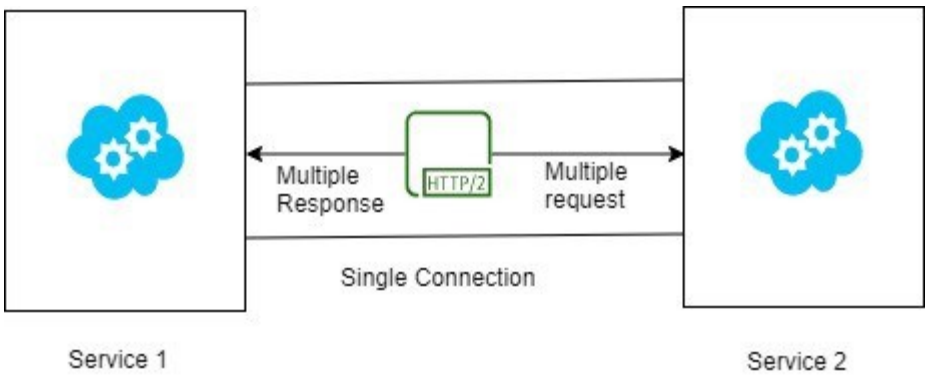
```
$ node client.js
```

Hasta el momento hemos creado una simple aplicación cliente-servidor usando gRPC y protobuf.

Ahora realizaremos una aplicación de tipo Request-Response Streaming

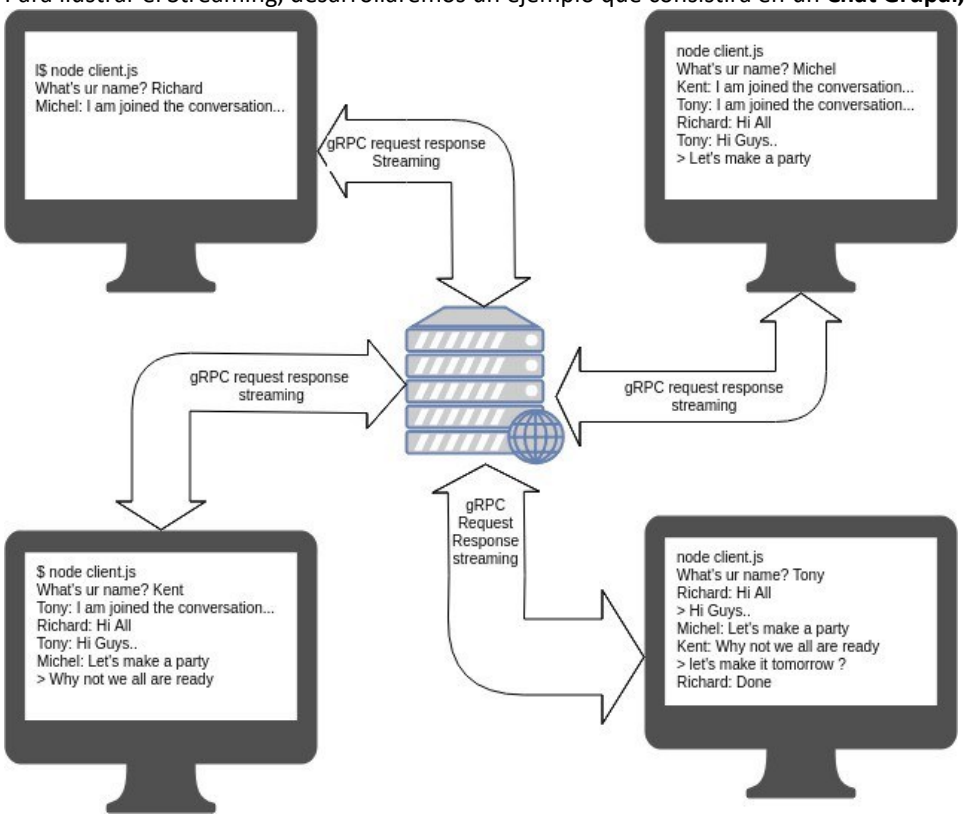
Request response streaming

Request response streaming, significa que puede enviar múltiples peticiones y recibir múltiples respuestas en una única conexión TCP. Así mejorará el rendimiento de la aplicación a la vez que se intercambian datos a través de la web. La siguiente figura presenta un Request-response streaming con contenido serializado con protobuf



Aquí dos servicios que comparten sus recursos a través de gRPC en contenidos proto serializados, permiten que gRPC abra una conexión de streaming entre los servicios para que múltiples peticiones y respuestas puedan ser compartidas dentro de una conexión.

gRPC abre un flujo de escritura para escribir múltiples respuestas.
Para ilustrar el Streaming, desarrollaremos un ejemplo que consistirá en un **Chat Grupal,** representado por la siguiente figura.

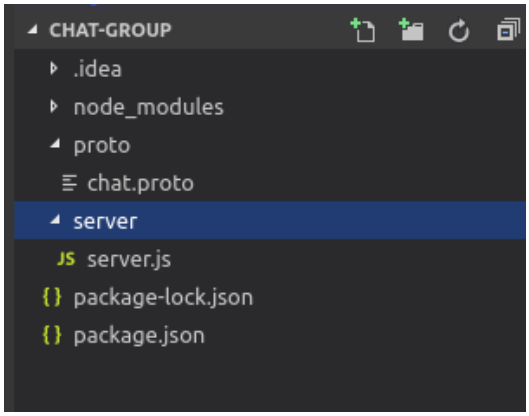


La idea principal de la aplicación es que el servicio creará un grupo de chat y los cuatro clientes pueden chatear entre sí en ese grupo de chat a través de un flujo de gRPC.

Paso-1

Cree un directorio llamado '**chat-group**', que contendrá nuestro servidor de chat principal y dentro del cual se creará un directorio llamado **proto** y otro llamado **server**.

A continuación, se presenta la estructura del proyecto:



Paso 2

Inicialice dentro de la carpeta un nuevo proyecto node así:

```
$ npm init
```

Instale el cargador de paquetes grpc y proto

```
$ npm install --save @grpc/proto-loader

$ npm install google-protobuf

$ npm install grpc
```

Cree el archivo **chat.proto** dentro del directorio proto.

Coloque el siguiente código en el archivo.

```
syntax = "proto3";
package chatGroup;
service Chat {
              rpc join(stream Message) returns (stream Message){}
}
message Message {
                  string user = 1;
                  string text = 2;
}
```

Lo que se ha creado es un método RPC en el cual múltiples flujos de mensajes se enviaran conteniendo el usuario y el texto.

Paso 3

Cree un archivo llamado `server.js` dentro del directorio **server** y coloque el siguiente código.

```
let grpc = require("grpc");
let protoLoader = require("@grpc/proto-loader");
```

```
const server = new grpc.Server();
const SERVER_ADDRESS = "0.0.0.0:2019";

// Load protobuf
let proto = grpc.loadPackageDefinition(
    protoLoader.loadSync("c://Users/Lenovo/Documents/chat-group/proto/chat.proto", {
        keepCase: true,
        longs: String,
        enums: String,
        defaults: true,
        oneofs: true
    })
);

// create an empty array to store the users.
let users = [];

// Method implementation of RPC join
let join = (call) => {
    users.push(call);
    // Get the data from request.
    call.on('data', (message) =>{
        // send join notification
        sendNotification({ user: message.user, text: message.text });
    })
}

// Send message to all connected clients
 let sendNotification =(message) => {
    // for each user write messages.
    users.forEach(user => {
        user.write(message);
    });
}
// Add the implemented methods to the service.
server.addService(proto.chatGroup.Chat.service, { join: join });

server.bind(SERVER_ADDRESS, grpc.ServerCredentials.createInsecure());

server.start();
```

Paso 4

Vamos a crear el servicio del cliente para la aplicación. Cree otro directorio de proyecto llamado chat-client.

Inicialice la aplicación cliente

$ npm init

Instale los paquetes de grpc y proto

- npm install - save grpc @grpc/proto-loader
- npm i protoc

Paso 5

Dentro del proyecto cree un directorio llamado proto, copie el archivo **chat.proto** de **chat-group/proto/chat.proto** y péguelo dentro del directorio **chat-cliente/proto**. Aquí estamos copiando las declaraciones RPC en nuestro cliente.

Paso-6

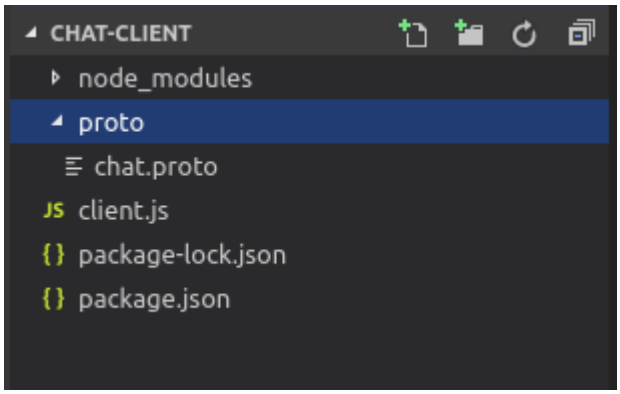
Cree un archivo **client.js** en el directorio raíz.

Pegue el siguiente código en el archivo client.js.

```
let grpc = require("grpc");
let protoLoader = require("@grpc/proto-loader");
let readline = require("readline");

//Read terminal Lines
let reader = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

//Load the protobuf
let proto = grpc.loadPackageDefinition(
    protoLoader.loadSync("c://Users/Lenovo/Documents/chat-group/proto/chat.proto", {
        keepCase: true,
        longs: String,
        enums: String,
        defaults: true,
        oneofs: true
    })
);

const REMOTE_SERVER = "0.0.0.0:2019";

let username;

//Create gRPC client
let client = new proto.chatGroup.Chat(
    REMOTE_SERVER,
    grpc.credentials.createInsecure()
);

// Ask the user to enter name.
reader.question("Please enter your name: ", answer => {
    username = answer;
```

```
    startChat();
});

//Start the stream between server and client
let startChat =() => {
    // Join the chat service
    let channel = client.join();
    // Write the request
    channel.write({user: username, text: "I am joined the conversation..."});
    // get the data from response
    channel.on("data",(message) => {
    if (message.user == username) {
        return;
    }
    console.log(`${message.user}: ${message.text}`);
    });
    // Read the line from terminal
    reader.on("line", (text) => {
        channel.write({user: username, text: text});
    });
}
```

La estructura de su proyecto debería verse así:



Generación del código para cada lenguaje

Ya tenemos la definición de nuestro servicio, conjuntamente con la definición de nuestros mensajes. Ahora necesitamos generar el código en el lenguaje en el que nos sintamos más cómodos para empezar a trabajar en nuestra aplicación. Para ello tendremos que descargar la aplicación protoc que corresponda a nuestro sistema operativo.
Tenemos que especificar el archivo o los archivos de entrada y el lenguaje en los que queremos obtener la salida.

Por ejemplo, nosotros tenemos el fichero chat.proto dentro del directorio proto y queremos generar la implementación en python, java y c#. Para ello, utilizaremos los siguientes comandos:

protoc -I=proto --python_out=proto/python proto/chat.proto
protoc -I=proto --csharp_out=proto/csharp proto/chat.proto
protoc -I=proto --java_out=proto/java proto/chat.proto

En nuestro caso podremos hacer la implementación con JavaScript usando el siguiente comando

```
PS C:\Users\Lenovo\Documents\chat-client\node_modules\protoc\protoc\bin> .\protoc.exe --
proto_path=C:\Users\Lenovo\Documents\chat-client\proto --
js_out=C:\Users\Lenovo\Documents\chat-client\proto C:\Users\Lenovo\Documents\chat-
client\proto\chat.proto
```

Ya tenemos creado el servicio de grupo de chat y cliente. Use los siguientes comandos para ejecutar el proyecto

Paso -1 (para ejecutar el server)

Vaya al directorio chat-group/server

$ cd chat-group/server/

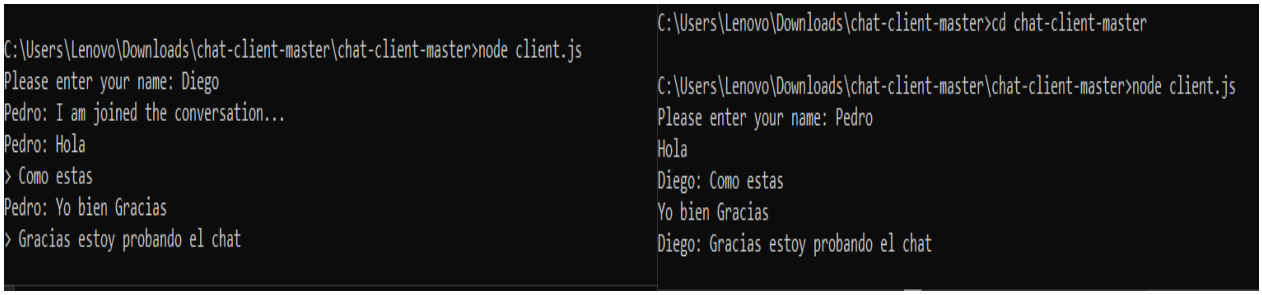
$ node server.js

Paso -2 (Abra varias terminales y ejecute el siguiente comando para ejecutar el cliente)

Vaya al directorio **chat-client**

$ cd chat-client

$node client.js



Ejercicio (Entrega Opcional) ENTREGA 16 Noviembre de 2020

Realice una aplicación cliente con gRPC y Protocol Buffers que según la cantidad de días de vacaciones acumulados y según la petición de días de permiso que solicite un empleado se le autorice o no el permiso.

Para el desarrollo de la aplicación utilice los siguientes archivos

vacaciones.proto

```
syntax = "proto3"; //Specify proto3 version.

package work_leave; //Optional: unique package name.

//Service. define the methods that the grpc server can expose to the client.
service EmployeeLeaveDaysService {
  rpc EligibleForLeave (Employee) returns (LeaveEligibility);
  rpc grantLeave (Employee) returns (LeaveFeedback);
}
```

```
// Message Type fefinition for an Employee.
message Employee {
  int32 employee_id = 1;
  string name = 2;
  float accrued_leave_days = 3;
  float requested_leave_days = 4;
}

// Message Type definition for LeaveEligibility response.
message LeaveEligibility {
  bool eligible = 1;
}

// Message Type definition for LeaveFeedback response.
message LeaveFeedback  {
  bool granted = 1;
  float accrued_leave_days = 2;
  float granted_leave_days = 3;
}
```

server/index.js

```
const grpc = require('grpc');

const proto = grpc.load('proto/vacaciones.proto');
const server = new grpc.Server();

//define the callable methods that correspond to the methods defined in the protofile
server.addProtoService(proto.work_leave.EmployeeLeaveDaysService.service, {
  /**
  Check if an employee is eligible for leave.
  True If the requested leave days are greater than 0 and within the number
  of accrued days.
  */
  eligibleForLeave(call, callback) {
    if (call.request.requested_leave_days > 0) {
      if (call.request.accrued_leave_days > call.request.requested_leave_days) {
        callback(null, { eligible: true });
      } else {
        callback(null, { eligible: false });
      }-1
    } else {
      callback(new Error('Invalid requested days'));
    }
  },

  /**
  Grant an employee leave days
  */
  grantLeave(call, callback) {
    let granted_leave_days = call.request.requested_leave_days;
    let accrued_leave_days = call.request.accrued_leave_days - granted_leave_days;

    callback(null, {
      granted: true,
      granted_leave_days,
      accrued_leave_days
```

```
    });
  }
});

//Specify the IP and and port to start the grpc Server, no SSL in test environment
server.bind('0.0.0.0:50050', grpc.ServerCredentials.createInsecure());

//Start the server
server.start();
console.log('grpc server running on port:', '0.0.0.0:50050');
```

SALIDA DEL SISTEMA

Si por ejemplo se tiene la información del siguiente empleado

```
    employee_id: 1,
    name: 'Diego Botia',
    accrued_leave_days: 10,
    requested_leave_days: 4
```

La salida debe ser similar a esta

```
$ node client/node
{ granted: true, accrued_leave_days: 6, granted_leave_days: 4 }
```