Programación de Sistemas Distribuidos Práctica 5: Comunicación Sockets

Carlos Bermúdez Expósito
Arturo Requejo Portilla
Augusto Messina

UNIVERSIDAD NEBRIJA

Tabla de contenido

I. Enunciados

PRÁCTICA 5: Para la práctica 5 queremos ofreceros varias opciones para que la hagáis en GRUPOS de 3, pero por supuesto si alguien quiere hacer alguna propuesta de algo relacionado con sistemas distribuidos, adelante. Las propuestas son las siguientes:

a) ¿Te atreves a hacer la práctica 2 de CORBA en otro lenguaje como Python o Ruby? Muestra evidencias del trabajo con capturas. Punto extra la grabación de un vídeo (no hace falta que la calidad sea buena), simplemente el resultado.

b) Para recordar el tema de Sockets, ¿te animas a hacer un chat con sockets? Hay bastante documentación para hacer un chat con Sockets en node. ¿Te gustaría hacerlo con otros lenguajes de programación?

c) Programar el algoritmo de Berkley ¿Cómo lo implementarías en caso de que caiga el nodo
principal? Implementar logs en los nodos. Utilizando el lenguaje de programación con el que os sintáis más cómodos.

d) Programar el algoritmo de Christian. ¿Cómo lo implementarías en caso de que caiga el nodo principal? Implementar logs en los nodos. Utilizando el lenguaje de programación con el que os sintáis más cómodos.

e) Opción abierta

II. Introducción

Para esta práctica, nosotros hemos decidido realizar un chat con Websockets utilizando el framework Node.js. Los websockets nos permiten realizar una comunicación bidireccional y en tiempo real entre cliente y servidor. No sigue el modelo HTTP tradicional de solicitud-respuesta, sino que se realiza una conexión inicial llamada “handshake” para posteriormente dejar abierta la comunicación.

Los Websockets establecen una conexión persistente entre cliente y servidor permitiendo así que los datos se envíen y reciban de manera simultánea en ambos extremos sin realizar solicitudes adicionales.

Permiten a varias máquinas comunicarse en una red fiable y eficiente basada en TCP. Es necesario realizar el handshake para poder establecer comunicación, una vez establecido el canal de comunicación se quedará abierto. Gracias a lo mencionado anteriormente podemos acceder a los datos de forma más rápida.

Para realizar la práctica, utilizamos Node.js, que es un entorno de ejecución diseñado para crear aplicaciones web, que permite ejecutar JavaScript y TypeScript en el servidor, en lugar de en el navegador del cliente. Sus principales ventajas son:

- Gran rendimiento debido a las conexiones no bloqueantes que genera, que permiten que la aplicación no pierda tiempo realizando tareas, delegando estas en hilos secundarios que no afectan al hilo principal.
- Su arquitectura basada en eventos le permite gestionar de forma eficiente una gran cantidad de peticiones simultáneas, haciéndolo ideal para aplicaciones en tiempo real.
- Node.js cuenta con una amplia gama de módulos tanto nativos como creados por la comunidad. Estos módulos, que se instalan con Node Package Manager, permiten a los desarrolladores aprovechar funcionalidades ya existentes y optimizadas, lo que ahorra tiempo de desarrollo.

III. Resultados

Link al repositorio https://github.com/augusMessina/WebSocket-Chat

Para levantar un servidor en Node.js, utilizamos nuestros conocimientos en Programación de Sistemas en Internet para crear un servidor de GraphQL.

GraphQL es un lenguaje de consulta, en el que el cliente envía al servidor una sentencia en la que indica los datos que requiere de este, y el servidor le devuelve exclusivamente estos datos consultados. Además, los servidores muestran un esquema con todas las llamadas y tipos de datos que se pueden pedir. Los servidores GraphQL dividen las peticiones en dos tipos principales:

- Queries: estas peticiones tienen como objetivo que el cliente obtenga datos por parte del servidor, sin realizar ninguna modificación sobre estos. Por ejemplo, en la siguiente query, el cliente estaría pidiendo el nombre de los usuarios albergados en el servidor:

```
query{
  users{
    name
  }
}
```

Ejemplo general de una Query.

Obviamente esto no es más que un ejemplo, las llamadas y datos que se pidan dependerá del esquema definido en cada servidor.

- Mutations: en este otro tipo de peticiones, los clientes realizan modificaciones sobre los datos, ya sea añadiendo un dato nuevo, borrándolo o editándolo. El siguiente ejemplo simula una petición en la que se añadiría un nuevo usuario en un servidor:

```
mutation{
  addUser(name: "user"){
    name
  }
}
```

Ejemplo general de una Mutation.

Las mutations, además de modificar datos, también pueden devolverlos. En el ejemplo de arriba se está devolviendo el campo “name” de la respuesta del servidor.

Además de estos dos tipos de peticiones, existe un tercero, que es de hecho la principal razón por la que hemos elegido levantar un servidor de este tipo, que son las suscripciones (subscriptions). Estas le permiten a un cliente recibir actualizaciones en tiempo real cuando un evento específico sucede en el servidor. Al realizar una petición de tipo subscription, el cliente se conecta al servidor utilizando una conexión de Websockets. Por ejemplo, si un cliente enviase la siguiente solicitud, se crearía una conexión por Websockets entre este y el servidor, mediante la cual el servidor le enviará al cliente un mensaje con el título y el cuerpo de un post cada vez que un nuevo post se publicase:

```
subscription{
  postAdded{
    title
    body
  }
}
```

Ejemplo general de una Subscription.

Sabiendo ya cómo funciona cada tipo de petición de GraphQL, decidimos levantar un servidor que almacena un array de mensajes, siendo cada mensaje un objeto formado por “user”, que corresponde con el identificador del usuario que mandó el mensaje, y “message”, que es el cuerpo del mensaje como tal. Para interactuar con este array, el servidor ofrece las siguientes funciones.

- getMessages: esta función es una query que devuelve al cliente un array con todos los mensajes almacenados en el servidor. El cliente puede elegir si quiere que el servidor le devuelva únicamente el campo “user” de cada mensaje, o el campo “message”, o ambos, como se ve en el ejemplo de abajo.

```
query{
  getMessages{
    user
    message
  }
}
```

Ejemplo de llamada a la query getMessages.

- addMessage: esta segunda función es una mutation que añade nuevos mensajes al servidor, tomando como parámetros el usuario que envía el mensaje y el texto enviado.

```
muation{
  addMessage(user: "user", message: "example text"){
    info
    message{
      user
      message
    }
  }
}
```

Ejemplo de llamada a la mutation addMessage.

En el ejemplo de arriba se está añadiendo un mensaje con “user” como usuario remitente del mensaje, y “example text” como cuerpo del mensaje. Además, esta mutation devuelve “info”, que es un mensaje de confirmación, y el propio mensaje que ha sido añadido.

- newMessage: por último, esta función es una suscripción, mediante la que el cliente recibirá los datos de los nuevos mensajes a medida que estos son añadidos.

```
subscription{
  newMessage{
    user
    message
  }
}
```

Ejemplo de llamada a la subscription newMessage.

Con esto quedan definidos los métodos de nuestro servidor. De una forma más gráfica, puede verse de la siguiente forma:

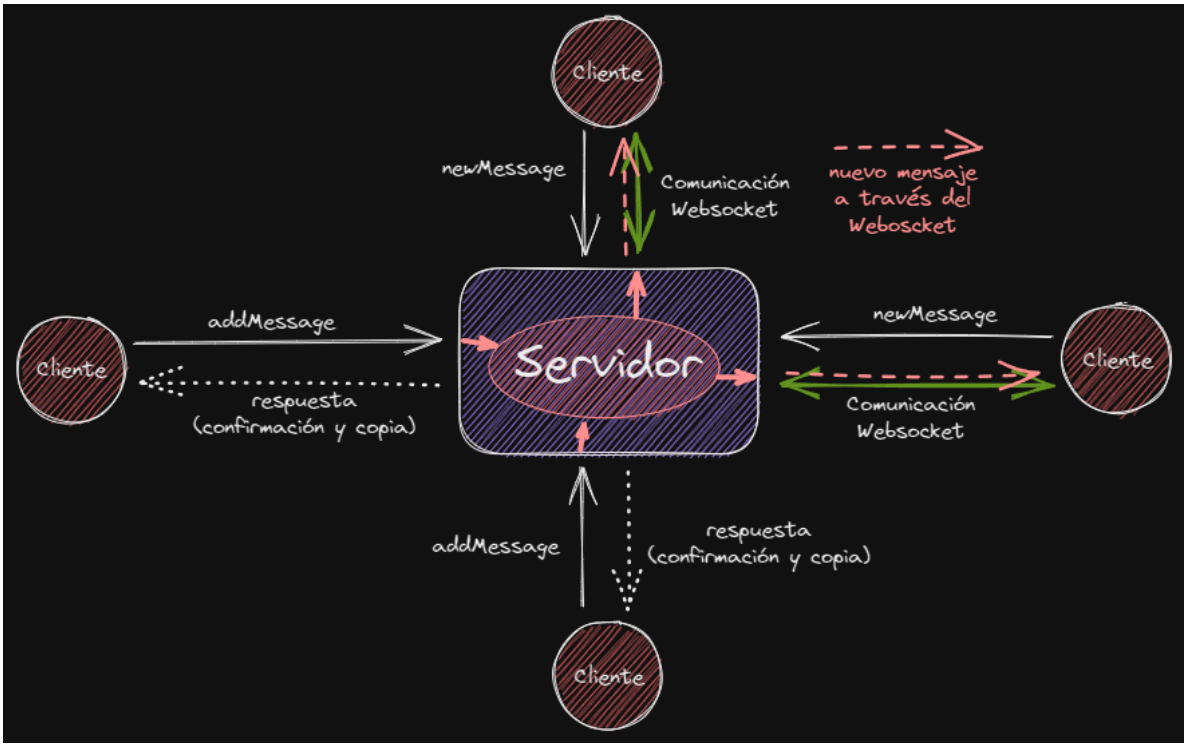
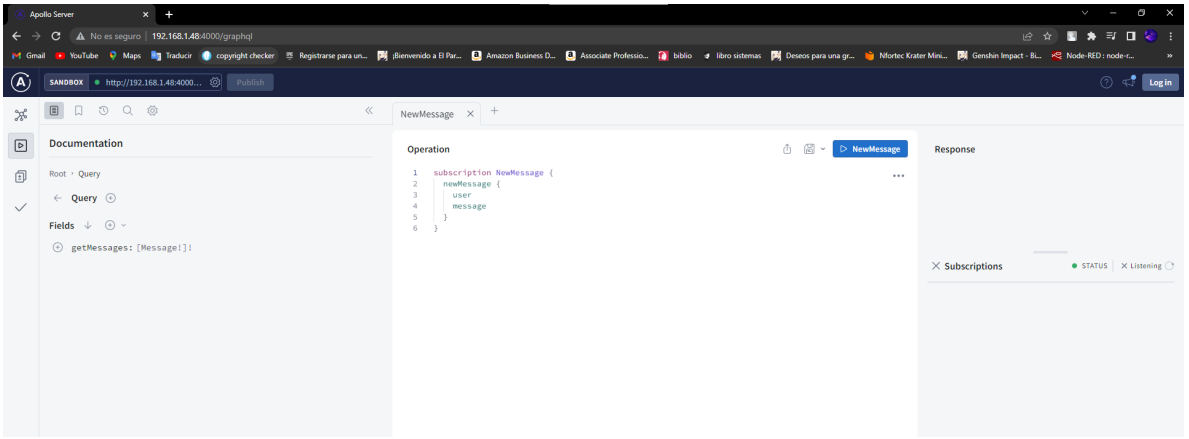


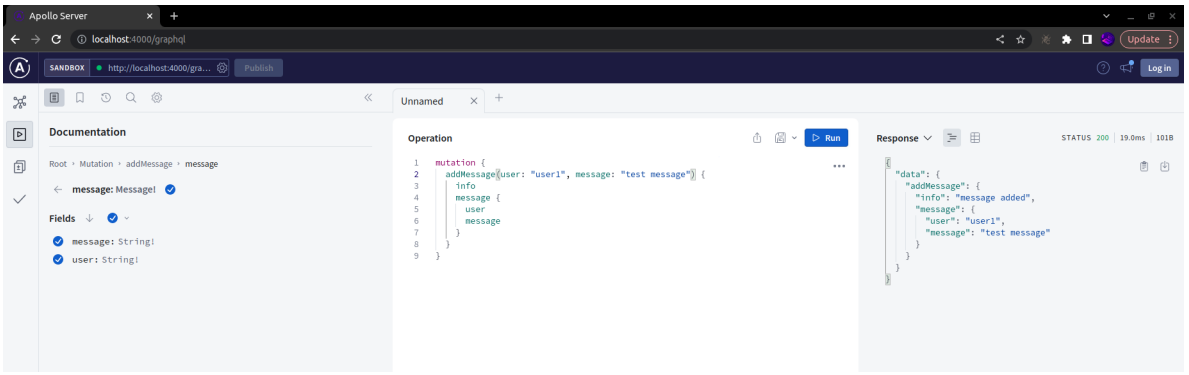
Diagrama del funcionamiento del servidor GraphQL y las conexiones con los clientes.

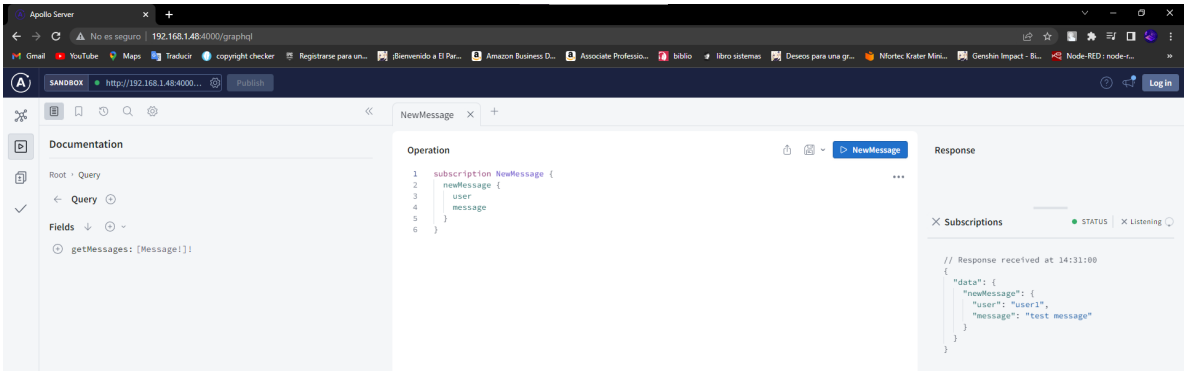
Ahora bien, para poder levantar este servidor hicimos uso de la herramienta Apollo Server, que permite crear un servidor GraphQL con gran facilidad y con una interfaz de usuario intuitiva. Para ello, hicimos uso del módulo “@apollo/server”, al igual que usamos otros paquetes requeridos como “ws” y “graphql-ws” para poder utilizar Websockets, siendo todo instalado con npm (Node Package Manager). El código puede encontrarse en nuestro repositorio: WebSocket Chat. Si montamos el servidor correctamente y siguiendo la guía oficial de Apollo, podremos acceder a nuestro servidor a través de la IP de este y el puerto designado. Una vez dentro, encontraremos una interfaz que nos permitirá realizar todas las operaciones definidas.



Llamada a la subscription newMessages utilizando la interfaz de Apollo.

Como podemos ver en la imagen superior, desde un cliente estamos realizando una operación de suscripción, quedando así la conexión Websocket abierta, a la espera de nuevos mensajes.

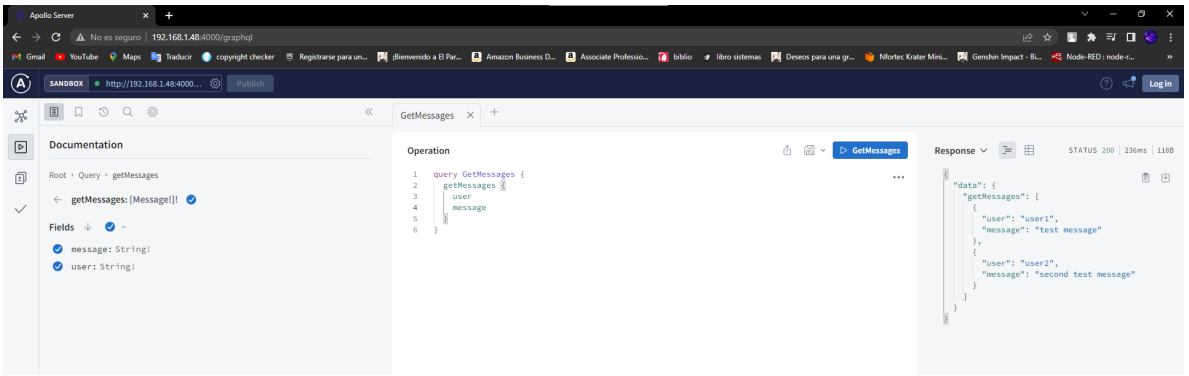




Llamada a la mutation addMessage y resultado devuelto por la suscripción en el cliente suscrito.

Podemos ver en las imágenes superiores que desde un ordenador se envía un nuevo mensaje, y al ordenador suscrito le llega, a través de la suscripción, este nuevo mensaje.

Por último, si llamamos a la query “getMessages”, obtendremos un array con todos los mensajes existentes:



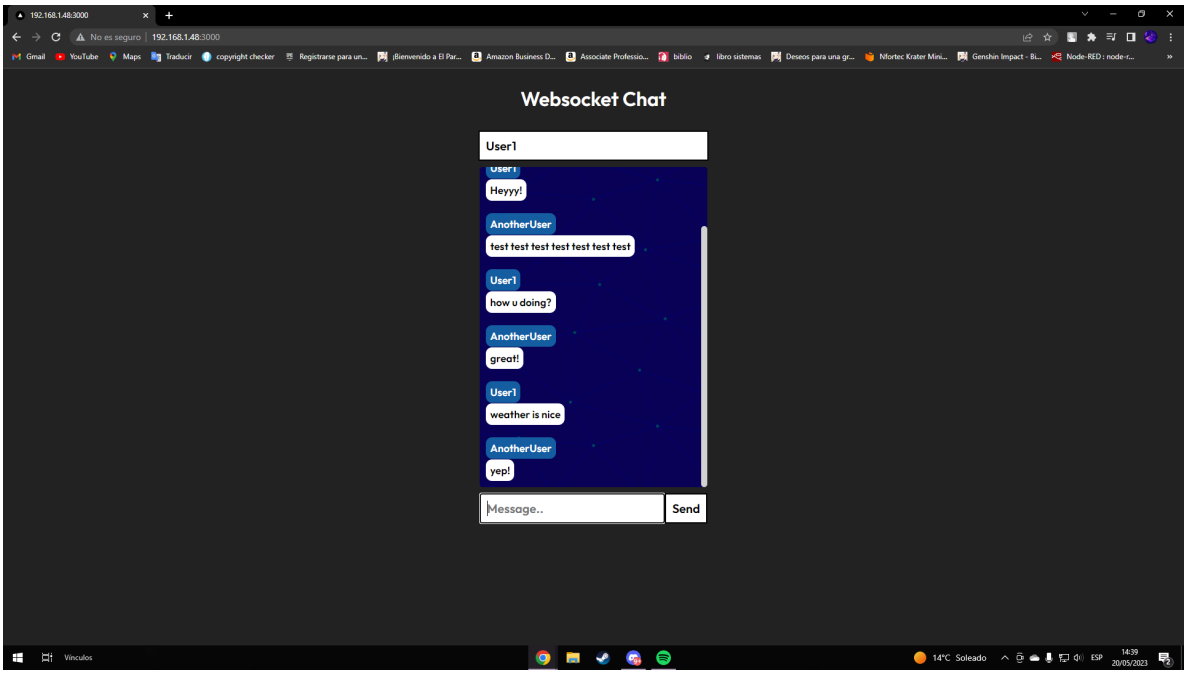
Llamada a la query getMessages utilizando la interfaz de Apollo.

Sin embargo, aunque esta interfaz de usuario que nos proporciona Apollo es intuitiva y fácil de usar, sí que es cierto que hay que saber utilizar y escribir las queries, mutations y subscriptions, y en general entender cómo funciona GraphQL. Para solucionar este problema y permitir que dos usuarios o más puedan mantener una comunicación de forma más sencilla, se nos ocurrió levantar un servidor web adicional, el cual proporcionaría una interfaz de usuario más amigable y facilitaría las conexiones necesarias con el servidor GraphQL.

Para crear este servidor secundario, utilizamos Next.js, un framework de desarrollo web basado en React, el cual nos permite que varios usuarios, al conectarse con el servidor, reciban una interfaz web interactiva capaz de enviar solicitudes a otras direcciones. En

concreto, las peticiones que este servidor manda a hacer a un cliente de forma automática son:

- Una primera petición de getMessages al servidor GraphQL. Tras recibir la respuesta, muestra el array recibido en un display de mensajes donde se puede ver el usuario que envió el mensaje junto al texto del mensaje.
- Tras ello, solicita la subscription newMessage, creando así la conexión websocket entre el cliente y el servidor GraphQL. Cada vez que el servidor le mande datos por este canal, serán añadidos al display de mensajes.
- Por último, en la interfaz se muestran dos cajas que el usuario puede rellenar con su nombre de usuario y el texto a enviar. Cada vez que el usuario presione en el botón “Send”, se enviará una petición addMessage, guardando así un nuevo mensaje en el servidor, y activando el flag que el servidor leerá para reenviar a los clientes suscritos el nuevo mensaje.



Interfaz creada por el servidor de Next.js, mediante la que el cliente se comunicará con el servidor GraphQL de forma más eficiente.

IV. Conclusión

En esta quinta práctica hemos realizado una investigación del funcionamiento de los websockets aplicados a un chat en tiempo real. Hemos realizado una comunicación entre varios nodos mediante un modelo de cliente-servidor, utilizando el framework Node.js y añadiendo de forma complementaria una interfaz gráfica usando Next.js.

V. Bibliografía

- ¿Qué es WebSocket?
- Introducción a Express/Node - Aprende desarrollo web | MDN
- Acerca | Node.js
- Subscriptions in Apollo Server - Apollo GraphQL Docs
- Subscriptions - Apollo GraphQL Docs