Escuela de ingeniería

Departamento de informática y sistemas

Proyecto N° 1

Middleware

INTEGRANTES:

Juan Felipe Londoño Gaviria

Juan David Pérez Sotelo

Felipe Ríos López

Profesor: Edwin Nelson Montoya Munera

Universidad EAFIT

Escuela de ingeniería

ST0263: Tópicos especiales de telemática

Medellín

23 De marzo de 2020

Contenido

Figuras

Escuela de ingeniería

Departamento de informática y sistemas

- **Introducción**

En este proyecto crearemos un middleware que permita a un conjunto de clientes enviar y recibir mensajes de datos donde será probado por una aplicación, la app1. Trata de dos módulos: un módulo que provee los anuncios y el otro es un módulo de cliente que recibe mensajes enviados a un canal por el módulo de anuncios donde evidenciaremos el manejo del middleware que creamos.

- **Objetivo**

El objetivo de este proyecto 1 es diseñar e implementar un MIDDLEWARE que permita a un conjunto de CLIENTES enviar y recibir mensajes de datos. Esto permitirá a los alumnos evidenciar, conocer y aplicar, muchas de las características subyacentes a los sistemas distribuidos (ej: heterogeneidad, transparencia, seguridad, escalabilidad, entre otros) que deben implementar las aplicaciones o los subsistemas base (sistema operativo, middlewares, frameworks, apis, etc). En este caso, dicha complejidad y características del sistema distribuido serán diseñadas e implementadas en un MIDDLEWARE, de tal manera que para las aplicaciones usuarias (CLIENTES) sea transparente y seguro su uso.

- **Requerimientos**

Para la ejecución del programa se deben instalar las librerías de HashMap y Queue con los comandos:
Npm install queue
Npm install hashmap
La ejecución del programa se hace por medio de
“npm run dev” para el servidor
Y node src/client.js para todos los clientes que quieran hacer uso del servicio.
Los comandos válidos son:
El primer mensaje debe ser “register” o “login”
El segundo debe ser un usuario y contraseña separados;
“MiUsuario MiContraseña”

Y por último, estos llegan al servidor:

create <canal> -> Crea un canal.
delete <canal> -> Borra un canal.
send <canal> <mensaje> -> Envía el mensaje a ese canal.

pull <canal> -> Trae todos los mensajes del usuario en ese canal.

Funcionales:

- La conexión / desconexión, debe ser con usuarios autenticados
- El envío y recepción de mensajes debe identificar los usuarios.
- La aplicación debe permitir la creación y eliminación de canales
- La aplicación debe permitir el registro e ingreso de usuarios
- La aplicación debe permitir a los usuarios el envio de mensaje a los canales creados
- La aplicación debe permitir a los usuarios obtener mensajes de los canales

No funcionales:

- El transporte de los mensajes debería ser encriptada, así como el servicio de autenticación.
- La aplicación deberá manejar buen nivel de seguridad
- La aplicación deberá ser intuitiva y clara para el usuario
- La aplicación deberá manejar una alta disponibilidad
- La aplicación deberá manejar tiempos de respuesta cortos
- La aplicación deberá ser escalable
- La aplicación deberá tener persistencia de datos

- **Atributos de calidad de la arquitectura:**

- **Mantenibilidad:** Para este apartado lo que hicimos fue evitar repetir código por lo cual creábamos una función encargada de hacer lo que teníamos en el código repetido para así llamarla y detectar los fallos más fácilmente y seguir reutilizándola, también tratamos de poner a las funciones y variables nombres descriptivos para que fuera más fácil entender su función asignada

- **Extensibilidad:** Es fácil añadir nuevos casos de usos para las colas, porque la definición de los mensajes posibles está en un switch que simplemente se le pueden ir añadiendo parámetros y la lógica de qué es lo que manda y qué se debe hacer con lo que llegue al servidor.

- **Simplicidad:** Intentamos mantener nuestro código de una manera simple donde cada función cumpliera solo un único trabajo. El nombre de las variables es relacionado a cada una de sus funciones, siguiendo principios de código limpio.

- **Rendimiento:** El rendimiento de la aplicación a la hora de buscar usuario y registrarlo se hace línea por línea, lo que lo hace tener un rendimiento de o(n) y a la hora de registrar vuelve a crear el documento de texto. Sin embargo, en el manejo de mensajes y canales, es más eficiente puesto que se implementó un hash map, que contiene un hashMap que contiene colas dentro de él y en esta estructura el acceso de los datos es O(1).

- **Escalabilidad:** Se utilizó una arquitectura de API rest de manera que permita mantener muchos usuarios conectados desde diferentes clientes, en contraparte con una arquitectura de sockets que sería muy ineficiente. El manejo de los mensajes es temporal y se hace en memoria, ya que es mucho mejor que hacerlo con una base de datos puesto que el almacenamiento en disco es muy costoso y además es poco escalable.

- **Seguridad:** Respecto al login utilizamos un documento de texto para guardar nuestros usuarios y para la comunicación al otro modulo utilizamos HTTP, encriptamos los mensajes a base 64 tanto para el envío de mensajes como para las contraseñas en el archivo.

  Respecto al borrado de canales, sólo el usuario que haya creado el canal puede borrarlo.

Ventajas y desventajas de nuestra arquitectura:

- Ventajas:

El uso de los hashes maps facilita el acceso y la manipulación de los mensajes, además la creación y el borrado de canales también se ve favorecido por esta estructura de datos.

- Desventajas:

El manejo de los mensajes se hace en memoria, por lo que los mensajes son todos temporales y en caso de que se caiga el servidor por algún motivo, se perderían todos los mensajes.

El manejo de los mensajes que envía el usuario desde consola se hace por métodos POST.

El login del usuario es muy sencillo y se ubica en un archivo txt en donde se encuentra el servidor.

Diseño

- **Elementos de la Infraestructura:**

Hardware: instancia en la nube de aws ec2

Software: Node.js

Comunicación: Protocolos en HTTP

- **Definición de Niveles de Arquitectura del Sistema**

Arquitectura de dos niveles(cliente-servidor) donde tenemos un módulo cliente que es el encargado de comunicarse el cliente con la otra capa que haría de servidor a través de la consola

El otro modulo que haría como servidor es el encargado de administrar todas las operaciones de los canales

En vez de utilizar una base de datos utilizamos un documento de texto que guarda los usuarios registrados

- **¿Por qué se escogió esta arquitectura?**

Porque entre las 2 que se nos dieron a implementar, esta era la más escalable y queríamos ponernos ese reto de crear algo que nunca habíamos hecho..

- **Desarrollo**

-Instancia ec2:

Luego pasamos a crear una instancia de ec2 lanzando una instancia de amazon Linux 2 asociando a nuestra vpc, nuestro subnet y una vez lanzada guardamos la claves.pem, creamos una ip elástica y se la asociamos a esta instancia de ec2

[1] podrás encontrar más información de como lanzar una instancia

-Conexión con la instancia

Para la conexión de la instancia utilizamos un software llamado Putty Aquí encuentras todos los pasos “[2]”

-**LogIn y Register:**

Se usa login para utilizar un usuario y contraseña que se encuentren en el archivo txt, que será recorrido para verificar la información.

Para el registro, se usa el comando register y luego se pide el usuario y contraseña, si es exitoso, el usuario estará loggeado y se escribirá persistentemente esta información en el archivo txt.

En este apartado lo que hicimos fue que al momento de registrar lo registrábamos en un documento txt con la contraseña codifica en base64 antes verificando que no existiera ese usuario, a la hora del login lo que hicimos fue coger el documento txt buscar el usuario y coger la contraseña encontrada para decodificarla y ver si coincide con la que el usuario metió para poder ingresar

Figura 1

Código de codificar y decodificar mensajes

```
function encode(message){
        var encode = Buffer.from(message).toString('base64');
        return encode;
}
function decode(encode){
        var decode = Buffer.from(encode,'base64').toString();
        return decode;
```

Figura 2

Código de login

```
case "login":
        fs.readFile("/home/ec2-user/Proyectol/src/auth.txt", 'utf8', (error, datos) => {
                if (error) throw error;
                datos = datos.split(/\n/g);
                var i = 0;
                while (user == false && i < datos.length) {
                        args2 = datos[i].split(/\s+/);
                        var passwordDecode = args2[1];
                        passwordDecode = decode(passwordDecode);
                        if (username == args2[0] && password == passwordDecode) {
                                user = true;
                                user_id = i;
                                ++cont;
                        }
                        i++;
                }
                if (user == true) {
                        AuthUser = username;
                        console.log("Log-in success");
                } else {
                        console.log("User unfound");
                }
        });
        break;
```

-Manejo de canales y mensajes:

En este apartado lo que hicimos fue utilizar una estructura de datos que nos facilitara el acceso a los canales, en este caso, el planteamiento de la lógica se hizo de la siguiente manera: Se pensó en crear un hashmap global (channelMap) que tendrá como keys los channels definidos y creados por los usuarios, dentro de cada canal habrá un hashmap diferente que se identificarán con las keys como el user_id de la persona, lo que a su vez tendrá dentro una pila, que guardará los mensajes de cada usuario. Para el método de “Borrar canal” creamos un hashmap adicional (ChannelOwner) que funciona de la siguiente manera: Cuando el usuario crea un canal, este hashmap obtendrá como key el nombre del canal y como valor el id del usuario que la creó, esto con el fin de tener una comprobación de que para borrar el canal debe ser el usuario que la creó.

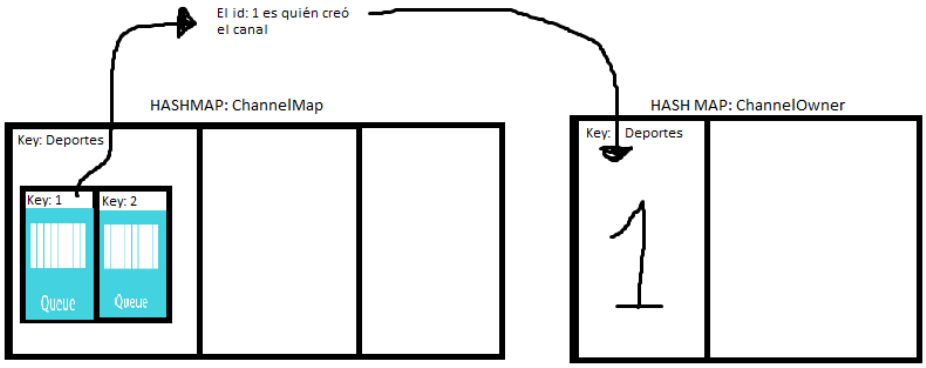
Figura 3

Declaración de hashmap

```
var channelMap = new Hashmap();
var channelOwner = new Hashmap();
```

Figura 4

Representación del hasmap



Se crearon tres métodos encargados de Crear canal, enviar mensajes, borrar el canal y obtener los mensajes del canal de la siguiente manera:

- **Crear canal:** En el cual se hace la verificación correspondiente de que el canal no exista para poder crearlo.

Figura 5

Código de creación de un canal úbicado en index.js

```
async function createChannel(req, res) {
        try {
                await bodyParser(req);
        }
        catch (error) {
                console.log("error en bodyP");
        }
        let channel = req.body.channel;
        let user_id = req.body.user_id;
        if (channelMap.has(channel)=== false){
                channelMap.set(channel, new Hashmap());
                channelMap.get(channel).set(user_id, new Queue());}
                channelOwner.set(channel,user_id);
                console.log("nombre canal creado: "+channel+ " por el usuario "+ user_id);
        else {
                console.log("channel exists");
        }
}
```

- **Enviar Mensajes:** Hace la correspondiente verificación de que primero el canal exista y que segundo la persona tenga su pila creada o explicado de otra manera: esté suscrita, en caso de no estarlo le crea la pila correspondiente y hace push del message enviado.

Figura 6

Código de enviar mensajes en index.js

```
async function MessageHandler(req, res) {
        try {
                await bodyParser(req);
                res.writeHead(200, { 'Content-Type': 'application/json' });
                res.write(JSON.stringify({ message: 'Mensaje recibido' }));
                res.end();
        } catch (error) {
                res.writeHead(200, { 'Content-Type': 'text/plain' });
                res.write(error);
                res.end();

        }
        let channel = req.body.channel;
        let user_id = req.body.user_id;
        let message = req.body.to_send;
        if(channelMap.has(channel) === true) {
                if(channelMap.get(channel).has(user_id) === false) {
                        channelMap.get(channel).set(user_id, new Queue());
                }
                let keys = channelMap.get(channel).keys();
                //console.log(channelMap.get(channel).get(user_id));
                for(i = 0; i < keys.length;i++) {
                        j= keys[i];
                        channelMap.get(channel).get(j).push(message);
                }
        }else{
                console.log("El canal no existe");
        }
```

- **Borrar Canal:** En borrar canal se hace una primera verificación de que el canal exista, esto con el fin de evitar un error, puesto que, si no existe, no tiene nada que borrar, el siguiente “filtro” es de si fue la persona quién creó el canal que quiere borrarlo, si es así, se borra, de lo contrario, enviará un mensaje de que no es el dueño del canal.

Figura 7

Código de borrar un canal en index.js

```
async function deleteChannel(req, res) {
        try {
                await bodyParser(req);
        }
        catch (error) {
                console.log("error en bodyP");
        }
        let channel = req.body.channel;
        let user_id = req.body.user_id;
        if (channelMap.has(channel)){
                if (channelOwner.get(channel) === user_id)? channelMap.delete(channel): console.log("Usted no es el dueño");
                //If then else
        }
        else {
                console.log("El canal no existe");
        }

        console.log("nombre canal eliminado: "+channel + "por el usuario" + user_id );
}
```

- **Obtener mensajes:** El siguiente método tiene como verificación si el canal existe, y como segundo si en el canal el usuario tiene una pila de mensajes, puesto que el get se hace es de la pila del usuario que la solicita, en caso de ser así, retornará el arreglo/pila de mensajes del canal correspondiente que se ve identificado con su user_id. Si el usuario NO está suscrito, deberá hacerlo con el comando SEND.

Figura 8

Código para obtener mensajes de los canales en index.js

```
async function getMessages(req, res) {
        try {
                await bodyParser(req);
                res.writeHead(200, { 'Content-Type': 'application/json' });
                //res.write(JSON.stringify({ message: 'Mensaje recibido' }));
                res.end();
        } catch (error) {
                res.writeHead(200, { 'Content-Type': 'text/plain' });
                res.write(error);
                res.end();

        }
        let channel = req.body.channel;
        let user_id = req.body.user_id;
        if (channelMap.has(channel)){
                if (channelMap.get(channel).has(user_id)) {
                        messages = channelMap.get(channel).get(user_id)['jobs'];
                        console.log(messages);
                }
                else {
                        console.log("No esta suscrito");
                }
        }
        else {
                console.log("El canal no existe");
        }

}
```

- **Conclusión**

Es de suma importancia el planteamiento de la lógica del programa antes de entrar directamente a la programación, esto te facilitará el manejo y la definición de todos los módulos que requieres y la implementación en sí.

El trabajo en equipo es de suma importancia en estos proyectos, en nuestro caso, el reparte equitativo de tareas fue vital para la realización del proyecto.

- **Bibliografía**

[1] “Tutorial: Cómo comenzar con instancias Amazon EC2 Linux - Amazon Elastic Compute Cloud.” https://docs.aws.amazon.com/es_es/AWSEC2/latest/UserGuide/EC2_GetStarted.html#ec2-connect-to-instance-linux (accessed Feb. 24, 2021).

[2] “Conectarse a la instancia de Linux desde Windows mediante PuTTY - Amazon Elastic Compute Cloud.” https://docs.aws.amazon.com/es_es/AWSEC2/latest/UserGuide/putty.html (accessed Feb. 24, 2021).