GRADO EN INGENIERÍA INFORMÁTICA

DESARROLLO DE SISTEMAS DISTRIBUIDOS

UNIVERSIDAD
DE GRANADA

P4 : NodeJs. Desarrollo de Sistemas Distribuidos

Juan Miguel Acosta Ortega (*acostaojuanmi@correo.ugr.es*)

27 de mayo de 2024

Índice

Objetivos

La primera parte consiste en seguir los pasos para implementar y probar los ejemplos del guión. Se ha de demostrar que funcionan y explicar qué ocurre en ellos.

La segunda parte consiste en implementar un sistema domótico con esta tecnología siguiendo las restricciones y requisitos descritos a continuación. Se ha de suponer un sistema domótico básico compuesto de dos sensores (luminosidad y temperatura), dos actuadores (motor persiana y sistema de Aire/Acondicionado), un servidor que sirve páginas para mostrar el estado y actuar sobre los elementos de la vivienda. Además dicho servidor incluye un agente capaz de notificar alarmas y tomar decisiones básicas.

1. Primera parte de la práctica : Revisión de ejemplos

Antes de comenzar con la revisión de ejemplos de han de adquirir e instalar las dependencias básicas para ello.

En primer lugar instalaremos Node.js, el cuál es un entorno de ejecución de un solo hilo, de código abierto y multiplataforma para crear aplicaciones de red y del lado del servidor rápidas y escalables. Se ejecuta en el motor de ejecución de JavaScript V8, y utiliza una arquitectura de E/S basada en eventos y sin bloqueos, lo que la hace eficiente y adecuada para aplicaciones en tiempo real.

A continuación, como tanto para los ejemplos como para el futuro sistema domótico necesitamos hacer uso de comunicaciones rápidas en tiempo real, instalaremos Socket.io. Es una biblioteca basada en eventos para aplicaciones web en tiempo real. Permite la comunicación bidireccional en tiempo real entre clientes y servidores web. Consta de dos componentes: un cliente y un servidor. Ambos componentes tienen una API casi idéntica.

Por último para poder tanto almacenar como crear/actualizar y acceder a los datos necesitamos un SGBD. Para este tipo de tareas es una muy buiena opción hacer uso de BD no relacionales. En este caso usamos MongoDB, un sistema de base de datos NoSQL, orientado a documentos y de código abierto.

Una vez hayamos comprobado que todo funciona correctamente (comprobamos las instalaciones, las versiones, acceso a la BD ...) procedemos al estudio de los ejemplos.

1.1. Primer ejemplo: Hola Mundo Cliente-Servidor

En este primer ejemplo creamos un servidor HTTP básico en Node.js (haciendo uso del módulo "http "). Usamos el método createServer() al que se le pasa una función "callback "que será ejecutada cada vez que recida una solicitud HTTP.

El cuerpo de dicha función se encarga tanto de escribir por consola los encabezados de la petición (headers) como de enviar como respuesta un Hola mundo. Toda la escucha se establece en el puerto 8080.

Ejecutamos el ejemplo con "node helloworls.js "y al mandar una petición a "localhost:8080 "recibimos lo siguiente:

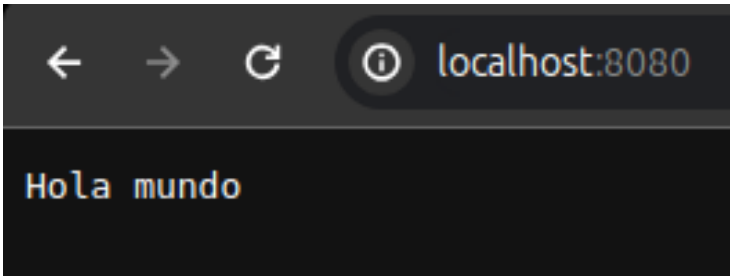

Figura 1: Primer ejemplo ejecutado.

Además podemos ver en consola los headers:

```
{
  host: 'localhost:8080',
  connection: 'keep-alive',
  'sec-ch-ua': '"Google Chrome";v="123", "Not:A-Brand";v="8", "Chromium";v="123"',
  'sec-ch-ua-mobile': '?0',
  'sec-ch-ua-platform': '"Linux"',
  'upgrade-insecure-requests': '1',
```

```
    'user-agent': 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/123.0.0.0 Safari/537.36',
    'sec-purpose': 'prefetch;prerender',
    purpose: 'prefetch',
    accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,
  image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7',
    'sec-fetch-site': 'none',
    'sec-fetch-mode': 'navigate',
    'sec-fetch-user': '?1',
    'sec-fetch-dest': 'document',
    'accept-encoding': 'gzip, deflate, br, zstd',
    'accept-language': 'es-ES,es;q=0.9',
    cookie: 'Studio-c10166b=0f556a5e-d3e1-495a-b124-228ce91da12e'
  }
```

Listing 1: Headers del ejemplo 1

1.2. Segundo ejemplo: Calculadora distribuida

Este segundo ejemplo es similar al anterior en cuanto a la creación del servidor y forma de enviar una respuesta. Sin embargo la primera peculiaridad de este servidor es que al haber una petición "request "primero se trata la url de modo que la obtenemos de la petición , le quitamos el primer caracter (/), la convertimos en un array cuyos componentes son lo contenido entre barras, y si son 3 (suponemos que serán válidos) serán los parámetros para la función calcular que nos proporcionará el dato a enviar en respuesta.

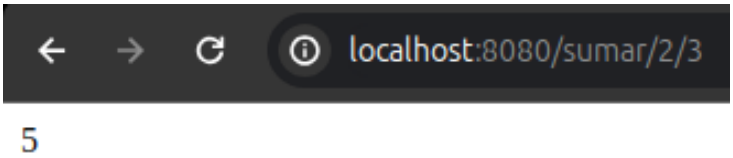

Figura 2: Segundo ejemplo ejecutado.

1.3. Tercer ejemplo: Calculadora distribuida con interfaz web

Para este tercer ejemplo importamos dos módulos más necesarios para trabajar con ficheros, pues en este ejemplo se manda del servidor un fichero html a modo de interfaz de usuario. Estos son "join de node:path "y "readFile de node:fs ".

Además al obtener la url en este ejemplo no comenzamos a tratarla como en el anterior, si no que la utilizamos más bien para comprobar si estamos en raíz (solo una barra) para enviar el fichero html con el formulario, o si por el contrario ha se ha enviado el formulario (suponemos que si hay 3 parámetros separados pro barras será una respuesta válida) y se ha de proceder con el envío del cálculo.

Esta es la forma utilizada en el ejemplo para servir la interfaz al cliente:

```
1
2   const filename = join(process.cwd(), url);
3
4   readFile(filename, (err, data) => {
5       if(!err) {
6           response.writeHead(200, {'Content=Type': 'text/html; charset=utf=8'});
7           response.write(data);
8       } else {
9           response.writeHead(500, { Content =Type: text /plain });
```

```
10              response.write(`Error en la lectura del fichero: ${url}`);
11      }
12      response.end();
```

Listing 2: Lectura y envío de calc.html

Y este sería un ejemplo de ejecución exitosa:

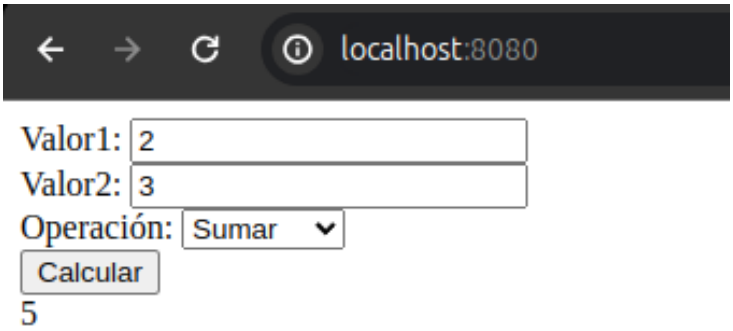

Figura 3: Tercer ejemplo ejecutado.

La interfaz web de usuario por su parte contiene un pequeño script javascript que recogerá los valores de los input del formulario para crear una petición al servidor con una url de formato indicado (conforme al ejemplo2).

1.4. Cuarto ejemplo: Aplicaciones en tiempo real con Socket.io

El siguiente ejemplo muestra la implementación sobre Socket.io de un servicio que envía una notificación que contiene las direcciones de todos los clientes conectados al propio servicio. Esta se envía a todos los clientes suscritos cada vez que uno nuevo se conecta o desconecta. El envío a todos los clientes se realiza llamando a la función "emit "sobre el conjunto de clientes conectados (io.sockets.emit(...)). Además, cuando el servicio recibe un evento de tipo "output-evt" "le envía al cliente el mensaje "Hola Cliente! ".

Para hacer uso de esta funcionalidad se importa el nuevo paquete "Server de socket.io ".

En este ejemplo también se le proporciona una interfaz web en forma de archivo html, la diferencia es que el servicio sólo se puede utilizar accediendo a la raíz (/).

La diferencia es que se crea un objeto Server que se vincula al servidor HTTP existente (conexión WebSocket). Además se establece un manejador de eventos para cuando un cliente se conecta al servidor "io.sockets.on('connection' ... ".

Dentro de este manejador se obtiene la IP y puerto de los clientes que hayan hecho una petición al servidor, se agrega la información al array conteniente de todos los clientes activos (allCLients) y se emite al cliente mediente un evento llamado 'all-connections' esa información. A continuación se establece otro manejador de eventos para el evento personalizado 'output-evt' que el cliente puede emitir para recibir un "Hola cliente! "

Se establece por último otro manejador de eventos sobre el evento default 'disconnect' para actualizar así el array allCLients borrando al usuario desconecxtado y mandándolo actualizado.

Asimismo en el cliente se encuentran tanto las funciones necesarias para dinamizar el DOM como los "manejadores "de eventos necesarios para comunicarse con el servidor.

A continuación se muestra el ejemplo de recepción por parte del tercer usuario en mandar una petición:

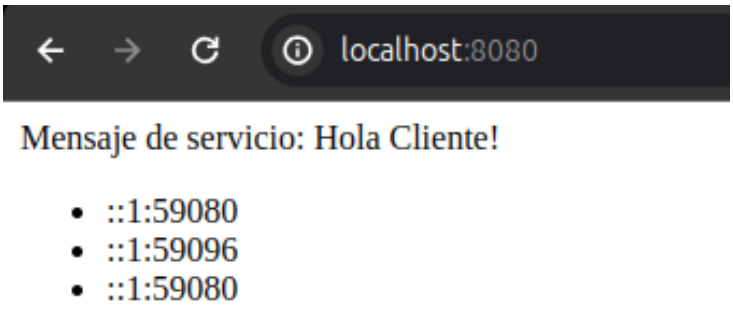

Figura 4: Cuarto ejemplo ejecutado.

1.5. Quinto ejemplo: Uso de MongoDB desde Node.js

Por último y para el manejo de la base de datos se importa el módulo "MongoClient de mongodb ".
Este ejemplo es muy similar al anterior, la diferencia está en el manejo de eventos personalizados relacionados con la inserción y consulta de información en la BD. Antes de establecer estos con conectamos a una base de datos y creamos o usamos una colección.

```
1
2   MongoClient.connect("mongodb://localhost:27017/").then((db) => {
3       const dbo = db.db("baseDatosTest"); // Nombre de la base de datos
4       const collection = dbo.collection("test"); // Nombre de la colecci n
5
6       const io = new Server(httpServer);
7       io.sockets.on('connection', (client) => {
8           client.emit('my-address', {host:client.request.socket.remoteAddress, port:client.
    request.socket.remotePort});
9           client.on('poner', (data) => {
10              collection.insertOne(data, {safe:true}).then((result) => {});
11          });
12          client.on('obtener', (data) => {
13              collection.find(data).toArray().then((results) => {
14                  client.emit('obtener', results);
15              });
16          });
17      });
```

Listing 3: Eventos relacionados con la BD

De la misma manera en la interfaz HTML se crean los mismos manejadores de eventos para ejecutar las instrucciones adecuadas tanto para mandar información (como la fecha) y obtener datos y actualizar el DOM.

Ejemplo de ejecución del tercer usuario:

Figura 5: Quinto ejemplo ejecutado.

2. Segunda parte de la práctica : Sistema domótico en Node.js

En esta sección de la práctica se crea un sistema domótico compuesto de tres sensores (de temperatura, luminosidad y viento), y tres actuadores (A/C, persianas y toldo), una entidad central (servidor) que sirve la interfaz gráfica para mostrar y modificar información en el cliente e interactuar con las demás entidades del sistema (base de datos, agente ...), y un agente que se encargará de dictar cuando se deben manipular los diferentes actuadores según unos umbrales y unos valores de los sensores mandados desde el servidor.

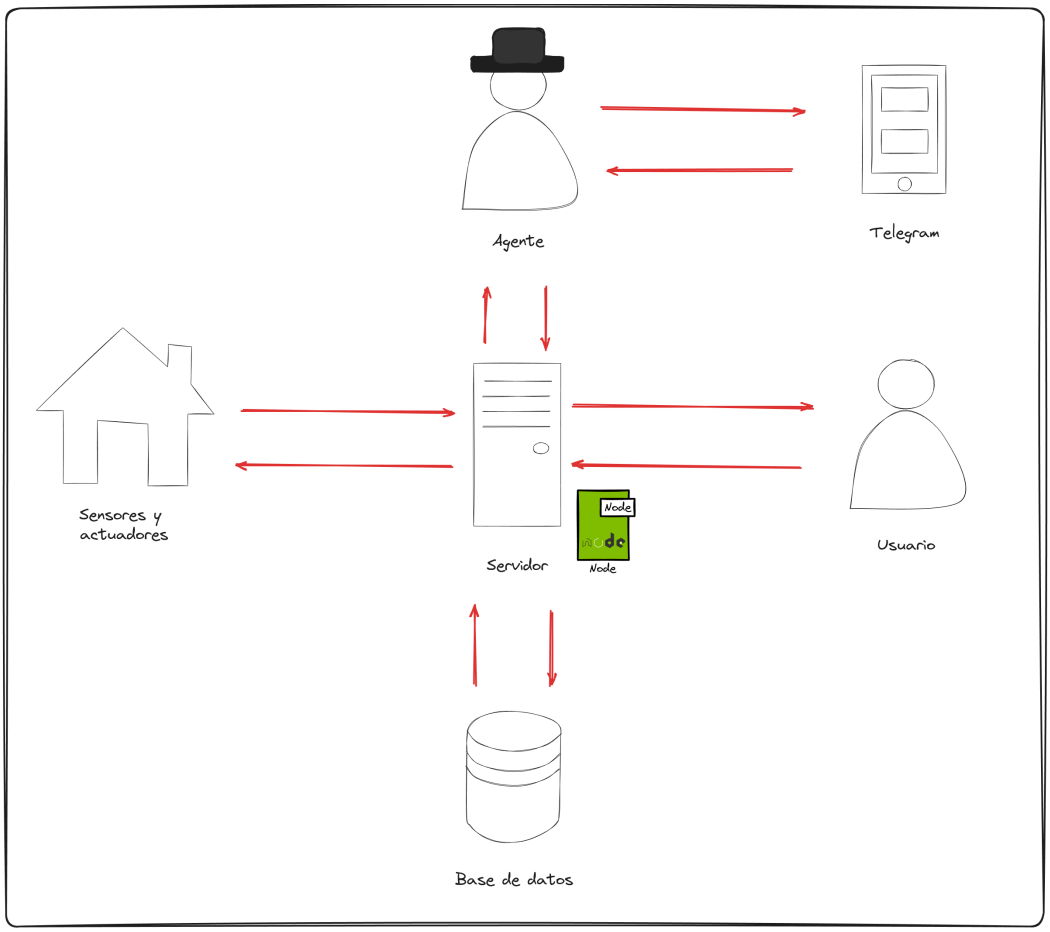

Figura 6: Esquema del sistema domótico.

Como vimos en los ejemplos anteriores podemos establecer acciones determinadas a las peticiones del cliente. En este caso cuando accedemos a la raíz del servidor (localhost:8080) servimos una única página con toda la lógica del cliente embebida para evitar múltiples peticiones. En esta página tendremos la información de los sensores, de los actuadores, de los umbrales para la activación / desactivación de los actuadores , y los eventos recientes.

En primer lugar se detalla la sección de los sensores:

Figura 7: Sección de los sensores.

En esta sección vemos tanto los valores más recientes enviados al servidor (e insertados en la base de datos) correspondientes con los sensores, y un sistema para mandar nuevos datos al servidor (para simular la recogida de datos de sensores reales).Al mandar nuevos datos se actualizarán los valores actuales con los recién mandados.

En la siguiente sección podemos ver el estado actual de los actuadores, y podemos activarlos/desactivarlos manualmente. Además en caso de haber alguna alerta podríamos verla en esta sección:

Figura 8: Sección de los actuadores.

Inmediatamente abajo podemos encontrarnos la sección de los umbrales, en esta al igual que en la sección de los sensores podemos visualizar los valores actuales, y actualizarlos mandando un nuevo valor:

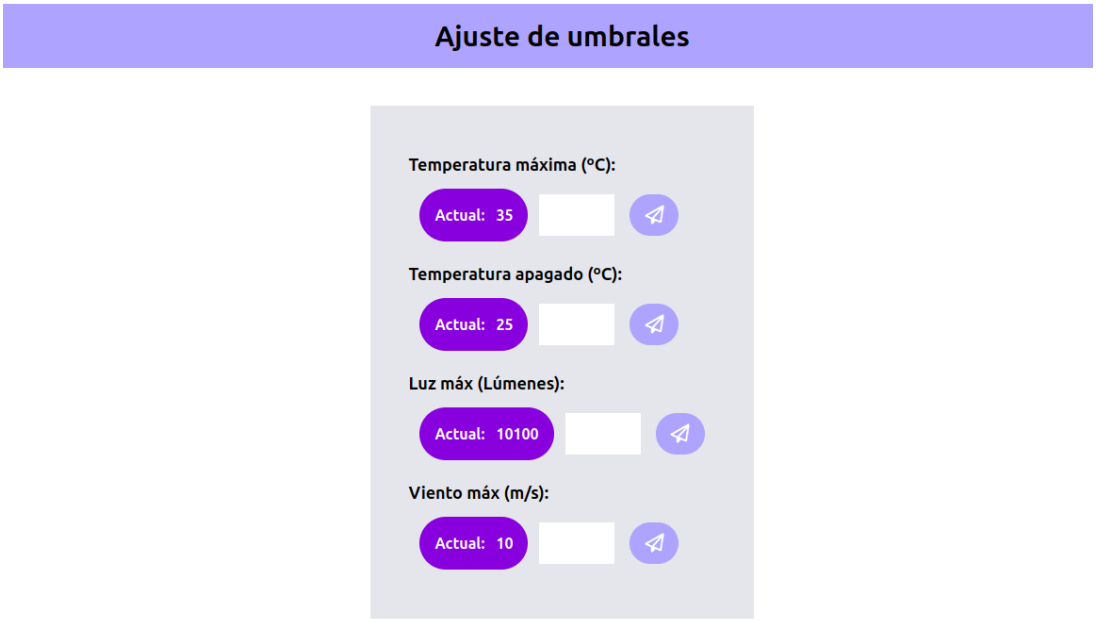

Figura 9: Sección de los umbrales.

Por último tenemos la sección de eventos recientes, en esta sección podemos visualizar los eventos recientes. Además se actualizan en tiempo real.

Figura 10: Sección de los eventos.

Una vez hemos revisado el diseño y funcionamiento de la interfaz de usuario pasaremos a ver realmente con qué eventos y cómo se produce la comunicación entre las entidades.

2.1. El servidor

En primer lugar repasaremos a fondo el servidor. Este se conecta a una base de datos no relacional MongoDB y accede a tres colecciones, una "Datos "en la que se insertan los datos referentes a los sensores y actuadores, otra de eventos y otra de umbrales.

El servidor en primer lugar está suscrito al evento "connection ", y en este sucede casi toda la lógica. En primer lugar se le manda a todos los clientes (navegadores y agentes) la información sobre los umbrales en base de datos, luego identificamos qué tipo de cliente es el que se ha conectado, esto lo podemos saber porque el agente hace petición al servidor con la "query "?clientType=agent. Distinguimos tipo de clientes porque al conectarse el agente simplemente enseñamos dirección y puerto de este, mientras que a los navegadores les mandamos mucha más información con los eventos "actualizarClienteEntrante y actualizarEventos ".

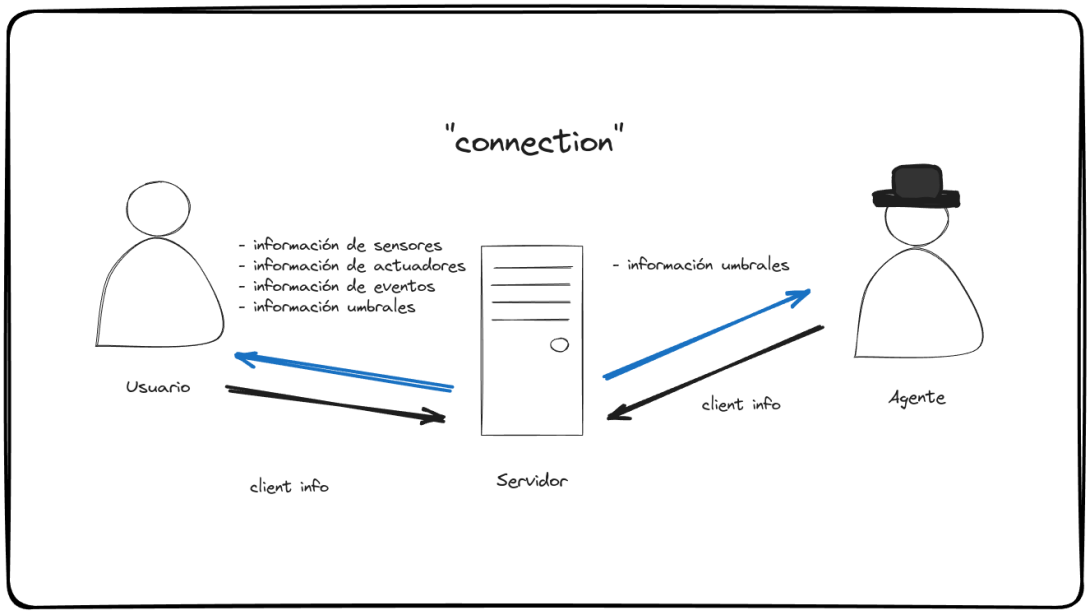

Figura 11: On connection en el servidor.

Además nos suscribimos a varios eventos que sucederás durante la ejecución :

1. "disconnect "Se notificará la desconeción del cliente
2. "cambiarMaxTemp, cambiarMinTemp ... "Este tipo de eventos actualizan los valores concretos de la última tupla de umbrales en la colección indicada.
3. "insertarDatos "Este evento se encarga de insertar los datos enviados sobre los sensores en base de datos. Uso un conjunto de promesas para asegurarme de que primero tengo los datos (cuando mando un campo vacío cojo su último registro en bd) antes de insertarlos. Esto lo consigo usando "Promise.all(promises).then(=>...) ".
   Aunque es similar a concatenar .then creo que esta estructura es más legible en algunos casos.

   Tras esto emitimos el evento "actualizarSensores "a todos los clientes. El agente activará un mecanismo para comprobar los datos enviados y si se ha de activar o desactivar algún actuador conforme a los umbrales, y los clientes en navegador actualizarán los valores de los sensores.

```
1
2    client.on('insertarDatos', (data) => {
3
4                console.log('Datos recibidos del cliente: ' + JSON.stringify(data));
```

```
5
6               let promises = [];
7
8               if (data.viento === "") {
9                   promises.push(collection.find().sort({ $natural: -1 }).limit(1).
    toArray().then((dataViento) => {
10                      data.viento = dataViento[0].viento;
11                  }));
12              }
13
14              if (data.temperatura === "") {
15                  promises.push(collection.find().sort({ $natural: -1 }).limit(1).
    toArray().then((dataTemperatura) => {
16                      data.temperatura = dataTemperatura[0].temperatura;
17                  }));
18              }
19
20              if (data.luminosidad === "") {
21                  promises.push(collection.find().sort({ $natural: -1 }).limit(1).
    toArray().then((dataLuminosidad) => {
22                      data.luminosidad = dataLuminosidad[0].luminosidad;
23                  }));
24              }
25
26              Promise.all(promises).then(() => {
27                  collection.insertOne(data, { safe: true }).then(() => { });
28                  console.log('Datos insertados en la base de datos con  xito . Datos:
    ' + JSON.stringify(data));
29                  io.sockets.emit('actualizarSensores', data);
30              });
31
32          });
```

Listing 4: Insertar datos (Promise.all())

4. "actualizarActuadores "Este evento va a ser llamado con el agente, y será el que almacene los nuevos estados de los actuadores y emita este mismo evento al cliente para que actualice su estado en la interfaz.

5. "almacenarEvento "Este evento también es activado por el agente, y es que este hace un emit del mismo cuando sucede un evento (cambio en actuador por umbrales). Del mismo modo hace un emit de "actualizarEventos "para que los actualicen los clientes en navegador.

6. "insertarDatosSinAgente "Este evento se activa desde los clientes de navegador, ya que se pueden activar o desactivar los actuadores manualmente y no hace falta que actúe el agente (No hace falta comprobar los datos ni los umbrales), pero sí se generan otro tipo de eventos (manipulacioens manuales).

```
1
2   client.on('insertarDatosSinAgente', (data) => {
3               console.log('ACTIVACI N MANUAL DE ACTIVADOR ' + JSON.stringify(data));
4
5               collection.find().sort({ $natural: -1 }).limit(1).toArray().then((result)
     => {
6                   return result[0];
7               }).then((result) => {
8                   collection.insertOne(data, { safe: true });
9                   console.log('Datos insertados en la base de datos con  xito . Datos:
    ' + JSON.stringify(data));
10
11                  if (result.estadoAC !== data.estadoAC) {
12                      return collection2.insertOne({ evento: 'Aire acondicionado o
    calefacci n accionado manualmente', fecha: new Date() }, { safe: true });
13                  }
14                  else if (result.estadoPersiana !== data.estadoPersiana) {
15                      return collection2.insertOne({ evento: 'Persiana accionada
    manualmente', fecha: new Date() }, { safe: true });
16                  }
17                  else if (result.estadoToldo !== data.estadoToldo) {
```

```
                        return collection2.insertOne({ evento: 'Toldo accionado manualmente', fecha: new Date() }, { safe: true });
                    }
                }).then(() => {
                    return collection2.find({ evento: { $exists: true } }).sort({ $natural: -1 }).limit(10).toArray();
                }).then((dataEventos) => {
                    io.sockets.emit('actualizarEventos', dataEventos);
                    io.sockets.emit('actualizarClienteEntrante', data);
                });
            });
```

Listing 5: Insertar datos sin agente (Concatenando .then())

2.2. Cliente navegador

Este se suscribe a los eventos mencionados anteriormente , los cuales son:

1. "actualizarUmbrales "
2. "actualizarClienteEntrante "
3. "actualizarSensores "
4. "actualizarActuadores "
5. "actualizarEventos "Este evento también es activado

El resto de lógica de este tipo de clientes es la encargada de actualizar el DOM con los estilos y la información conveniente en cada momento y de mandar la información al servidor mediante "eventListener ".

Para dar estilo a la página he decidido usar Tailwind CSS, un framework de CSS que hace tiempo que quería probar. Para este tipo de proyectos pequeños y con los estilos embebidos en el mismo fichero viene muy bien y hace más rápido el desarrollo. Además aporta la facilidad de integrar pequeñas animaciones como la del ventilador cuando se activa.

```

    function mandarDatos() { //Manda los datos al servidor

        console.log("Mandando datos");

        var temperatura = document.getElementById("temperatura").value;
        var luminosidad = document.getElementById("luminosidad").value;
        var viento = document.getElementById("viento").value;
        var fecha = new Date();

        var estadoAC = document.getElementById("sensorAC").innerHTML;
        var estadoPersiana = document.getElementById("sensorPersiana").innerHTML;
        var estadoToldo = document.getElementById("sensorToldo").innerHTML;

        socket.emit('insertarDatos', {
            temperatura: temperatura, luminosidad: luminosidad, viento: viento,
            fecha: fecha, estadoAC: estadoAC, estadoPersiana: estadoPersiana, estadoToldo: estadoToldo
        });
    }
```

Listing 6: Función que manda los datos insertados de los sensores y actuales de los actuadores al servidor

2.3. Agente

El agente se encarga principalmente de comprobar si se ha de almacenar un nuevo evento conforme a los datos entrantes y los umbrales obtenidos mediante el servidor. Es por eso que sólo está suscrito a dos eventos, uno que actualiza los umbrales, y otro que es el que comprueba si se ha de generar un evento.

Esta entidad se encarga realmente de mandar tanto los eventos generados como los nuevos estados de los actuadores para que el servidor sea el que inserte en Base de Datos y se comunique con los demás clientes.

```
1
2    if (Number(temperatura) > Number(UMBRAL_TEMPERATURA_CALOR)) {
3
4        if (estadoAC === 'Apagado') {
5            console.log('Temperatura extrema, se enciende el aire acondicionado o la
     calefacci n');
6            datosAMandar.estadoAC = 'Encendido';
7            socket.emit('almacenarEvento', { evento: 'Aire acondicionado o calefacci n
     encendidos por temperatura extrema', fecha: new Date() })
8        }
9
10   } else if (Number(temperatura) < Number(UMBRAL_TEMPERATURA_FRIO)) {
11
12       if (estadoAC === 'Encendido') {
13           console.log('Temperatura normal, se apaga el aire acondicionado o la calefacci n'
     );
14           datosAMandar.estadoAC = 'Apagado';
15           socket.emit('almacenarEvento', { evento: 'Aire acondicionado o calefacci n
     apagados por temperatura normal', fecha: new Date() })
16       }
17   }
```

Listing 7: Ejemplo de creación de eventos y cambio en actuadores

Las alertas se muestran en el cliente cuando hay un evento natural (un actuador se activa/desactiva de manera no manual).

```
1
2    if (datos.estadoAC == "Encendido") { //A adir clases si no las tienen
3            if(!datos.manual) document.getElementById("alertaAC").classList.remove("hidden");
4            document.getElementById("simboloAC").classList.add("fa-spin");
5            document.getElementById("sensorAC").classList.add("on");
6            document.getElementById("sensorAC").classList.remove("off");
7            document.getElementById("manualAC").checked = true;
8        } else {
9            if(!datos.manual) document.getElementById("alertaAC").classList.add("hidden");
10           document.getElementById("simboloAC").classList.remove("fa-spin");
11           document.getElementById("sensorAC").classList.add("off");
12           document.getElementById("sensorAC").classList.remove("on");
13           document.getElementById("manualAC").checked = false;
14       }
```

Listing 8: Ejemplo de cambio en actuador en cliente, y generar/esconder alerta

Para levantar tanto el servidor como el agente he usado una librería externa "concurrently"Este paquete es una herramienta que se utiliza para ejecutar múltiples comandos npm simultáneamente. Es útil en situaciones en las que se necesita ejecutar varios scripts de Node.js al mismo tiempo. Para iniciar estos simplemente se ha de ejecutar "npm start", y esto ejecuta este pequeño script en package.json:

```
1  "scripts": {
2    "start": "concurrently \"node server-domotica.js\" \"node agente.js\""
3  }
```

Listing 9: Script levantar servidor y agente

Además para que el agente actúe como cliente del servidor he utilizado la librería "socket.io-client", este paquete es el cliente de Socket.IO, que permite establecer conexiones de socket en tiempo real entre el cliente y el servidor. Se utiliza en el lado del cliente para conectarse a un servidor Socket.IO y escuchar y emitir eventos.

Se conecta de la siguiente forma:

```
import { io } from 'socket.io-client';

const url = 'http://localhost:8080?clientType=agent';
const socket = io(url);
```

Listing 10: Conexión del agente con el servidor

2.4. Adicional : Mandar notificaciones a Telegram

Para la creación del Bot he usado "BotFather", un bot de Telegram oficial de la red social para la creación de bots. Tras esto sólo tuve que configurar su nombre y se me proporcionó la key del bot.
Para gestionar esta uso un paquete externo llamado "dotenv"para manejar variables de entorno. Además para usar el bot y configurar su comportamiento en Nodejs también he usado la librería de Telegram para hacerlo, Telegraf.

Los datos que se mandan a Telegram son los datos recibidos el agente (se guardan los últimos de cada), y al hacer /start en el bot, manda un menú con las acciones posibles:

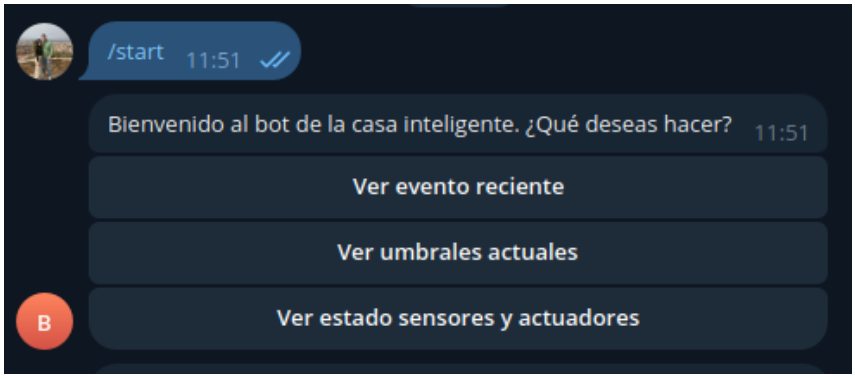

Figura 12: Menú del bot de Telegram.

```
bot.telegram.sendMessage(ctx.chat.id, bienvenida, {
    reply_markup: {
        inline_keyboard: [
            [
                { text: 'Ver evento reciente', callback_data: 'verEvento' },
            ],
            [
                { text: 'Ver umbrales actuales', callback_data: 'verUmbrales' },
            ],
            [
                { text: 'Ver estado sensores y actuadores', callback_data: '
verSensoresActuadores' },
            ]
        ]
    }

});
```

Listing 11: Menú del bot

Los diferentes botones del bot nos enseñarán la información correspondiente que almacena el agente:

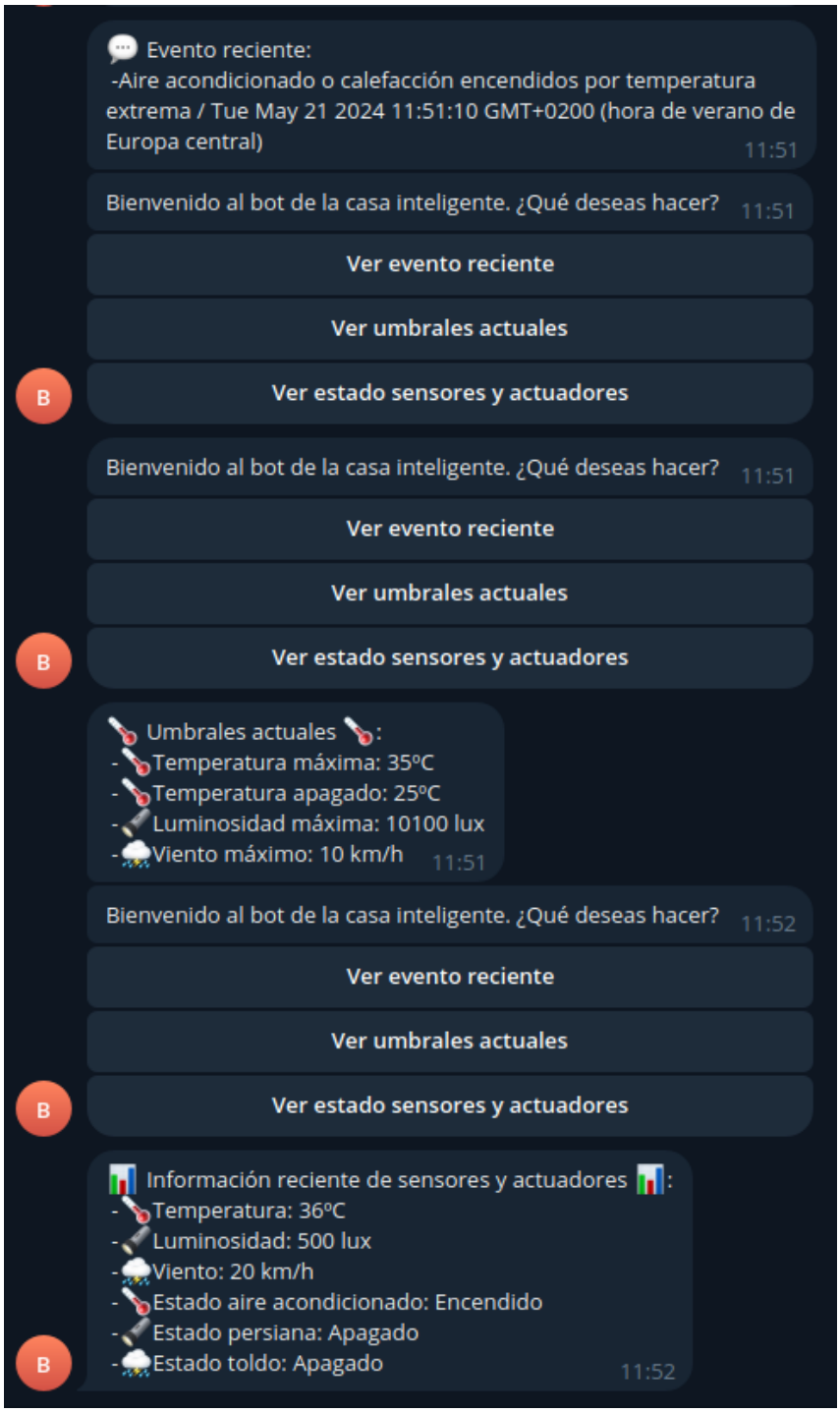

Figura 13: Respuestas del bot.