Índice

1. Requisitos Previos

En este capítulo se describe la instalación de Node.js, Socket.io y MongoDB.

1.1. Instalación de Node.js

Para instalar Node.js se ha de visitar la sección de descargas de la página oficial y descargarse el instalador apropiado en función de nuestro sistema operativo. En el caso de Linux, se aconseja pinchar en la sección *Installing Node.js via package manager* y seleccionar nuestra distribución.

Tras la instalación de Node.js se deberá comprobar que se pueden ejecutar las órdenes "nodejs" y "npm". Esta última orden es el gestor de paquetes de Node.js y nos va a permitir realizar la instalación de aquellos paquetes que queramos utilizar en nuestros proyectos.

1.2. Proyectos en Node.js

Los proyectos de Node.js están organizados en torno a un directorio de trabajo que contendrá todos los archivos necesarios para poder ejecutarlo. Asimismo, los paquetes se instalarán, por defecto, en el directorio desde el cual se lance la ejecución de "npm" y, por consiguiente, serán locales al proyecto en cuestión.

1.3. Instalación de Socket.io

Para instalar Socket.io se deberá ejecutar en el directorio de trabajo del proyecto actual:

```
> npm install socket.io
```

Si todo es correcto, en nuestro directorio del proyecto deberemos tener una nueva carpeta llamada "node_modules" que contendrá una subcarpeta llamada "socket.io". Asimismo, se habrá creado el archivo "package.json" que listará a este paquete como una dependencia:

```
{
  "dependencies": {
    "socket.io": "^4.7.4"
  }
}
```

Este hecho ocurrirá cada vez que se instale un nuevo paquete en el proyecto actual.

1.4. Instalación de MongoDB

La instalación de MongoDB se realizará en dos pasos. En primer lugar se deberá instalar el sistema de gestion de bases de datos MongoDB Community Edition siguiendo la guía de instalacíon de la página oficial.

MongoDB deberá iniciarse automáticamente como un servicio en segundo plano. Podemos comprobar que la instalación ha sido satisfactoria ejecutando la orden "mongosh". Con dicha orden accedemos desde nuestro terminal a una shell que nos permite interactuar con el servidor de MongoDB.

A continuación se deberá instalar en el directorio de trabajo el paquete que nos permitirá desarrollar un cliente Node.js que haga uso de MondoDB:

```
> npm install mongodb
```

1.5. Sistemas de módulos

Node.js soporta dos sistemas de módulos: CommonJS y ECMAScript. Los distintos ejemplos que se listan a lo largo de este documento usan ECMAScript, que es el formato estándar del lenguaje. Aunque existen varios métodos para comunicar a Node.js qué sistema vamos a emplear, una opción es modificar el archivo de configuración "package.json". Concretamente, se añadirá la propiedad "type" con el valor "module".

```
{
  "type": "module",
  "dependencies": {
    "mongodb": "^6.3.0",
    "socket.io": "^4.7.4"
  }
}
```

2. Implementación de Servicios con Node.js

Node.js es una plataforma que permite implementar servicios web haciendo uso del lenguaje de programación JavaScript. Los servicios implementados sobre Node.js tienen un modelo de comunicación asíncrono y dirigido por eventos, tratando de maximizar la escalabilidad y la eficiencia de dichos servicios.

Para comenzar, vamos a analizar un ejemplo sencillo de servicio web escrito en Node.js:

```
1  import http from 'node:http';
2
3  http.createServer((request, response) => {
4          console.log(request.headers);
5          response.writeHead(200, {'Content-Type': 'text/plain'});
6          response.write('Hola mundo');
7          response.end();
8      })
9      .listen(8080);
10
11 console.log('Servicio HTTP iniciado');
```

helloworld.js

Podemos probar su funcionamiento ejecutando desde el terminal:

```
> node helloworld.js
```

Una vez ejecutado el servicio, podremos abrir nuestro navegador y comprobar que, si visitamos la dirección "http://localhost:8080/", nos aparecerá el mensaje "Hola Mundo".

En el código anterior, se comienza importando el módulo "node:http", que permite implementar servicios que sirvan contenidos usando el protocolo http. En Node.js se pueden importar, de forma parcial o completa, tantos módulos como se requieran y hacer uso de ellos de manera combinada.

A continuación se crea un servidor http. El parámetro del constructor de un servidor es, a su vez, una función con dos parámetros: request y response. Dicha función se llamará cada vez que se reciba una petición mediante el protocolo http (p. ej. usando un navegador para acceder al servicio implementado). En el objeto request se codifica el mensaje de petición recibido en el servicio. Dicho mensaje se imprime en la línea 4. Con el objeto response podemos crear una respuesta a la petición recibida. Un ejemplo de respuesta se implementa en las líneas 5-7.

En la línea 9, se hace que el servidor http comience a recibir peticiones en el puerto 8080. Nótese como a continuación se puede ejecutar más código y que, cuando se llega al final, el servicio no termina su ejecución. Esto se debe a que en Node.js la mayoría de las operaciones son no bloqueantes, pero se impide la finalización de un programa mientras existan puertos del sistema operativo en uso. Eso facilita el desarrollo de servicios con una arquitectura dirigida por eventos.

En el siguiente ejemplo se muestra como implementar una calculadora distribuida usando una interfaz tipo REST:

```
import http from 'node:http';

function calcular(operacion, val1, val2) {
    if (operacion == 'sumar') return val1+val2;
    else if (operacion == 'restar') return val1-val2;
    else if (operacion == 'producto') return val1*val2;
    else if (operacion == 'dividir') return val1/val2;
    else return 'Error: Parámetros no válidos';
}

http.createServer((request, response) => {
        let {url} = request;
        url = url.slice(1);
        const params = url.split('/');
        let output='';
        if (params.length >= 3) {
            const val1 = parseFloat(params[1]);
            const val2 = parseFloat(params[2]);
            const result = calcular(params[0], val1, val2);
            output = result.toString();
        }
        else output = 'Error: El número de parámetros no es válido';

        response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
        response.write(output);
        response.end();
    })
    .listen(8080);

console.log('Servicio HTTP iniciado');

```

calculadora.js

Este servicio recibe peticiones REST del tipo "http://localhost:8080/sumar/2/3". Utilizando un navegador se puede comprobar como el servicio devolverá el resultado de la operación solicitada a partir los números facilitados como operandos.

A continuación se muestra una mejora del servicio anterior. Este nuevo ejemplo proporciona una interfaz web para la calculadora en caso de que el usuario acceda desde el navegador a la url "http://localhost:8080". El código fuente de este nuevo servicio es el siguiente:

```
import http          from 'node:http';
import {join}        from 'node:path';
import {readFile}    from 'node:fs';

function calcular(operacion, val1, val2) {
    if (operacion == 'sumar') return val1+val2;
    else if (operacion == 'restar') return val1-val2;
    else if (operacion == 'producto') return val1*val2;
    else if (operacion == 'dividir') return val1/val2;
    else return 'Error: Parámetros no válidos';
}

http.createServer((request, response) => {
        let {url} = request;
        if(url == '/') {
            url = '/calc.html';
            const filename = join(process.cwd(), url);

            readFile(filename, (err, data) => {
                if(!err) {
                    response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
                    response.write(data);
                } else {
                    response.writeHead(500, {"Content-Type": "text/plain"});
                    response.write(`Error en la lectura del fichero: ${url}`);
                }
                response.end();
            });
        }
        else {
            url = url.slice(1);
            const params = url.split('/');
            let output='';
            if (params.length >= 3) {
                const val1 = parseFloat(params[1]);
                const val2 = parseFloat(params[2]);
                const result = calcular(params[0], val1, val2);
                output = result.toString();
            }
            else output = 'Error: El número de parámetros no es válido';

            response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
            response.write(output);
            response.end();
        }
    })
```

```
47      .listen(8080);
48
49  console.log('Servicio HTTP iniciado');
50
```

calculadora–web.js

Se puede observar como se hace uso de un módulo llamado “node:fs”. Este módulo permite realizar operaciones de entrada/salida sobre el sistema de ficheros del servidor. En este caso, se utiliza para leer aquellos almacenados en el directorio de ejecución del servicio y devolverlos al cliente como respuesta. Nótese como la lectura de un fichero se realiza de manera asíncrona, notificándose el fin de lectura en una función pasada como parámetro. También se hace uso de un módulo llamado “node:path” para construir la ruta al archivo a partir de cadenas de caractéres.

Como se ha comentado anteriormente, este servicio dispone de un sencillo cliente web. Su código fuente (“calc.html”) se muestra a continuación:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset=”uft−8”>
5          <title>Calculadora</title>
6      </head>
7      <body>
8          <form id=”calculadora”>
9              Valor1: <input type=”label” id=”val1”><br>
10             Valor2: <input type=”label” id=”val2”><br>
11             Operación:
12             <select id=”operacion”>
13                 <option value=”sumar”>Sumar</option>
14                 <option value=”restar”>Restar</option>
15                 <option value=”producto”>Producto</option>
16                 <option value=”dividir”>Dividir</option>
17             </select><br>
18             <input type=”submit” value=”Calcular”>
19         </form>
20         <span id=”resultado”></span>
21     </body>
22     <script type=”text/javascript”>
23         const calc = document.getElementById('calculadora');
24
25         calc.addEventListener('submit', (e) => {
26             e.preventDefault();
27
28             const serviceURL = document.URL;
29             const val1 = document.getElementById('val1').value;
```

```
30            const val2 = document.getElementById('val2').value;
31            const oper = document.getElementById('operacion').value;
32            const url  = serviceURL + oper + '/' + val1 + '/' +val2;
33
34            fetch(url)
35            .then(response => response.text())
36            .then(response => {
37                let resultado = document.getElementById('resultado');
38                resultado.textContent = response;
39            });
40        });
41    </script>
42 </html>
```

calc.html

El cliente muestra un formulario cuyo envío se realiza mediante un manejador del evento 'submit' implementado en JavaScript. Dicha manejador obtiene los valores introducidos por el usuario en el formulario, crea una petición tipo REST y finalmente envía la petición al servicio de manera asíncrona mediante la función fetch.

La explicación detallada sobre como desarrollar aplicaciones web cliente queda fuera de los ámbitos de este documento. No obstante, para facilitar su implementación, se recomienda el uso del framework jQuery para JavaScript.

3. Aplicaciones en Tiempo Real con Socket.io

Un gran problema que había existido durante años en la web era la imposibilidad de enviar información desde un servicio hacia un cliente sin que el propio cliente la hubiera solicitado previamente. Por ejemplo, en un chat era inviable tecnológicamente notificar a todos los usuarios que un nuevo mensaje había sido enviado por algún usuario. Debido a este problema, se optaba por realizar consultas periódicas desde los clientes hacia los servicios para comprobar si había o no nueva información que mostrar al usuario. Dicho sistema de comunicación (conocido como *polling* o modelo de notificaciones tipo *pull*) es altamente ineficiente e impide que los servicios puedan atender adecuadamente a un número elevado de usuarios de manera simultánea, presentando, por tanto, problemas de escalabilidad. El estándar HTML5 trata de solventar el problema anterior agregando soporte a sockets (*WebSockets*) al lenguaje JavaScript. De esta manera, es posible mantener conexiones permanentes desde el un cliente hacia un servicio y que este último transmita información al cliente cuando sea necesario (notificaciones tipo *push*).

Socket.io es un módulo para Node.js que permite implementar aplicaciones web en tiempo real mediante WebSockets. El modelo de comunicaciones utilizado se conoce como "publish-subscribe". Dicho modelo de comunicaciones, asíncrono y dirigido por eventos, permite que un cliente solicite la recepción de determinadas notificaciones (*suscripción*). El servicio, a partir de ese instante, cuando reciba una notificación, la enviará a todos los suscriptores conectados (*publicación*). Por defecto, Socket.io incorpora, entre otros, los siguientes eventos:

- **'connect'** - Se emite al realizarse una conexión correctamente.
- **'disconnect'** - Notificación de la desconexión entre cliente y servicio.
- **'connection_error'** - Se emite cuando se cierra la conexión de forma inesperada.
- **'error'** - Notificación de un error de conexión por parte del cliente.
- **'reconnect'** - Se emite cuando un cliente se reconecta a un servicio con éxito.
- **'reconnect_attempt'** - Se emite cuando un cliente intenta conectarse de nuevo a un servicio.
- **'reconnect_failed'** - Error de reconexión durante un 'reconnect_attempt'.

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. Dicha notificación 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!".

```
import http          from 'node:http';
import {join}        from 'node:path';
import {readFile}    from 'node:fs';
import {Server}      from 'socket.io';

const httpServer = http
    .createServer((request, response) => {
        let {url} = request;
        if(url == '/') {
            url = '/connections.html';
            const filename = join(process.cwd(), url);

            readFile(filename, (err, data) => {
                if(!err) {
                    response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
                    response.write(data);
                } else {
                    response.writeHead(500, {"Content-Type": "text/plain"});
                    response.write(`Error en la lectura del fichero: ${url}`);
                }
                response.end();
            });
        } else {
            console.log('Peticion invalida: ' + url);
            response.writeHead(404, {'Content-Type': 'text/plain'});
            response.write('404 Not Found\n');
            response.end();
        }
    });

let allClients = new Array();

const io = new Server(httpServer);
io.sockets.on('connection', (client) => {
        const cAddress = client.request.socket.remoteAddress;
        const cPort = client.request.socket.remotePort;

        allClients.push({address:cAddress, port:cPort});
        console.log(`Nueva conexión de ${cAddress}:${cPort}`);

        io.sockets.emit('all-connections', allClients);

        client.on('output-evt', (data) => {
            client.emit('output-evt', 'Hola Cliente!');
        });

```

```
47          client.on('disconnect', () => {
48              console.log(`El usuario ${cAddress}:${cPort} se va a desconectar`);
49
50              const index = allClients.findIndex(cli => cli.address == cAddress && cli.port ==
        cPort);
51
52              if (index != -1) {
53                  allClients.splice(index, 1);
54                  io.sockets.emit('all-connections', allClients);
55              } else {
56                  console.log('¡No se ha encontrado al usuario!')
57              }
58              console.log(`El usuario ${cAddress}:${cPort} se ha desconectado`);
59          });
60      }
61  );
62
63  httpServer.listen(8080);
64  console.log('Servicio Socket.io iniciado');
```

connections.js

El cliente web que se muestra a continuación ("connections.html") se conecta al servicio y le envía un evento de tipo "output-evt" con el mensaje "Hola Servicio!". El cliente, además, se suscribe a los eventos "output-evt", "all-connections" y "disconnect". Cuando recibe un evento tipo "output-evt" muestra un mensaje con el contenido enviado desde el servicio. Al recibir un evento tipo "all-connections" muestra un listado con todos los usuarios conectados (su dirección IP y su puerto de conexión). Finalmente, al recibir el evento "disconnect", que se recibe cuando el servicio deja de estar disponible (por ejemplo, ha ocurrido algún error de conectividad), se muestra el mensaje "El servicio ha dejado de funcionar!!".

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="uft-8">
5           <title>Connections</title>
6       </head>
7       <body>
8           <span id="mensaje_servicio"></span>
9           <div id="lista_usuarios"></div>
10      </body>
11      <script src="/socket.io/socket.io.js"></script>
12      <script type="text/javascript">
13          function mostrar_mensaje(msg) {
14              const span_msg = document.getElementById('mensaje_servicio');
```

```
15            span_msg.textContent = msg;
16        }
17
18        function actualizarLista(usuarios) {
19            const listCont = document.getElementById('lista_usuarios');
20            while(listCont.firstChild && listCont.removeChild(listCont.firstChild));
21
22            const listElement = document.createElement('ul');
23            listCont.appendChild(listElement);
24
25            const num = usuarios.length;
26            for(var i=0; i<num; i++) {
27                const listItem = document.createElement('li');
28                listItem.textContent = usuarios[i].address + ':' + usuarios[i].port;
29                listElement.appendChild(listItem);
30            }
31        }
32
33        const serviceURL = document.URL;
34        const socket = io(serviceURL);
35        socket.on('connect', () => {
36            socket.emit('output-evt', 'Hola Servicio!');
37        });
38        socket.on('output-evt', (data) => {
39            mostrar_mensaje('Mensaje de servicio: ' + data);
40        });
41        socket.on('all-connections', (data) => {
42            actualizarLista(data);
43        });
44        socket.on('disconnect', () => {
45            mostrar_mensaje('El servicio ha dejado de funcionar!!');
46        });
47    </script>
48 </html>
```

connections.html

Se puede comprobar el funcionamiento del cliente web descrito abriendo varias ventanas de navegación y escribiendo como url la dirección IP y puerto del servicio. Cada vez que se abra una nueva ventana o se cierre podremos observar como la lista de usuarios va modificándose dinámicamente, agregándose o eliminándose usuarios de la lista, respectivamente. Además, se puede probar a terminar la ejecución del servicio. Veremos como el cliente web muestra un mensaje indicándolo. Por último, si ejecutamos el servicio de nuevo observaremos como los clientes se conectan automáticamente al mismo y actualizan apropiadamente la lista de usuarios.

4. Uso de MongoDB desde Node.js

MongoDB es una base de datos tipo NoSQL. Este tipo de bases de datos permiten guardar información no estructurada. De tal forma, cada entrada en una base de datos MongoDB podrá tener un número variable de parejas claves-valor asociados mediante colecciones (*collections*).

Podemos acceder a bases de datos MongoDB desde cualquier servicio escrito sobre la plataforma Node.js una vez instalado el módulo correspondiente, tal y como se describe en la sección 1.4.

El siguiente ejemplo muestra la implementación de un servicio que recibe dos tipos notificaciones mediante Socket.io: “poner” y “obtener”. El contenido del primer tipo de notificación es introducido por el servicio en la base de datos “baseDatosTest”, dentro de la colección de claves-valor “test”. Al recibir el segundo tipo de notificación, el servicio hace una consulta sobre la colección “test” en base al contenido de la propia notificación y le devuelve los resultados al cliente. Además, cuando un cliente se conecta, el servicio le devuelve su dirección de conexión.

```
1  import http          from 'node:http';
2  import {join}        from 'node:path';
3  import {readFile}    from 'node:fs';
4  import {Server}      from 'socket.io';
5  import {MongoClient} from 'mongodb';
6
7  const httpServer = http
8      .createServer((request, response) => {
9          let {url} = request;
10         if(url == '/') {
11             url = '/mongo-test.html';
12             const filename = join(process.cwd(), url);
13
14             readFile(filename, (err, data) => {
15                 if(!err) {
16                     response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
17                     response.write(data);
18                 } else {
19                     response.writeHead(500, {"Content-Type": "text/plain"});
20                     response.write(`Error en la lectura del fichero: ${url}`);
21                 }
22                 response.end();
23             });
24         }
25         else {
26             console.log('Peticion invalida: ' + url);
27             response.writeHead(404, {'Content-Type': 'text/plain'});
28             response.write('404 Not Found\n');
29             response.end();
```

```
30        }
31    });
32
33 MongoClient.connect("mongodb://localhost:27017/").then((db) => {
34    const dbo = db.db("baseDatosTest");
35    const collection = dbo.collection("test");
36
37    const io = new Server(httpServer);
38    io.sockets.on('connection', (client) => {
39        client.emit('my-address', {host:client.request.socket.remoteAddress, port:client.request
       .socket.remotePort});
40        client.on('poner', (data) => {
41            collection.insertOne(data, {safe:true}).then((result) => {});
42        });
43        client.on('obtener', (data) => {
44            collection.find(data).toArray().then((results) => {
45                client.emit('obtener', results);
46            });
47        });
48    });
49    httpServer.listen(8080);
50 }).catch((err) => {console.error(err);});
51
52 console.log('Servicio MongoDB iniciado');
```

mongo-test.js

El siguiente cliente web introduce en la base de datos su host, su puerto y el momento en el que se conectó al servicio. A continuación, trata de recuperar de la base de datos todo el historial de conexiones realizadas desde el host donde se ejecuta el cliente.

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="uft-8">
5          <title>MongoDB Test</title>
6      </head>
7      <body>
8          <div id="resultados"></div>
9      </body>
10     <script src="/socket.io/socket.io.js"></script>
11     <script type="text/javascript">
12         function actualizarLista(usuarios) {
13             const listCont = document.getElementById('resultados');
14             while(listCont.firstChild && listCont.removeChild(listCont.firstChild));
15
```

```
16            const listElement = document.createElement('ul');
17            listCont.appendChild(listElement);
18
19            const num = usuarios.length;
20            for(var i=0; i<num; i++) {
21                const listItem = document.createElement('li');
22                listItem.textContent = JSON.stringify(usuarios[i]);
23                listElement.appendChild(listItem);
24            }
25        }
26
27        const serviceURL = document.URL;
28        const socket = io(serviceURL);
29        socket.on('my-address', (data) => {
30            var d = new Date();
31            socket.emit('poner', {host:data.host, port:data.port, time:d});
32            socket.emit('obtener', {host:data.host});
33        });
34        socket.on('obtener', (data) => {actualizarLista(data);});
35        socket.on('disconnect', ()  => {actualizarLista({});});
36    </script>
37 </html>
```

mongo-test.html

Se puede observar como cada vez que se abre el cliente web aparece una nueva entrada en la lista de conexiones.

5. Ejercicio

Suponga 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 (véase la figura 1). Además dicho servidor incluye un agente capaz de notificar alarmas y tomar decisiones básicas. El sistema se comporta como se describe a continuación:

- Los **sensores** difunden información acerca de las medidas tomadas a través del servidor. Dichas medidas serán simuladas mediante un formulario de entrada que proporcionará el servidor para poder actulizar las mediciones de ambos sensores. La introducción en el formulario de una nueva medida en cualquiera de los sensores conllevará la publicación del correspondiente evento que incluirá dicha medida. La misma página mostrará los cambios que se produzcan en el estado de los actuadores.

- El **servidor** proporcionará el formulario/página comentado en el punto anterior y la página a la que accederá el usuario descrita en el punto siguiente. Además, mantendrá las subscriciones, de los usuarios que se encuentren accediendo al sistema y del agente, a los eventos relacionados con luminosidad y temperatura. Por último, también guardará un histórico de los eventos (cambios en las medidas) producidos en el sistema en una base de datos con la correspondiente marca de tiempo asociada a cada evento.

- Cada **usuario** accederá al estado del sistema a través de la página correspondiente facilitada por el servidor. Dicha página mostrará las nuevas medidas que generen los sensores cuando éstas se produzcan, el histórico de eventos y las notificaciones recibidas. Además, el usuario podrá abrir/cerrar la persiana, y/o encender/apagar el sistema de A/C en cualquier momento.

- El **agente** detectará cuando las medidas sobrepasan ciertos umbrales máximos y mínimos tanto de luminosidad como de temperatura, enviando en tales casos alarmas (eventos con el texto de la alarma producida) a todos usuarios conectados al sistema. Además, la detección de determinados eventos producirá cambios en el estado de los actuadores correspondientes, tal y como se muestra a continuación:
  - Si la luminosidad sobrepasa el máximo establecido, se producirá el cierre automático de la persiana.
  - Si la temperatura cae por debajo del mínimo establecido, se producirá el apagado del aire acondicionado.
  - Si la temperatura excede el máximo establecido, se encederá el aire acondicionado.

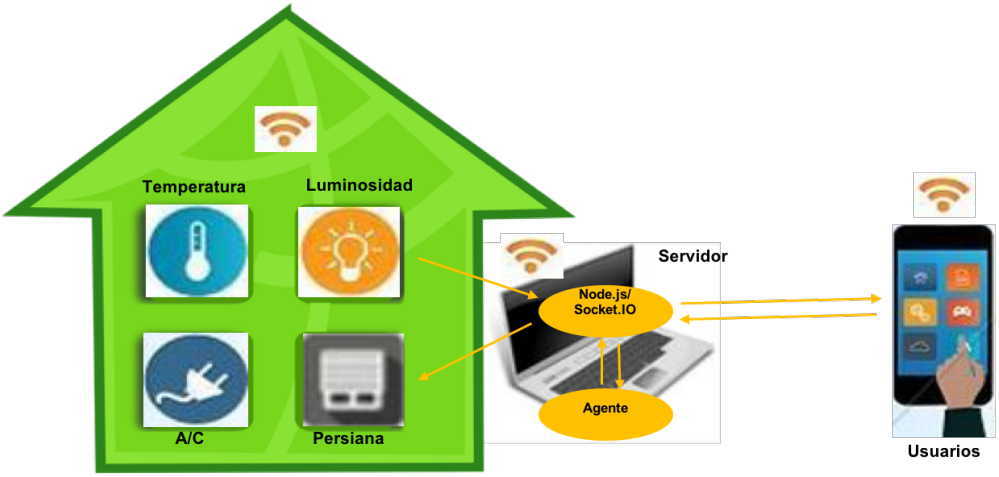

Figura 1: Sistema domótico a implementar