Índice

1. Requisitos Previos

En este capítulo se describe la instalación en el sistema operativo Ubuntu 12.04 de Node.js, Socket.io y MongoDB.

1.1. Instalación de Node.js

Para instalar Node.js desde Ubuntu se deberán ejecutar las siguientes órdenes desde el terminal:

```
> sudo apt-get update
> sudo apt-get install python-software-properties
> sudo apt-get install python g++ make
> sudo add-apt-repository ppa:chris-lea/node.js
> sudo apt-get update
> sudo apt-get install nodejs
```

Tras la instalación de Node.js se deberá comprobar que se pueden ejecutar las órdenes "nodejs" y "npm".

1.2. Instalación de Socket.io

Para instalar Socket.io se deberá ejecutar desde el terminal:

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

Si todo es correcto, en nuestro directorio personal deberemos tener una nueva carpeta llamada "node_modules" que deberá tener una subcarpeta llamada "socket.io".

1.3. Instalación de MongoDB

MongoDB se deberá instalar en dos pasos. Primero se deberá instalar el servidor de bases de datos de MongoDB:

```
> sudo apt-get install mongodb
```

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

A continuación se deberá instalar el módulo que nos permitirá desarrollar un cliente Node.js para una base de datos MondoDB.:

```
> sudo npm install mongodb
```

2. Implementación de Servicios con Node.js

Node.js (`http://nodejs.org`) 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.

Los servicios se implementan en ficheros con extensión ".js". Un ejemplo sencillo de servicio web escrito en Node.js es el siguiente:

helloworld.js

```
1  var http = require("http");
2  var httpServer = http.createServer(
3    function(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  );
10 httpServer.listen(8080);
11 console.log("Servicio HTTP iniciado");
```

Podemos probar su funcionamiento ejecutando desde el terminal:

```
> nodejs helloworld.js
```

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

En el código anterior, se comienza obteniendo un módulo de entre los disponibles en Node.js. En este caso se obtiene el módulo "http", que permite implementar servicios que sirvan contenidos usando el protocolo http. En Node.js se pueden obtener referencias a 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 una función con dos parámetros: request y response. Dicha función se llamará cada vez que se reciba una petición mediante el protocolo http (por ejemplo 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 10, 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 del código, el servicio no termina su ejecución. Ésto 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:

calculadora.js

```
1  var http = require("http");
2  var url = require("url");
3
4  function calcular(operacion, val1, val2) {
5    if (operacion=="sumar") return val1+val2;
6    else if (operacion == "restar") return val1-val2;
7    else if (operacion == "producto") return val1*val2;
8    else if (operacion == "dividir") return val1/val2;
9    else return "Error: Par&aacute;metros no v&aacute;lidos";
10 }
11
12 var httpServer = http.createServer(
13   function(request, response) {
14     var uri = url.parse(request.url).pathname;
15     var output = "";
16     while (uri.indexOf('/') == 0) uri = uri.slice(1);
17     var params = uri.split("/");
18     if (params.length >= 3) {
19       var val1 = parseFloat(params[1]);
20       var val2 = parseFloat(params[2]);
21       var result = calcular(params[0], val1, val2);
22       output = result.toString();
23     }
24     else output = "Error: El n&uacute;mero de par&aacute;metros
           no es v&aacute;lido";
25
26     response.writeHead(200, {"Content-Type": "text/html"});
27     response.write(output);
28     response.end();
29   }
30 );
31 httpServer.listen(8080);
32 console.log("Servicio HTTP iniciado");
```

Este servicio recibe peticiones REST del tipo “http://localhost:8080/sumar/2/3”. Se puede comprobar utilizando un navegador como el servicio devolverá el resultado de la

operación solicitada sobre los números aportados. En este caso los resultados devueltos tienen formato HTML. Por ello es necesario especificar que el tipo MIME de la respuesta es "text/html", tal y como se puede observar en la línea 26.

En el siguiente ejemplo se realiza una mejora del servicio anterior. Con este nuevo servicio se proporciona una web con una interfaz 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:

calculadora–web.js

```
1  var http = require("http");
2  var url = require("url");
3  var fs = require("fs");
4  var path = require("path");
5  var mimeTypes = { "html": "text/html", "jpeg": "image/jpeg", "
       jpg": "image/jpeg", "png": "image/png", "js": "text/
       javascript", "css": "text/css", "swf": "application/x-
       shockwave-flash"};
6
7  function calcular(operacion, val1, val2) {
8    if (operacion=="sumar") return val1+val2;
9    else if (operacion == "restar") return val1-val2;
10   else if (operacion == "producto") return val1*val2;
11   else if (operacion == "dividir") return val1/val2;
12   else return "Error: Par&aacute;metros no v&aacute;lidos";
13 }
14
15 var httpServer = http.createServer(
16   function(request, response) {
17     var uri = url.parse(request.url).pathname;
18     if (uri=="/") uri = "/calc.html";
19     var fname = path.join(process.cwd(), uri);
20     fs.exists(fname, function(exists) {
21       if (exists) {
22         fs.readFile(fname, function(err, data){
23           if (!err) {
24             var extension = path.extname(fname).split(".")[1];
25             var mimeType = mimeTypes[extension];
26             response.writeHead(200, mimeType);
27             response.write(data);
28             response.end();
29           }
30           else {
31             response.writeHead(200, {"Content-Type": "text/
```

```
                plain"});
32              response.write('Error de lectura en el fichero: '+
                    uri);
33              response.end();
34            }
35          });
36        }
37        else{
38          while (uri.indexOf('/') == 0) uri = uri.slice(1);
39          var params = uri.split("/");
40          if (params.length >= 3) { //REST Request
41            console.log("Peticion REST: "+uri);
42            var val1 = parseFloat(params[1]);
43            var val2 = parseFloat(params[2]);
44            var result = calcular(params[0], val1, val2);
45            response.writeHead(200, {"Content-Type": "text/html"
                  });
46            response.write(result.toString());
47            response.end();
48          }
49          else {
50            console.log("Peticion invalida: "+uri);
51            response.writeHead(200, {"Content-Type": "text/plain"
                  });
52            response.write('404 Not Found\n');
53            response.end();
54          }
55        }
56      });
57    }
58  );
59  httpServer.listen(8080);
60  console.log("Servicio HTTP iniciado");
```

Se puede observar como se hace uso de un módulo llamado “fs”. Este módulo permite realizar operaciones de entrada/salida sobre el sistema de ficheros en el servidor. En este caso se utiliza para leer los ficheros almacenados en la carpeta donde se ejecuta el 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 “path” que permite extraer información de una ruta a partir de cadenas de caractéres.

Además del servicio, a continuación se presenta un ejemplo de cliente web. Su código fuente (“calc.html”) se muestra a continuación:

calc.html

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset
       =utf-8">
    <title>Calculadora</title>
  </head>
  <body>
    <form action="javascript:void(0);" onsubmit="javascript:
       enviar();">
      Valor1: <input type="label" id="val1" /><br />
      Valor2: <input type="label" id="val2" /><br />
      Operaci&oacute;n:
      <select id="operacion">
        <option value="sumar">Sumar</option>
        <option value="restar">Restar</option>
        <option value="producto">Producto</option>
        <option value="dividir">Dividir</option>
      </select><br />
      <input type="submit" value="Calcular" />
    </form>
    <span id="resul"></span>
  </body>
  <script type="text/javascript">
    var serviceURL = document.URL;
    function enviar() {
      var val1 = document.getElementById("val1").value;
      var val2 = document.getElementById("val2").value;
      var oper = document.getElementById("operacion").value;

      var url = serviceURL+"/"+oper+"/"+val1+"/"+val2;
      var httpRequest = new XMLHttpRequest();
      httpRequest.onreadystatechange = function() {
          if (httpRequest.readyState === 4){
              var resultado = document.getElementById("resul");
              resultado.innerHTML = httpRequest.responseText;
            }
        };
      httpRequest.open("GET", url, true);
      httpRequest.send(null);
    }
  </script>
</html>
```

El cliente web muestra un formulario cuyo envío se realiza mediante una función JavaScript. Dicha función 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 (con XMLHttpRequest).

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 ha existido durante años en la web es la imposibilidad de enviar información desde un servicio hacia un cliente sin que el propio cliente la haya solicitado previamente. Por ejemplo, en un chat, hasta hace poco era inviable tecnológicamente notificar a todos los usuarios que un nuevo mensaje había sido enviado por algún usuario. Por ello, se optaba por realizar consultas periódicas desde los clientes hacia los servicios para comprobar si había o no nueva información para mostrar al usuario. Dicho sistema de comunicación (conocido como *polling* o modelo de notificaciones tipo *pull*) es altamente ineficiente e impide que los servicios pudieran atender adecuadamente a un alto número de usuarios de manera simultánea, presentando, por tanto, problemas de escalabilidad. El nuevo 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 el servicio 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 los siguientes eventos:

- **'connect'** / **'connection'** - Se emite al realizarse una conexión correctamente.
- **'reconnecting'** - Notifica un intento de reconexión.
- **'disconnect'** - Notificación de la desconexión entre cliente y servicio.
- **'connect_failed'** - Se emite cuando socket.io no es capaz de establecer una conexión.
- **'error'** - Notificación de un error que no puede ser tratado mediante cualquier otro evento por defecto.
- **'message'** - La función *emit* es la que permite en Socket.io notificar eventos. No obstante, también existe la función *send*, que permite enviar información arbitraria a nivel de sockets. Si en vez de utilizar la función "emit" para transferir información hacemos uso de la función "send", entonces se notificará este evento en el receptor para que éste pueda gestionar la información recibida.
- **'anything'** - Notificación de que se ha recibido cualquier tipo de evento definido por el usuario. Esta notificación no se recibe para los eventos que aporta Socket.io por defecto (los enumerados en esta lista).
- **'reconnect'** - Se emite cuando un cliente trata de reconectarse a un servicio.

- **'reconnect_failed'** - Error de reconexión a un servicio.
- **'reconnecting'** - Notificación de que un cliente está tratando de reconectarse a un servicio.

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 cliente suscritos cada vez que un nuevo cliente se conecta o desconecta. El envío a todos los clientes se realiza llamando a la función *emit* sobre el array 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!".

connections.js

```
1  var http = require("http");
2  var url = require("url");
3  var fs = require("fs");
4  var path = require("path");
5  var socketio = require("socket.io");
6  var mimeTypes = { "html": "text/html", "jpeg": "image/jpeg", "
       jpg": "image/jpeg", "png": "image/png", "js": "text/
       javascript", "css": "text/css", "swf": "application/x-
       shockwave-flash"};
7
8  var httpServer = http.createServer(
9    function(request, response) {
10     var uri = url.parse(request.url).pathname;
11     if (uri=="/") uri = "/connections.html";
12     var fname = path.join(process.cwd(), uri);
13     fs.exists(fname, function(exists) {
14       if (exists) {
15         fs.readFile(fname, function(err, data){
16           if (!err) {
17             var extension = path.extname(fname).split(".")[1];
18             var mimeType = mimeTypes[extension];
19             response.writeHead(200, mimeType);
20             response.write(data);
21             response.end();
22           }
23           else {
24             response.writeHead(200, {"Content-Type": "text/
                 plain"});
25             response.write('Error de lectura en el fichero: '+
                 uri);
```

```
26              response.end();
27            }
28          });
29        }
30        else{
31          console.log("Peticion invalida: "+uri);
32          response.writeHead(200, {"Content-Type": "text/plain"})
              ;
33          response.write('404 Not Found\n');
34          response.end();
35        }
36      });
37    }
38  );
39  httpServer.listen(8080);
40  var io = socketio.listen(httpServer);
41
42  var allClients = new Array();
43  io.sockets.on('connection',
44    function(client) {
45      allClients.push({address:client.request.connection.
          remoteAddress, port:client.request.connection.remotePort
          });
46      console.log('New connection from ' + client.request.
          connection.remoteAddress + ':' + client.request.
          connection.remotePort);
47      io.sockets.emit('all-connections', allClients);
48      client.on('output-evt', function (data) {
49        client.emit('output-evt', 'Hola Cliente!');
50      });
51      client.on('disconnect', function() {
52        var index = allClients.indexOf(client.request.connection.
            remoteAddress);
53        if (index != -1) {
54          allClients.splice(index, 1);
55          io.sockets.emit('all-connections', allClients);
56        }
57        console.log('El usuario '+client.request.connection.
            remoteAddress+' se ha desconectado');
58      });
59    }
60  );
61
62  console.log("Servicio Socket.io iniciado");
```

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!!".

connections.html

```
1  <html>
2    <head>
3      <meta http-equiv="Content-Type" content="text/html; charset
           =utf-8">
4      <title>Connections</title>
5    </head>
6    <body>
7      <span id="mensaje_servicio"></span>
8      <div id="lista_usuarios"></div>
9    </body>
10   <script src="/socket.io/socket.io.js"></script>
11   <script type="text/javascript">
12     function mostrar_mensaje(msg){
13       var span_msg = document.getElementById('mensaje_servicio'
             );
14       span_msg.innerHTML = msg;
15     }
16
17     function actualizarLista(usuarios){
18       var listContainer = document.getElementById('
             lista_usuarios');
19       listContainer.innerHTML = '';
20       var listElement = document.createElement('ul');
21       listContainer.appendChild(listElement);
22       var num = usuarios.length;
23       for(var i=0; i<num; i++) {
24               var listItem = document.createElement('li');
25               listItem.innerHTML = usuarios[i].address+":"+
                     usuarios[i].port;
26               listElement.appendChild(listItem);
```

```
27              }
28          }
29
30      var serviceURL = document.URL;
31      var socket = io.connect(serviceURL);
32      socket.on('connect', function(){
33          socket.emit('output-evt', 'Hola Servicio!');
34      });
35      socket.on('output-evt', function(data) {
36          mostrar_mensaje('Mensaje de servicio: '+data);
37      });
38      socket.on('all-connections', function(data) {
39          actualizarLista(data);
40      });
41      socket.on('disconnect', function() {
42          mostrar_mensaje('El servicio ha dejado de funcionar!!');
43      });
44  </script>
45 </html>
```

Se puede comprobar el funcionamiento de 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 1.3.

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 "mibd", dentro de la colección de claves-valor "test". Al recibir el segundo tipo de notificación, el servicio hace una consulta sobre la base de datos "mibd" 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.

mongo–test.js

```
1  var http = require("http");
2  var url = require("url");
3  var fs = require("fs");
4  var path = require("path");
5  var socketio = require("socket.io");
6  var MongoClient = require('mongodb').MongoClient;
7  var MongoServer = require('mongodb').Server;
8  var mimeTypes = { "html": "text/html", "jpeg": "image/jpeg", "
       jpg": "image/jpeg", "png": "image/png", "js": "text/
       javascript", "css": "text/css", "swf": "application/x-
       shockwave-flash"};
9
10 var httpServer = http.createServer(
11   function(request, response) {
12     var uri = url.parse(request.url).pathname;
13     if (uri=="/") uri = "/mongo-test.html";
14     var fname = path.join(process.cwd(), uri);
15     fs.exists(fname, function(exists) {
16       if (exists) {
17         fs.readFile(fname, function(err, data){
18           if (!err) {
19             var extension = path.extname(fname).split(".")[1];
20             var mimeType = mimeTypes[extension];
21             response.writeHead(200, mimeType);
22             response.write(data);
```

```
23              response.end();
24            }
25            else {
26              response.writeHead(200, {"Content-Type": "text/
                   plain"});
27              response.write('Error de lectura en el fichero: '+
                   uri);
28              response.end();
29            }
30          });
31        }
32        else{
33          console.log("Peticion invalida: "+uri);
34          response.writeHead(200, {"Content-Type": "text/plain"})
                ;
35          response.write('404 Not Found\n');
36          response.end();
37        }
38      });
39    }
40  );
41
42  var mongoClient = new MongoClient(new MongoServer('localhost',
       27017));
43  mongoClient.connect("mongodb://localhost:27017/mibd", function(
       err, db) {
44    httpServer.listen(8080);
45    var io = socketio.listen(httpServer);
46
47    db.createCollection("test", function(err, collection){
48        io.sockets.on('connection',
49      function(client) {
50        client.emit('my-address', {host:client.request.connection
              .remoteAddress, port:client.request.connection.
              remotePort});
51        client.on('poner', function (data) {
52          collection.insert(data, {safe:true}, function(err,
                result) {});
53        });
54        client.on('obtener', function (data) {
55          collection.find(data).toArray(function(err, results){
56            client.emit('obtener', results);
57          });
58        });
```

```
59      });
60      });
61  });
62
63  console.log("Servicio MongoDB iniciado");
```

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.

mongo–test.html

```
1  <html>
2    <head>
3      <meta http-equiv="Content-Type" content="text/html; charset
          =utf-8">
4      <title>MongoDB Test</title>
5    </head>
6    <body>
7      <div id="resultados"></div>
8    </body>
9    <script src="/socket.io/socket.io.js"></script>
10   <script type="text/javascript">
11     function actualizarLista(usuarios){
12       var listContainer = document.getElementById('resultados')
            ;
13       listContainer.innerHTML = '';
14       var listElement = document.createElement('ul');
15       listContainer.appendChild(listElement);
16       var num = usuarios.length;
17       for(var i=0; i<num; i++) {
18               var listItem = document.createElement('li');
19               listItem.innerHTML = JSON.stringify(usuarios[i]);
20               listElement.appendChild(listItem);
21           }
22       }
23
24     var serviceURL = document.URL;
25     var socket = io.connect(serviceURL);
26
27     socket.on('my-address', function(data) {
28       var d = new Date();
29       socket.emit('poner', {host:data.host, port:data.port,
            time:d});
```

```
30          socket.emit('obtener', {host: data.address});
31        });
32        socket.on('obtener', function(data) {
33          actualizarLista(data);
34        });
35        socket.on('disconnect', function() {
36          actualizarLista({});
37        });
38    </script>
39  </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 y proporcionadas mediante un formulario de entrada que proporcionará el servidor para poder incluir las medidas 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 tal como se comenta 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. Y por último, 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 del servidor mostrando la información en la correspondiente página que éste enviará. Dicha página mostrará las nuevas medidas que generen los sensores cuando éstas se produzcan. 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, en tal caso enviando alarmas (eventos con el texto de la alarma producida) a todos usuarios conectados al sistema. Además cuando tanto la luminosidad como la temperatura sobrepasen los valores umbrales máximos automáticamente producirá el cierre de la persiana.

El ejercicio a desarrollar con Node.js y Socket.IO consiste en el diseño e implementación de dicho sistema domótica básico.

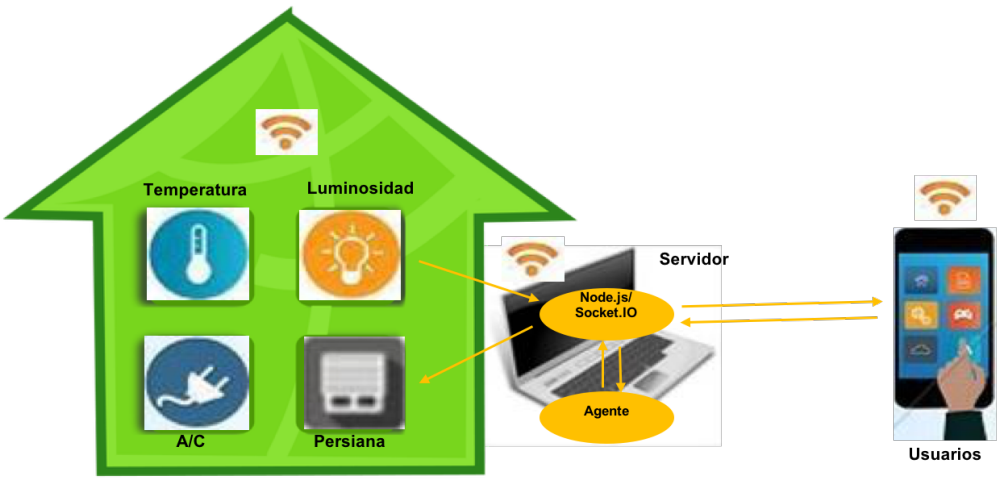

Figura 1: Sistema domótico a implementar.