



Sockets en la Web

Servidores Web de Altas Prestaciones

3º Grado en Ingeniería Informática 15/16

1- ¿Qué son los websockets?

Un *websocket* es una tecnología que nos permite abrir una comunicación bidireccional, full-duplex, con baja latencia, y basada en el protocolo TCP.

La primera aparición de un borrador de trabajo sobre esta tecnología fue en 2009 y, es una tecnología con grandes expectativas, siendo calificada como "el TCP de la Web".

Se encuentran estandarizados por la W3C mediante el WebSocket API, desde donde se pueden enviar mensajes a un servidor y recibir respuestas controladas por eventos sin tener que consultar al servidor para una respuesta, es decir, hace posible abrir una sesión de comunicación interactiva entre el navegador del usuario y un servidor.

Los datos se transfieren inmediatamente a través de una conexión full-duplex, permitiendo que ambos extremos reciban y envíen mensajes en tiempo real y de forma simultánea.

Ésta es una solución ideal para juegos en tiempo real, notificaciones instantáneas de redes sociales, presentación actualizada de cotizaciones o información meteorológica. En resumen, para aplicaciones que requieren una transferencia de datos segura y rápida.

Para establecer una conexión WebSocket se intercambia información en un protocolo de enlace de conexión específico basado en HTTP, entre el cliente y el servidor. Si se realiza correctamente la conexión el protocolo de nivel de aplicación se "actualiza" de HTTP a WebSockets, usando la conexión TCP establecida anteriormente (quedando HTTP fuera de juego).

Algo importante que debemos tener en cuenta es que un cliente no puede usar WebSockets para transferir datos a menos que el servidor también use el protocolo WebSocket. Si el servidor no admite WebSockets, se debe usar otro método de transferencia de datos.

2- Problema baja latencia cliente-servidor-cliente.

Si disponemos de un socket abierto el servidor puede enviar datos a todos los clientes conectados a ese socket, sin tener que estar constantemente procesando peticiones de Ajax (método candidato en caso de no poder usar este método innovador).

Es ésto lo que hace evidente, por tanto, la ventaja en cuanto a rendimiento, escalabilidad y latencia de las comunicaciones con el uso de WebSocket.

Como el socket está siempre abierto y escuchando, los datos son enviados inmediatamente desde el servidor al navegador, reduciendo el tiempo al mínimo en comparación con un paradigma basado en Ajax, donde hay que realizar una petición, procesar la respuesta y enviarla de nuevo de vuelta.

Para terminar es de destacar que los datos a transmitir se reducen también de manera drástica, pasando de un mínimo de 200-300 bytes en peticiones Ajax, a 10-20 bytes utilizando websockets.

3-¿Y esto... para qué sirve? ¿Alternativas?

Tal y como hemos comentado, sobre todo para aplicaciones que requieren constantes actualizaciones en el frontal debido a la interacción de terceras personas (o sistemas), las denominadas “*aplicaciones en tiempo real*”.

Antes de la aparición de WebSockets se utilizaban dos técnicas para implementar este comportamiento:

AJAX polling:

Consistente en realizar constantemente peticiones al servidor preguntándole si se ha producido algún evento que requiera una actualización en la vista.

Por ejemplo: en un chat, desde la pantalla donde llegan los posibles mensajes que recibe el usuario por parte de otros usuarios se estarían enviando peticiones HTTP al servidor (cada 2 segundos, 5, 10, o lo que sea) preguntándole ¿tengo mensajes?, ¿tengo mensajes?, ¿tengo mensajes?, etc...

Esta es una solución bastante ineficiente debido a que se genera una gran cantidad de tráfico con el servidor, sobre todo en aplicaciones con un elevado número de usuarios e intervalo entre peticiones pequeño.

Además, la mayoría de las veces el servidor responderá lo mismo: “no, no hay cambios”.

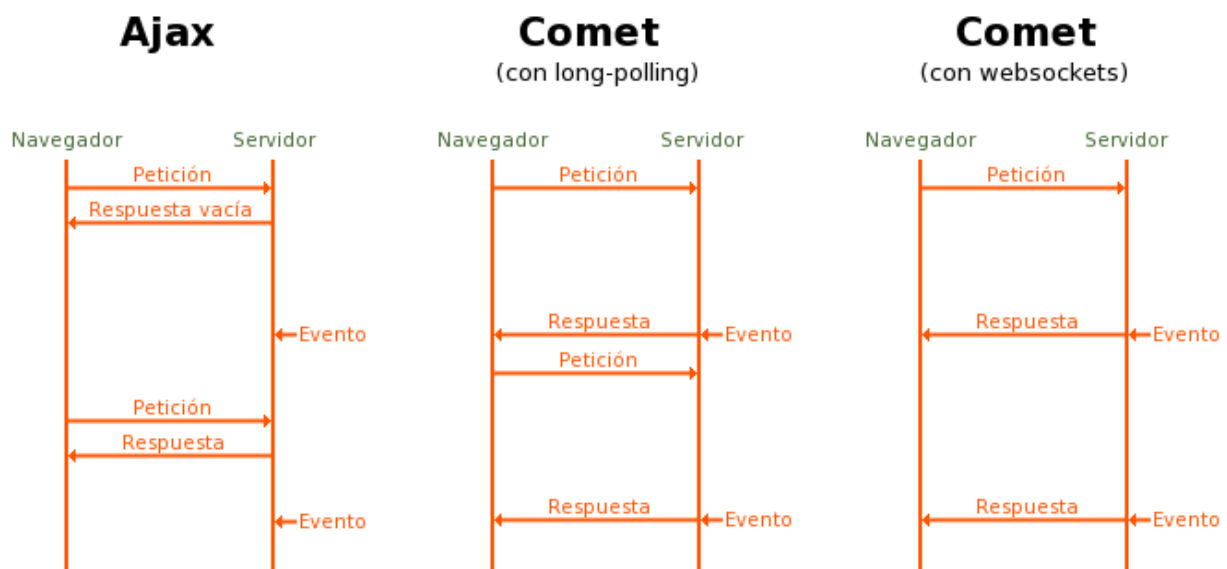
Comet o “long polling”:

Es una técnica muy parecida a la que se utiliza con los WebSockets pero con una implementación mucho más compleja.

Consiste en realizar una única petición al servidor de forma que éste responde diciendo que va a devolver la respuesta en “trozos” (streaming).

La petición queda abierta hasta que el servidor responda con todas las porciones de respuesta que solicita el cliente, que no son otra cosa que eventos que se producen en el servidor notificando un cambio de estado en el cliente.

Ajax vs. Comet



4-Uso actual de websockets

Hoy en día existe la necesidad de crear aplicaciones en tiempo real, aplicaciones que permiten que el cliente pueda tener rápidamente datos para que pueda tomar decisiones.

Ejemplos de este tipo de aplicaciones: Juegos multijugador, aplicaciones de monitorización, chats, herramientas de trabajo colaborativo, etc...

⑦ En el caso de una herramienta de *trabajo colaborativo*, cuando un equipo de trabajo está delante de la pantalla (cada uno en su ordenador) y un miembro finaliza una tarea y actualiza su estado a “finalizada”, la aplicación notifica inmediatamente al resto de usuarios (o a uno, o a varios) de que esa tarea está cerrada e inmediatamente ven un cambio en el estado de esa tarea.

⑦ En un *juego multijugador* podríamos visualizar el movimiento de otro jugador.

⑦ En un *chat* veríamos como nos llega un mensaje de otro usuario en el momento en que nos lo envía.

⑦ En una herramienta de monitorización, por ejemplo, podríamos conocer la temperatura que marca el sensor de algún componente de un sistema, etc...

La idea principal es que tenemos la información desde el mismo instante en que se genera.

5- Ámbito de aplicación y lenguajes que soporta

Se pueden utilizar en prácticamente cualquier plataforma y podemos encontrar numerosas implementaciones con su correspondiente API.

- Node.js
 - [Socket.IO](#)
 - [WebSocket-Node](#)
 - [ws](#)
- Java
 - [Jetty](#)
- Ruby
 - [EventMachine](#)
- Python
 - [pywebsocket](#)
 - [Tornado](#)
- Erlang
 - [Shirasu](#)
- C++
 - [libwebsockets](#)
- .NET
 - [SuperWebSocket](#)

Para el desarrollo y el uso de esta tecnología es necesario, además del navegador web en la parte del cliente, un lado servidor que soporte el protocolo (ws).

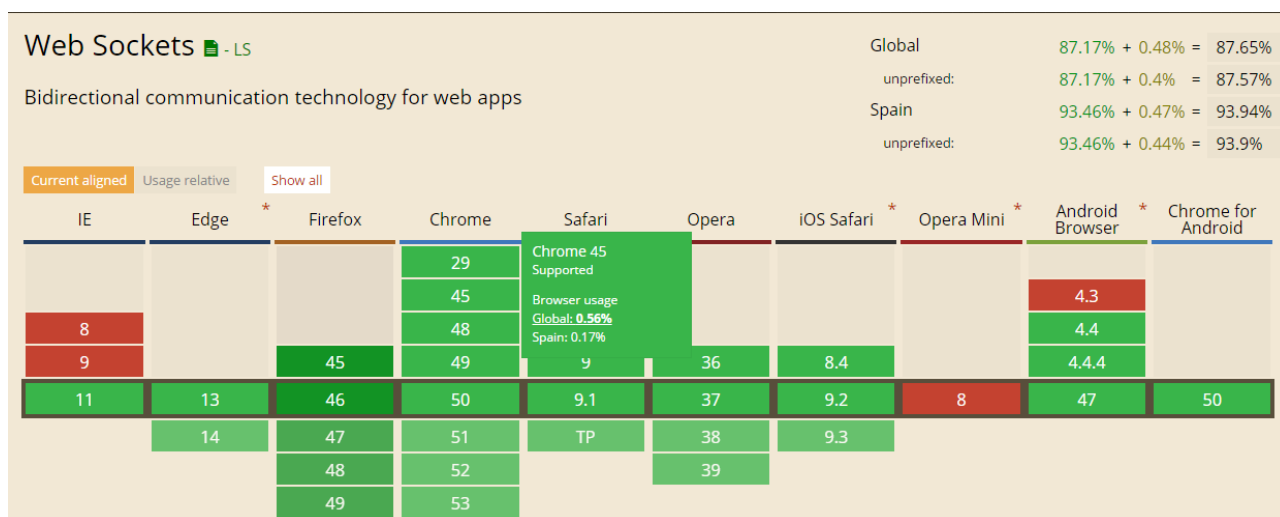
Gran parte de los lenguajes más utilizados en el año 2015 disponen de API para WebSocket: C (Libwebsockets), Java (javax.websocket), Objective-C (SocketRocket), PHP (Elephant.IO) y, obviamente, JavaScript, entre otros.

Además, también existen soluciones multiplataforma, como Socket.IO para NodeJS.

6- Comparativa de compatibilidades Y Activación en navegadores

En cuanto a la compatibilidad con el cliente hemos de decir que prácticamente todos los navegadores soportan la tecnología WebSockets. Para su uso basta con activarlo en la configuración del navegador.

Podemos comprobar si la versión de nuestro navegador es aceptada directamente a través de la web <http://caniuse.com/#feat=websockets>, donde podemos observar una tabla con los diferentes navegadores y las versiones que soportan esta tecnología.



De cualquier modo, hay navegadores en los que hay que activarlos manualmente, como es el caso de Mozilla, Firefox y Opera.

→ Como activar WebSockets:

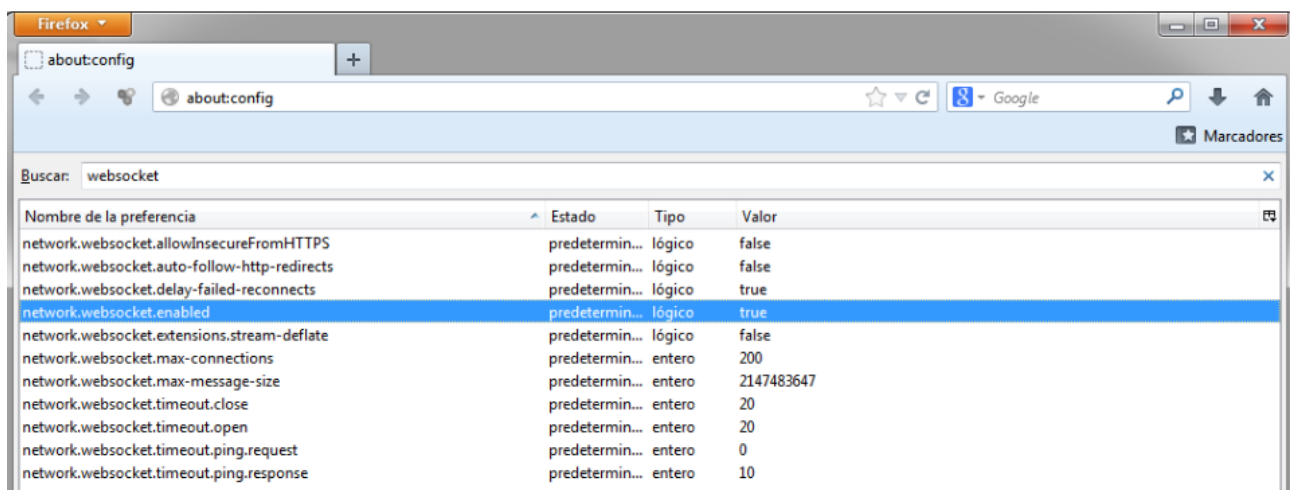
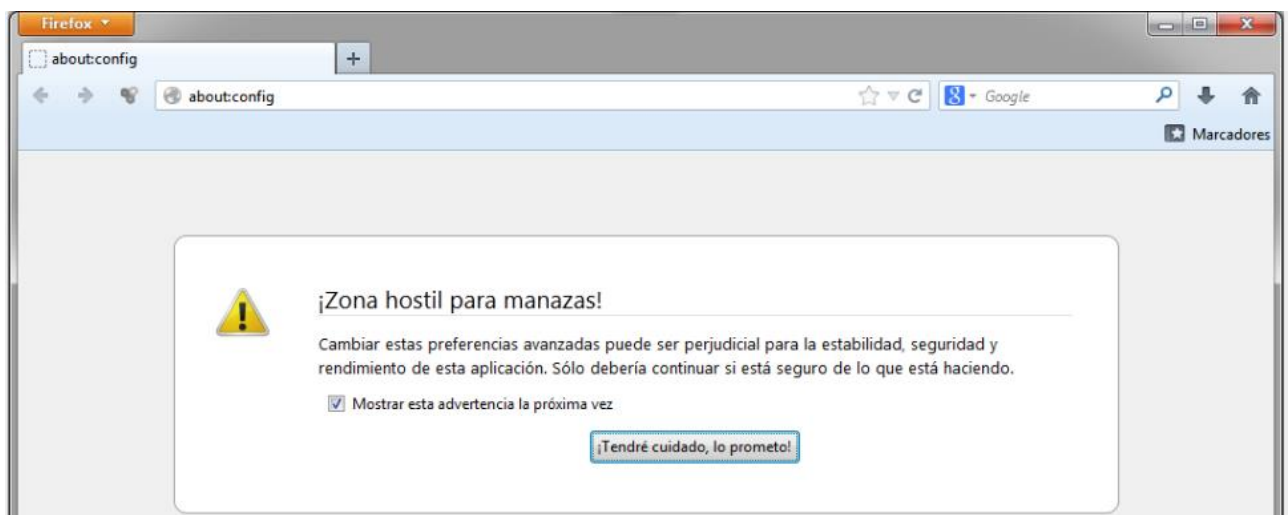
1. Iniciamos el navegador y escribimos en la barra de título : 'about:config'.

Solo en Firefox deberemos confirmar el mensaje de seguridad.

2. Introducir 'websocket' en el filtro.

3. En Firefox, cambiar el estado a verdadero del campo "network.websocket.enabled".

En Opera, Activar Websockets y guardar.



7- Caso Real

Después de ver el funcionamiento de WebSocket y todo lo que puede aportar como tecnología de comunicaciones a la integración de aplicaciones vamos a destacar un caso muy utilizado entre la mayoría de usuarios que utiliza este protocolo: [WhatsApp Web](#).

El servicio de mensajería WhatsApp es un servicio distribuido. Su cliente web utiliza WebSocket para comunicarse con los servidores, que se encargan de almacenar información y transmitir a los clientes web la información que solicitan indirectamente a los dispositivos móviles.

Utilizando las herramientas para desarrolladores que proporciona Google Chrome es posible analizar las comunicaciones entre un cliente WhatsApp Web y los servidores del servicio.

En la siguiente captura de pantalla se observa la negociación de apertura que inicia el cliente y la respuesta que recibe del servidor:

×
Headers
Frames
Timing

▼ General

Request URL: wss://w7.web.whatsapp.com/ws
Request Method: GET
Status Code: ● 101 Switching Protocols

▼ Response Headers view source

Connection: Upgrade
Sec-WebSocket-Accept: WTtQi4AJou+AGPxh8HlXum3hp1E=
Upgrade: websocket

▼ Request Headers view source

Accept-Encoding: gzip, deflate, sdch
Accept-Language: ca,es;q=0.8,en;q=0.6
Cache-Control: no-cache
Connection: Upgrade
Host: w7.web.whatsapp.com
Origin: https://web.whatsapp.com
Pragma: no-cache
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
Sec-WebSocket-Key: dLLzW+0EfplZtoB1MF4Vtw==
Sec-WebSocket-Version: 13
Upgrade: websocket
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.80 Safari/537.36

Además, también es posible comprobar los mensajes enviados y recibidos, y de qué tipo son:

×	Headers	Frames	Timing
Data	Length	Time	▲
Binary Frame (Opcode 2, mask)	90	16:34:53.321	
Binary Frame (Opcode 2, mask)	74	16:34:53.388	
s3,["Stream","update",false,"0.1.4855"]	39	16:34:53.388	
s4,["Props",{"MESSAGE_INFO":true,"media":16,"maxSubject":25,"m...	122	16:34:53.390	
Binary Frame (Opcode 2)	9166	16:34:53.397	
892.--2,	8	16:34:54.248	
892.--3,	8	16:34:54.251	
892.--4,	8	16:34:54.252	
892.--5,	8	16:34:54.253	
Binary Frame (Opcode 2)	21670	16:34:54.254	
892.--6,["query","ProfilePicThumb","34678838946@c.us"]	55	16:34:54.971	
892.--7,["query","ProfilePicThumb","34600260257@c.us"]	55	16:34:54.975	
892.--8,["query","ProfilePicThumb","34610001206@c.us"]	55	16:34:54.977	
1 s4,["Props",{"MESSAGE_INFO":true,"media":16,"maxSubject":25,"maxPartic...			

8- Como crear un chat usando Websocket

Chat usando PHP

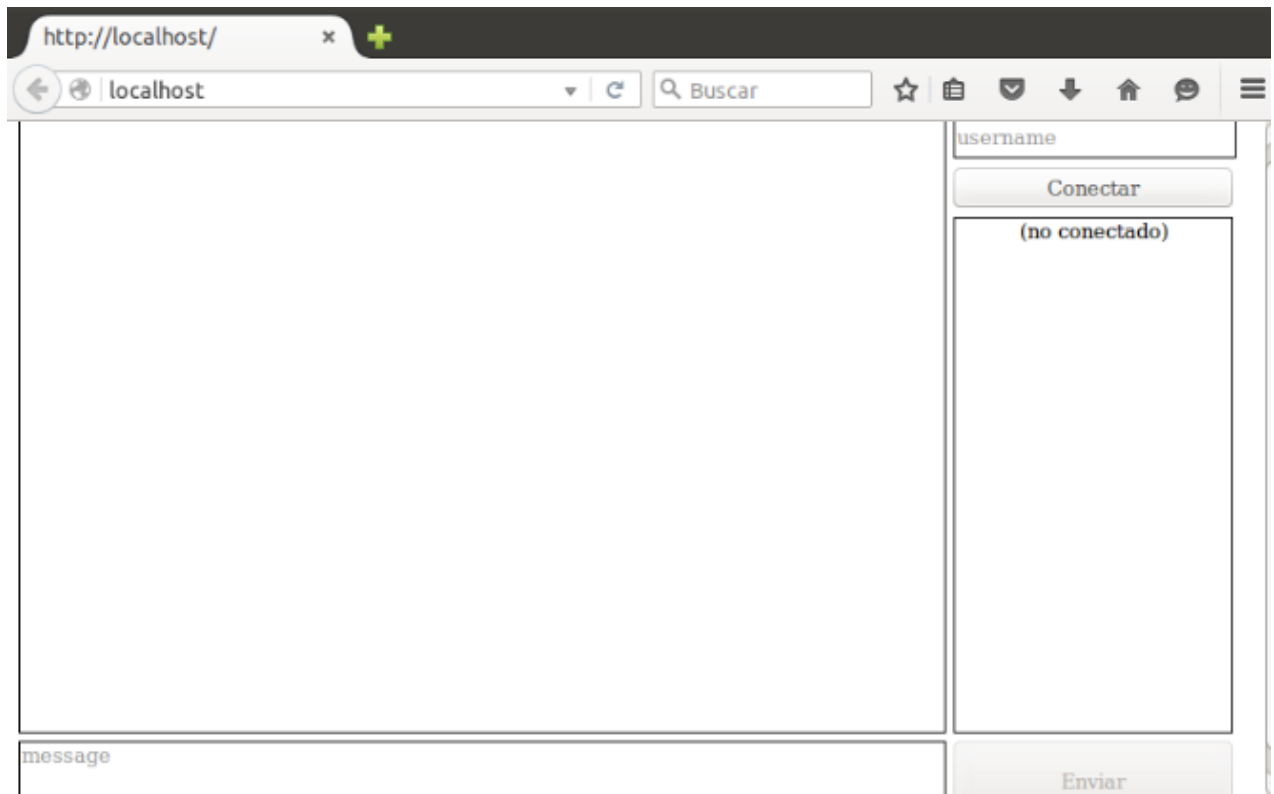
Vamos a usar una API de PHP para crear un chat.

El ejemplo está compuesto un html y css sencillo que será la interfaz del chat, y un archivo javascript.

Por otro lado tenemos el código de nuestro servicio (demonio) que es el que recibe las peticiones del cliente y se comunica directamente con la api del websockets.

```
index.html
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4   <meta charset="UTF-8" />
5   <link rel="stylesheet" type="text/css" href="main.css" />
6   <script type="text/javascript" src="main.js"></script>
7 </head>
8
9 <body>
10   <div class="box1">
11     <textarea id="chatLines" readonly="readonly"></textarea><br />
12     <textarea id="chatInput" placeholder="message"></textarea>
13   </div>
14
15   <div class="box2">
16     <input id="usernameInput" type="text" placeholder="username" /><br />
17     <input id="connectButton" type="button" value="Conectar" onclick="cb.onClickConnect()" /><br />
18     <ul id="usersList"><li style="text-align:center;">(no conectado)</li></ul>
19     <input id="sendButton" type="button" value="Enviar" onclick="cb.onClickSend()" />
20   </div>
21
22 </body>
23
24 </html>
25
```

```
index.html x main.css x
1 * {
2   font-family: Verdana;
3   font-size: 13px;
4 }
5
6 .box1 {
7   float: left;
8 }
9 .box2 {
10  float: left;
11  margin-left: 5px;
12 }
13
14 #chatLines {
15   width: 600px;
16   height: 400px;
17   margin: 0px;
18   margin-bottom: 5px;
19   border: solid 1px #000000;
20   resize: none;
21 }
22 #chatInput {
23   width: 600px;
24   height: 50px;
25   margin: 0px;
26   border: solid 1px #000000;
27   resize: vertical;
28 }
29
30 #usernameInput {
31   width: 180px;
32   height: 24px;
33   margin-bottom: 5px;
34   border: solid 1px #000000;
35 }
36 #connectButton {
37   width: 184px;
38   height: 28px;
39   margin-left: -1px;
40   margin-bottom: 5px;
41 }
42 #usersList {
```



CONEXIÓN:

Cuando un usuario se conecta se genera un evento en el cual se valida si el usuario está conectado o no.

Se recoge el valor del campo *username* en una variable y lo validamos.

Activamos los elementos inicialmente desactivados *setelementsdisabled* (como el botón enviar)

Y se realiza la conexión con el servidor.

Mediante la función *connect* creamos el objeto WS y se habilitan los eventos de la API.

El primer evento es el que ha realizado la conexión correctamente

El servidor genera un evento *onopen* (abre la conexión WS) y ponemos la variable *conected* a *true*.

Se envía mediante la api un mensaje (cabecera=CONECTANDOSE) y el nombre de usuario

La API lo transforma en una función con 4 parámetros *wsonmessage*

Y el servidor valora el mensaje

Y añade el usuario a la lista de usuarios

Y el servidor lo propaga al resto de usuarios.

```

onClickConnect: function() {
    if (chat.isConnected()) {
        chat.disconnect();
        return;
    }

    var Username = document.getElementById('usernameInput').value;

    // validamos el usuario
    if (Username == '') {
        chat.log('Debes poner un nombre de usuario.');
```

```

    }
    else if (Username.indexOf(' ') != -1) {
        chat.log('El nombre de usuario no puede contener espacios.');
```

```

    }
    else if (Username.length > chat.usernameMaxLength) {
        chat.log('El nombre de usuario no puede tener mas de '+chat.usernameMaxLength+' caracteres.');
```

ENVIO DE MENSAJES:

Para el envío de mensajes se genera un evento en el cliente *ONCLICKSEND*

Donde comprobamos la conexión y verificamos que existe un texto.

Mediante el método “*send*” que es el que va a recibir la API, enviamos al servidor dicho texto con una cabecera “TEXT”.

La API transforma los datos en una función con 4 parametros (id usuario, mensaje, tamaño, tipo) *wsOnMessage* que recibe el servidor.

Hace las comprobaciones, y mediante la función *sendchat* (server) se llama a *wsSend* (api) y divulga el mensaje a todos los usuarios conectados.

```
//funcion cuando un usuario envia un mensaje
onClickSend: function() {

    var ChatInputElement = document.getElementById('chatInput');
    var Text = ChatInputElement.value;

    // comprobamos conexion y el texto
    if (!chat.isConnected()) {
        chat.log('No conectado.');
```

```
    }
    else if (Text == '') {
        chat.log('Debes escribir un texto.');
```

```
    }
    else {
        // se envia el tenxto al servidor con la cabecera TEXT
        chat.socket.send('TEXT '+Text);

        // limpiar el campo chatinput
        ChatInputElement.value = '';
```

```
    }
},
```

```
function wsOnMessage($clientID, $message, $messageLength, $binary) {
    // comprueba la longitud
    if ($messageLength == 0) {
        wsClose($clientID); // llama a api para cerrar la comunicacion
        return;
    }

    // si no esta vacio crea un array para capturar las cabeceras
    $message = explode(' ', $message);
    $command = array_shift($message);

    // comprueba la cabecera
    if ($command == 'TEXT') {
        // un usuario ha enviado un texto al servidor, comprueba el usuario sea valido

        if (!isUser($clientID)) {
            wsClose($clientID);
            return;
        }

        // coloca el mensaje en la cadena string (implode junta los elementos del string)
        $text = implode(' ', $message);

        if ($text == '') {
            // si esta vacio
            wsSend($clientID, 'SERVER Mensaje vacio.');
```

```
            return;
        }
    }

    //recojo en nombre de usuario y mediante la funcion sendchat (api) envio a todos
    $username = getUsername($clientID);
    sendChat($username, $text);
}
```

DESCONEXION:

Si pulsamos el botón para desconectarnos se genera el evento *disconnect* y enviamos al servidor mediante “*send*” la cabecera QUIT, y mediante “*close*” indicamos que cerramos el canal para ese usuario.

El servidor recibe de la API la cabecera, comprueba que sea un usuario valido, lo elimina de la lista usuarios.

Y por defecto si no hay una cabecera o un comando valido, cierra la sesión.

```
//desconectar
disconnect: function() {
  chat.socket.send('QUIT');
  chat.socket.close();
},
```

```

}
elseif ($command == 'QUIT') {
  // un usuario abandona el chat

  if (!isUser($clientID)) { // comprueba por seguridad que no se pueda enviar quit sin ser
    usuario validado
    wsClose($clientID);
    return;
  }

  // esta en la lista y hace quit elimina usuario
  removeUser($clientID);
}

```

```

function removeUser($clientID) {
  global $users;

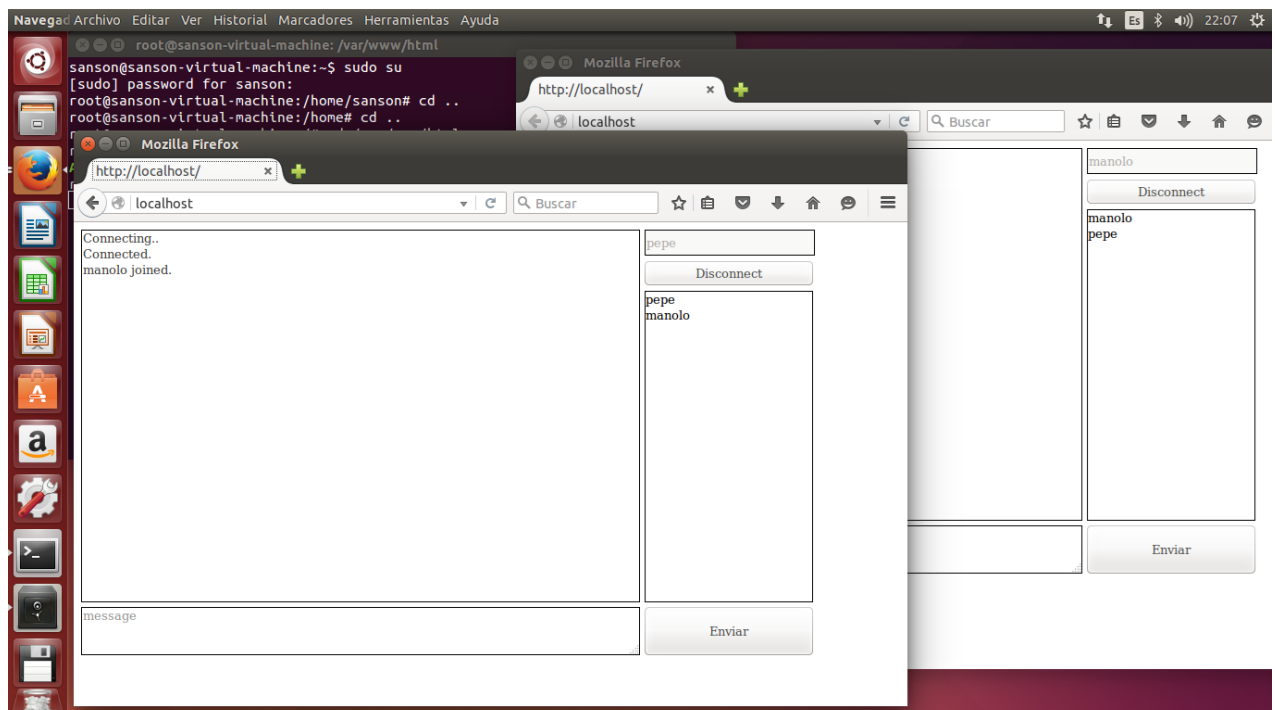
  // recoge el nombre usuario por su id
  $username = getUsername($clientID);

  // lo borramos de la lista de usuarios users
  unset($users[$clientID]);

  // envia a todos que ese usuario ha sido desconectado
  foreach ($users as $clientID2 => $username2) {
    wsSend($clientID2, 'ONQUIT '.$username);
  }
}

```

RESULTADO: Un chat interactivo

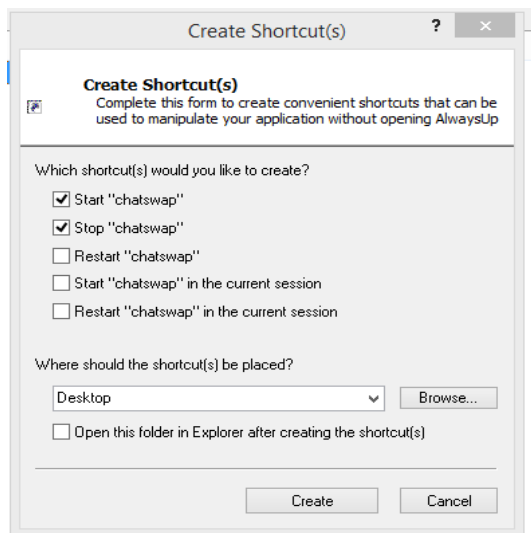
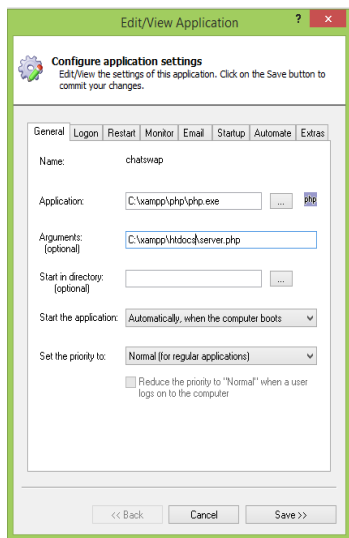


WEBSOCKETS como Soporte Técnico con

Podemos pensando en el mundo laboral, usar este tipo de chat como soporte técnico, pero es difícil encontrar empresas que trabajen con Linux, y mucho menos que un usuario, o empresario sepa defenderse bajo comandos.

Hoy en día, los usuarios buscan la simplicidad, por lo cual, investigando un poco he encontrado una aplicación que genera el arranque y la parada del servicio para que el servidor este activo continuamente, como si invocáramos un demonio Linux.

Además se generan unos botones de “encendido” y “apagado”, que facilitara al usuario su utilización.



Chat usando Node.js

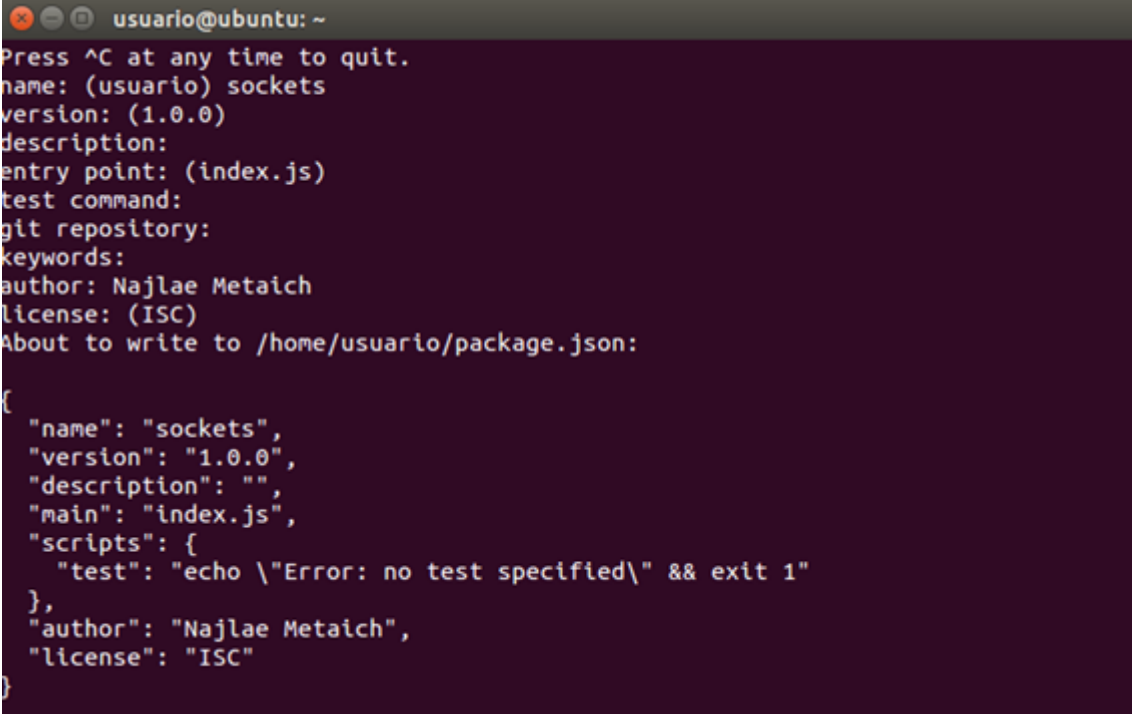
Una de las mejores formas de ver cómo funciona un websocket es desarrollando un chat. Esto, como dijimos anteriormente, puede hacerse en diversos lenguajes. Uno de los que vamos a usar será Node.js.

Lo que necesitamos para ponerlo en marcha es un servidor de websockets, cosa que crearemos con la librería Socket.io.

Lo primero que haremos será instalar node.js y npm con los siguientes comandos:

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

Hacemos un `sudo npm init` – y generamos un `package.json` de la siguiente forma:



```
usuario@ubuntu: ~
Press ^C at any time to quit.
name: (usuario) sockets
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author: Najlae Metaich
license: (ISC)
About to write to /home/usuario/package.json:
{
  "name": "sockets",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Najlae Metaich",
  "license": "ISC"
}
```

A continuación creamos un fichero `server/main.js` que va a ser nuestro servidor web. Instalaremos para ello las librerías `express` y `socket.io` mencionada anteriormente vía `npm`:

```
sudo npm install --save express
sudo npm install --save socket.io
```

Ahí creamos una aplicación con `express`, que pasaremos a un servidor `http` y se ligará al servidor que creamos con `socket.io`:

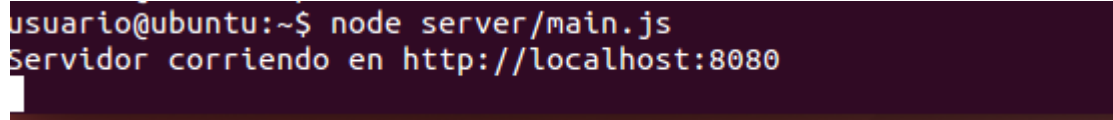
```
var express = require('express');
```

```
var app = express();  
var server = require('http').Server(app);  
var io = require('socket.io')(server);
```

Pondremos el servidor a escuchar en localhost con el puerto 8080:

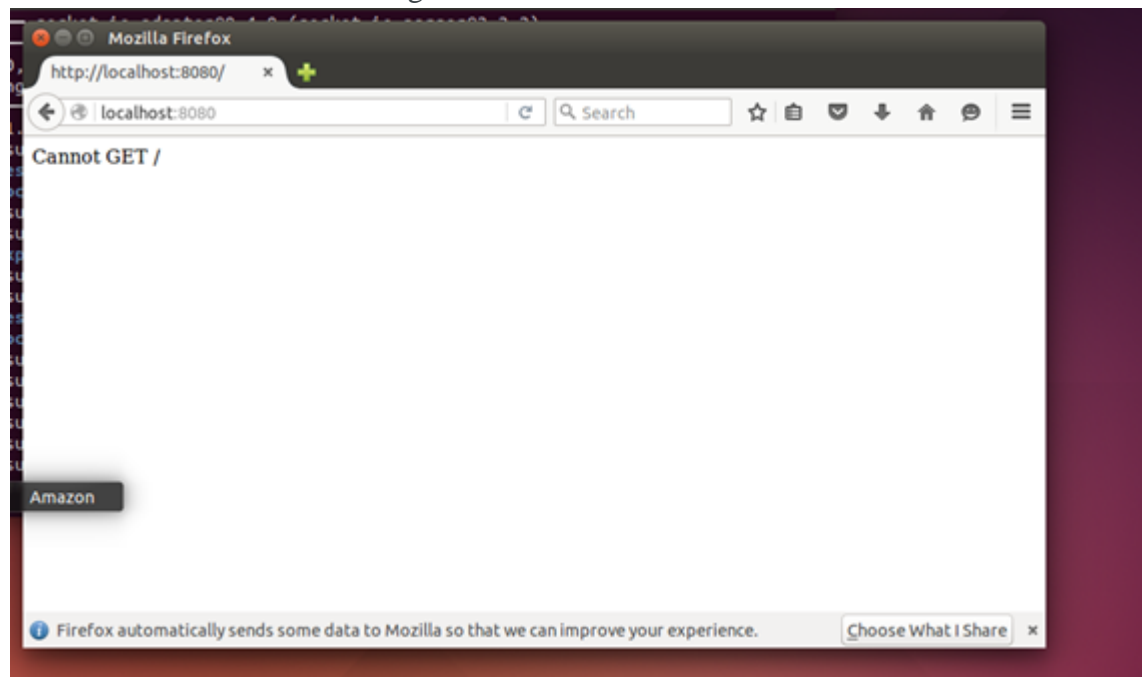
```
server.listen(8080, function() {  
  console.log('Servidor corriendo en http://localhost:8080');  
});
```

Guardamos el documento y en terminal escribimos `node server/main.js` que aparecerá de la siguiente forma:

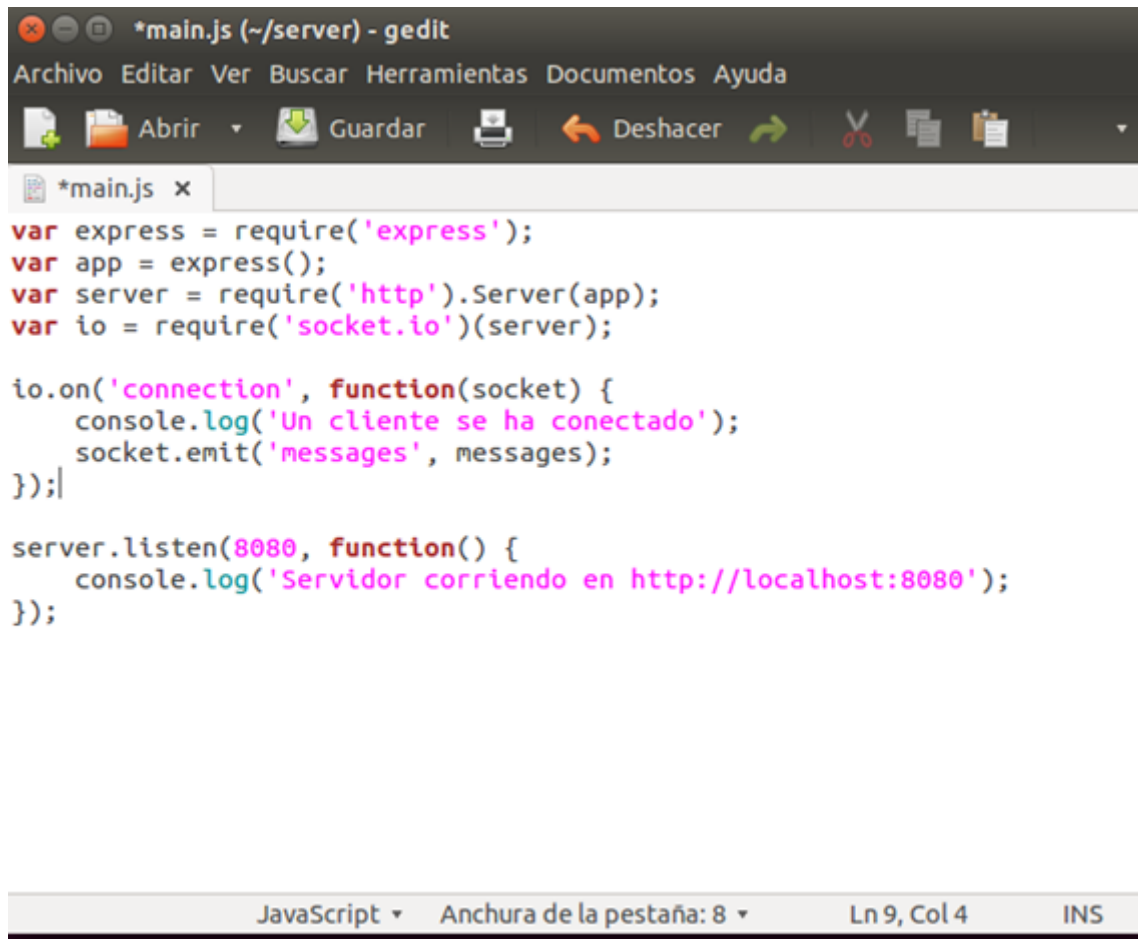


```
usuario@ubuntu:~$ node server/main.js  
Servidor corriendo en http://localhost:8080
```

Para comprobar que funciona, sin cerrar el terminal vamos a Firefox y ponemos: `localhost:8080`. Debe salir lo siguiente:



Abrimos de nuevo el `main.js` y le introducimos lo siguiente para que el servidor de websockets esté atento a que se realice una conexión, cosa que logramos con `io.on()` y pasándole el mensaje `connection`. Dentro de este método enviaremos el array de objetos mensaje con el evento `'messages'`:



```
*main.js (~/.server) - gedit
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
Abrir Guardar Deshacer
*main.js x
var express = require('express');
var app = express();
var server = require('http').Server(app);
var io = require('socket.io')(server);

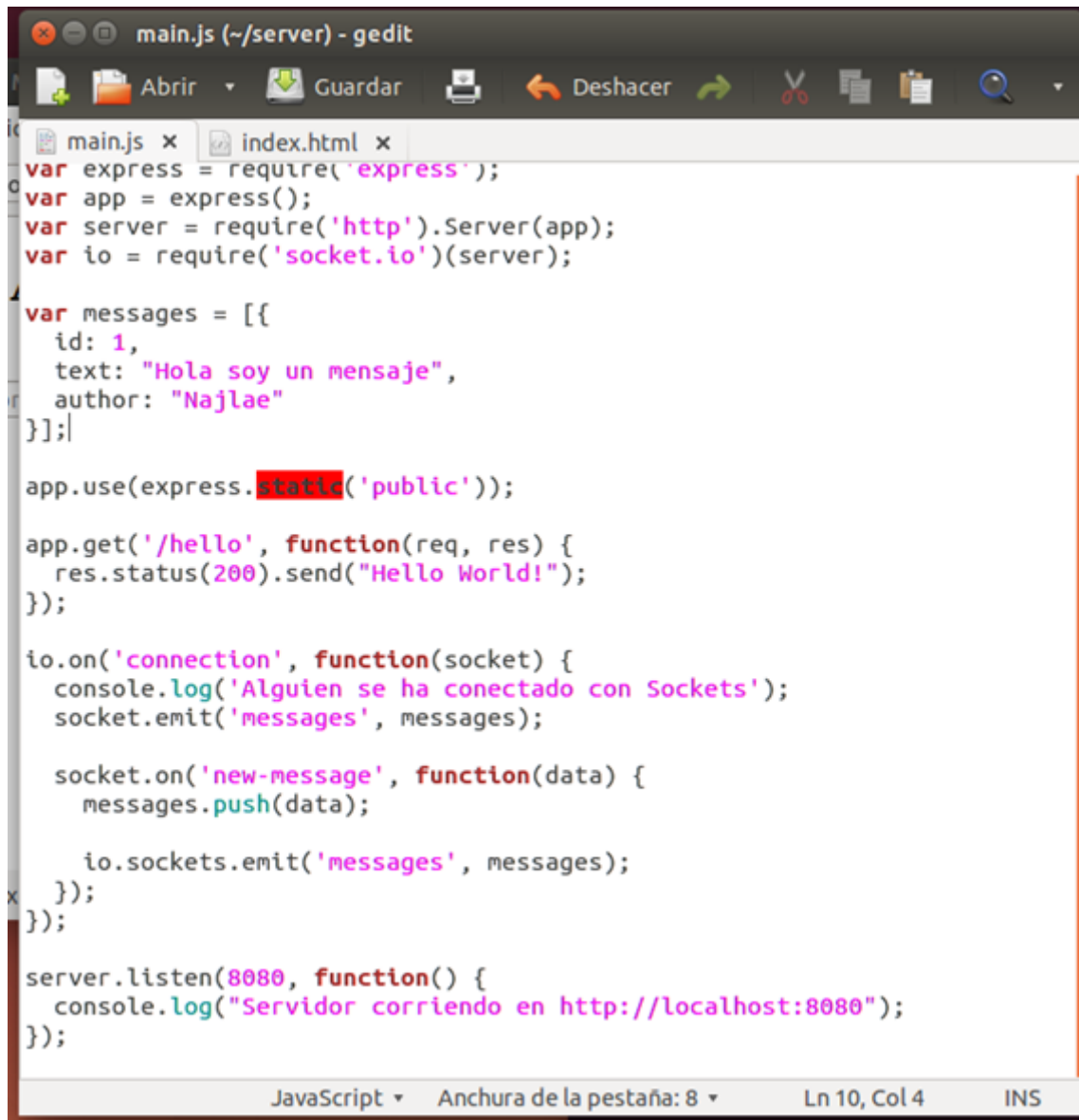
io.on('connection', function(socket) {
  console.log('Un cliente se ha conectado');
  socket.emit('messages', messages);
});

server.listen(8080, function() {
  console.log('Servidor corriendo en http://localhost:8080');
});

JavaScript Anchura de la pestaña: 8 Ln 9, Col 4 INS
```

El evento messages lo recogeremos en el cliente, en el fichero JavaScript de la parte cliente. Así que ahora es momento de crear la parte pública de la aplicación con un index.html y un main.js.

En el documento del server que ya hemos comprobado que funciona lo borramos todo y le pegamos lo siguiente:



```
main.js (~/.server) - gedit
Abrir Guardar Deshacer
main.js x index.html x
var express = require('express');
var app = express();
var server = require('http').Server(app);
var io = require('socket.io')(server);

var messages = [{
  id: 1,
  text: "Hola soy un mensaje",
  author: "Najlae"
}];

app.use(express.static('public'));

app.get('/hello', function(req, res) {
  res.status(200).send("Hello World!");
});

io.on('connection', function(socket) {
  console.log('Alguien se ha conectado con Sockets');
  socket.emit('messages', messages);

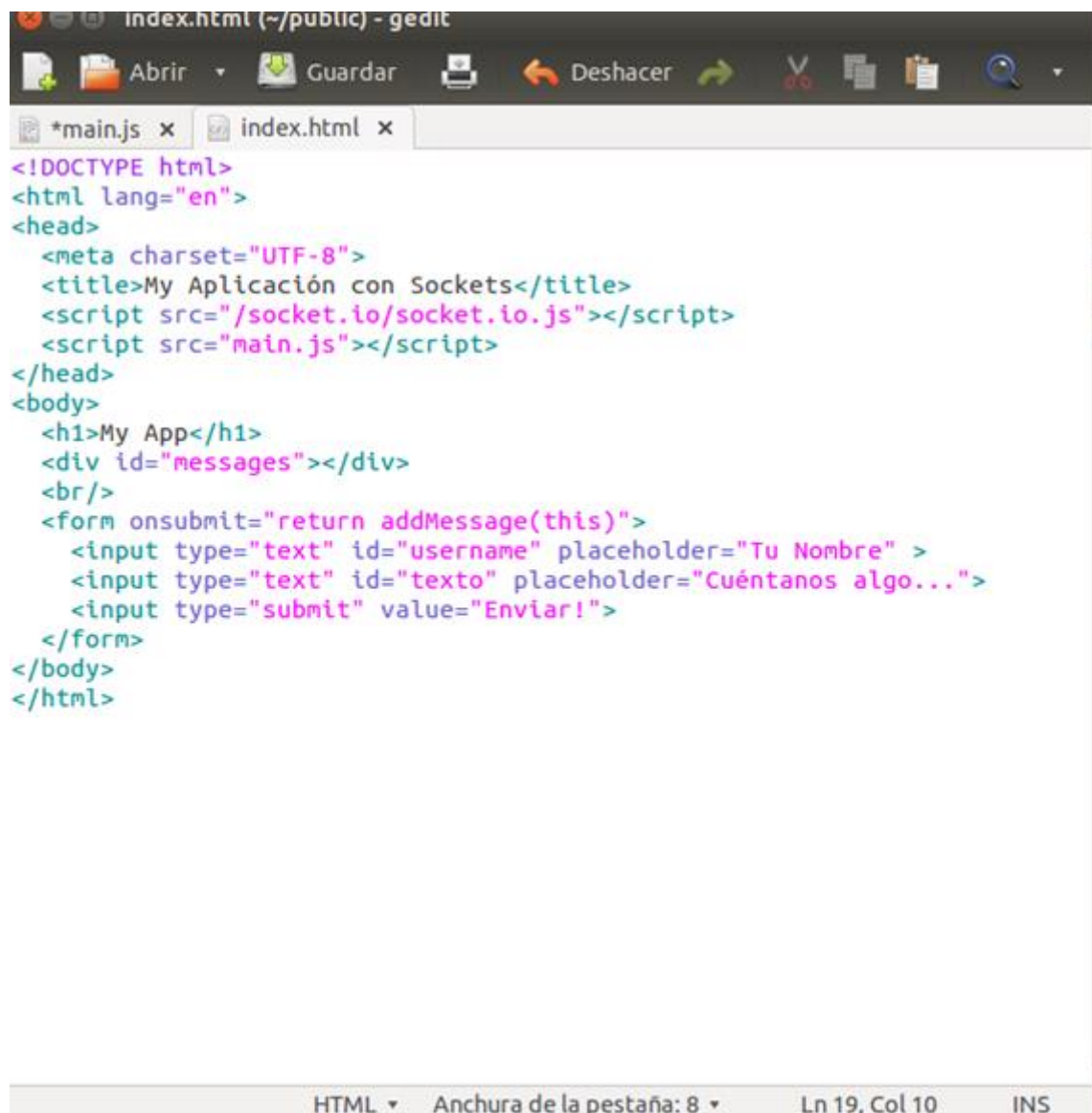
  socket.on('new-message', function(data) {
    messages.push(data);

    io.sockets.emit('messages', messages);
  });
});

server.listen(8080, function() {
  console.log("Servidor corriendo en http://localhost:8080");
});

JavaScript Anchura de la pestaña: 8 Ln 10, Col 4 INS
```

Al index.html del directorio public le pegamos esto:



```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My Aplicación con Sockets</title>
  <script src="/socket.io/socket.io.js"></script>
  <script src="main.js"></script>
</head>
<body>
  <h1>My App</h1>
  <div id="messages"></div>
  <br/>
  <form onsubmit="return addMessage(this)">
    <input type="text" id="username" placeholder="Tu Nombre" >
    <input type="text" id="texto" placeholder="Cuéntanos algo...">
    <input type="submit" value="Enviar!">
  </form>
</body>
</html>
```

HTML ▾ Anchura de la pestaña: 8 ▾ Ln 19, Col 10 INS

Luego vamos al public/main.js y le pegamos el siguiente código de aplicación:

```
*main.js (~/.public) - gedit
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
Abrir Guardar Deshacer

*main.js x
var socket = io.connect('http://localhost:8080', { 'forceNew': true });

socket.on('messages', function(data) {
  console.log(data);
  render(data);
})

function render (data) {
  var html = data.map(function(elem, index) {
    return `<div>
      <strong>${elem.author}</strong>:
      <em>${elem.text}</em>
    </div>`;
  }).join(" ");

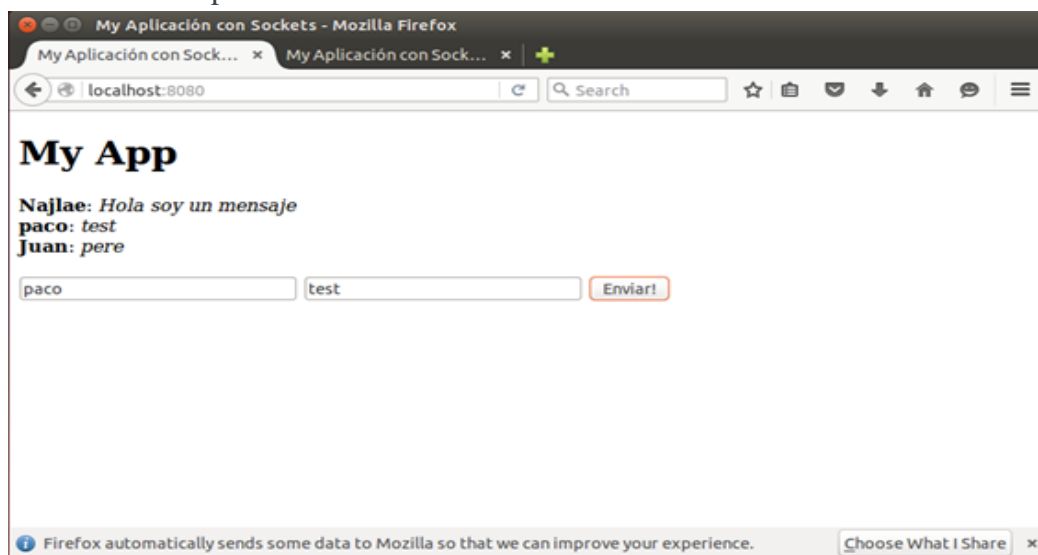
  document.getElementById('messages').innerHTML = html;
}

function addMessage(e) {
  var message = {
    author: document.getElementById('username').value,
    text: document.getElementById('texto').value
  };

  socket.emit('new-message', message);
  return false;
}
```

JavaScript Anchura de la pestaña: 8 Ln 28, Col 1 INS

Escribimos en terminal `sudo node server/main.js`. Abrimos dos ventanas en el navegador en `localhost:8080` y empezamos a mandar mensajes. Veremos que llegan los mensajes a la otra ventana en tiempo real.



9- Ventajas y Desventajas

Tal y como se ha explicado durante el desarrollo del documento podemos afirmar que este protocolo permite establecer comunicaciones bidireccionales en tiempo real en la Web, posibilidad que antes solo existía de forma simulada y bastante costosa mediante técnicas como long polling.

Optar por este protocolo permite reducir la saturación de cabeceras que ocurriría si se utilizase HTTP en su lugar, especialmente para aplicaciones que requieren un gran volumen de comunicaciones.

Además, evita que cada aplicación utilice una solución de integración diferente, con los problemas de compatibilidad que ello conlleva.

Por otra parte, se ha visto que su funcionamiento es extremadamente sencillo: se establece una conexión, se envían/reciben mensajes y se cierra la conexión.

Por último, al funcionar bajo los mismos puertos que HTTP evitamos problemas relacionados con cortafuegos, de esta manera se facilita el intercambio de datos en productos basados en arquitecturas orientadas a servicios.

El principal *inconveniente* es que es necesario gestionar y mantener un gran número de conexiones que han de permanecer abiertas mientras ambas partes sigan interactuando, y esto puede llegar a ser un problema en determinados casos, teniendo en cuenta que el número máximo de conexiones simultáneas que admite un puerto TCP es de 64.000 y que, además, mantener las conexiones abiertas requiere memoria del servidor.

10. Conclusiones

Queda claro que los WebSockets permiten que dos aplicaciones establezcan una comunicación bidireccional independientemente de la plataforma en la que estén ejecutándose y del lenguaje en el que hayan sido escritas.

Que además, existen muchísimas implementaciones para prácticamente cualquier lenguaje que permiten a los desarrolladores centrarse en sus aplicaciones olvidándose de implementar las comunicaciones.

Se abren muchísimas posibilidades para la integración de aplicaciones, permitiendo a éstas intercambiar información en tiempo real y de forma sencilla, contribuyendo además a la estandarización de los mecanismos de comunicación.

Sin duda, parece que WebSocket puede ser una muy buena solución para aplicaciones que necesitan actualizaciones constantes en tiempo real como chats, juegos multijugador en línea o retransmisiones interactivas en directo.

Por el contrario, no resulta una opción tan válida para aplicaciones que únicamente necesitan actualizaciones periódicas o basadas en eventos generados por el usuario.

11. Bibliografía

<http://www.loxone.com/es-es/servicio/documentacion/visualizacion/compatibilidad.html>

<https://azure.microsoft.com/es-es/blog/introduction-to-websockets-on-windows-azure-web-sites/>

<https://www.w3.org/TR/websockets/>

<http://caniuse.com/#feat=websockets>

<http://www.html5rocks.com/en/tutorials/websockets/basics/>

https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

<https://code.msdn.microsoft.com/windowsapps/Connecting-with-WebSockets-643b10ab/>

<http://www.adictosaltrabajo.com/tutoriales/web-sockets-java-tomcat/>

<https://developer.mozilla.org/es/docs/WebSockets>

https://developer.mozilla.org/es/docs/WebSockets-840092-dup/Escribiendo_servidor_WebSocket

<http://www.arkaitzgarro.com/html5/capitulo-13.html>

<http://www.arquitecturajava.com/java-websockets/>

<https://www.nuget.org/packages/WebSocket.Portable.Core/>

<http://lineadecodigo.com/html5/crear-un-websocket/>

http://www.htmlgoodies.com/html5/other/create-a-bi-directional-connection-to-a-php-server-using-html5-websockets.html#fbid=FDhH9_nbENA

<https://www.sanwebe.com/2013/05/chat-using-websocket-php-socket>

<http://www.sitepoint.com/how-to-quickly-build-a-chat-app-with-ratchet/>

<http://socketo.me/docs/websocket>