S9: Websockets

Jump to bottom

Juan Gonzalez-Gomez edited this page on 3 May · 118 revisions

Sesión 9: Websockets

- **Tiempo**: 2h (50 + 50min)
- **Fecha**: Lunes, 26-Abril-2021
- **Objetivos de la sesión**:
  - Entender los Websockets
  - Ejemplos de uso de websockets

- Manejar la biblioteca socket.io

Contenido

Introducción

Los **websockets** nos permiten tener comunicación **bidireccional** y **full-duplex** entre clientes y servidores. En esta sesión los estudiaremos y veremos ejemplos de uso

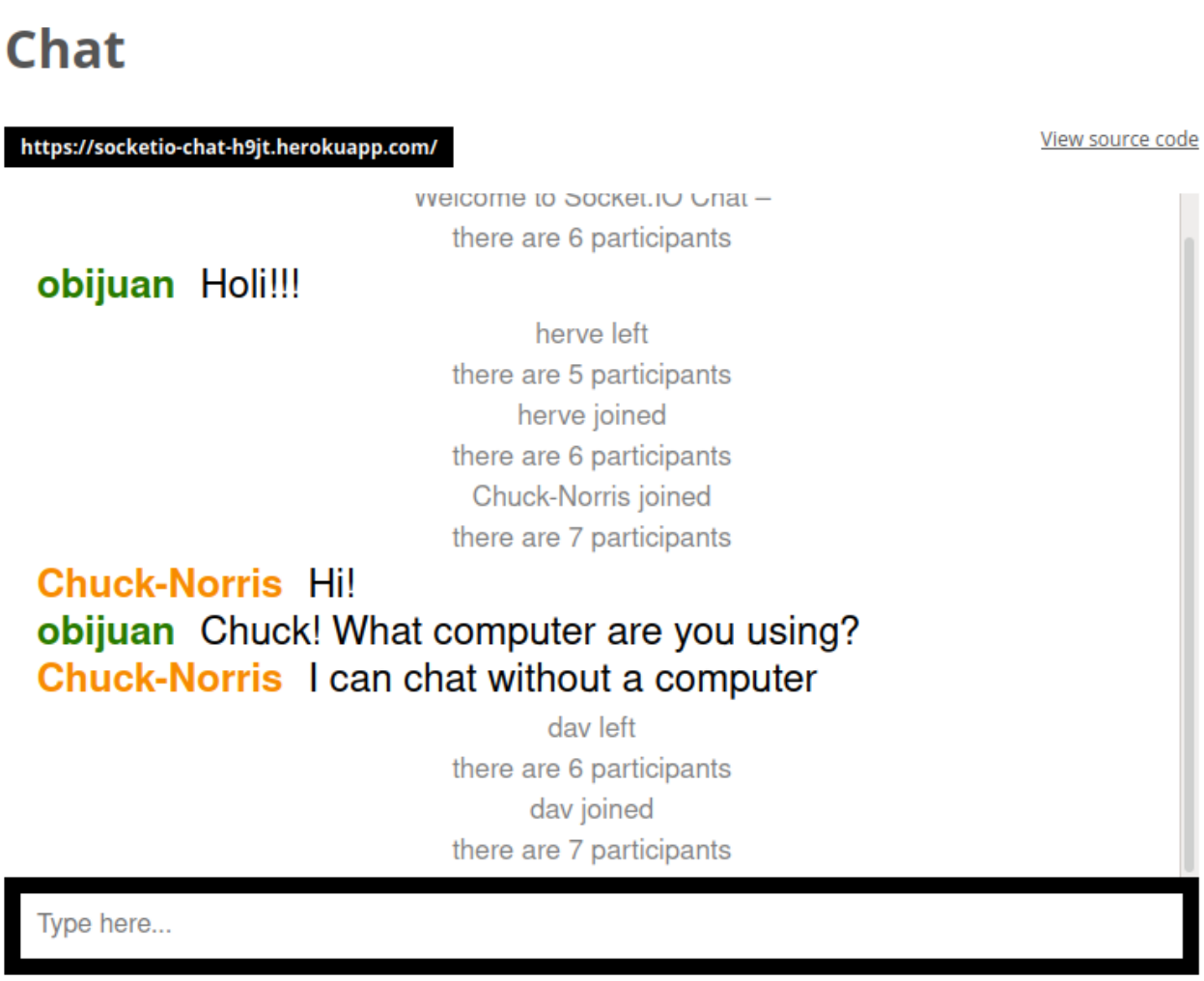
Aplicaciones Web en tiempo real

¿Cómo podemos hacer **aplicaciones Web interactivas** en **tiempo real**? Ejemplos de estas aplicaciones serían:

- **Sistemas de monitorización**: Un servidor nos envía datos sobre el estado de lo que estamos monitorizando: alarmas, sensores, pacientes, máquinas, la bolsa...

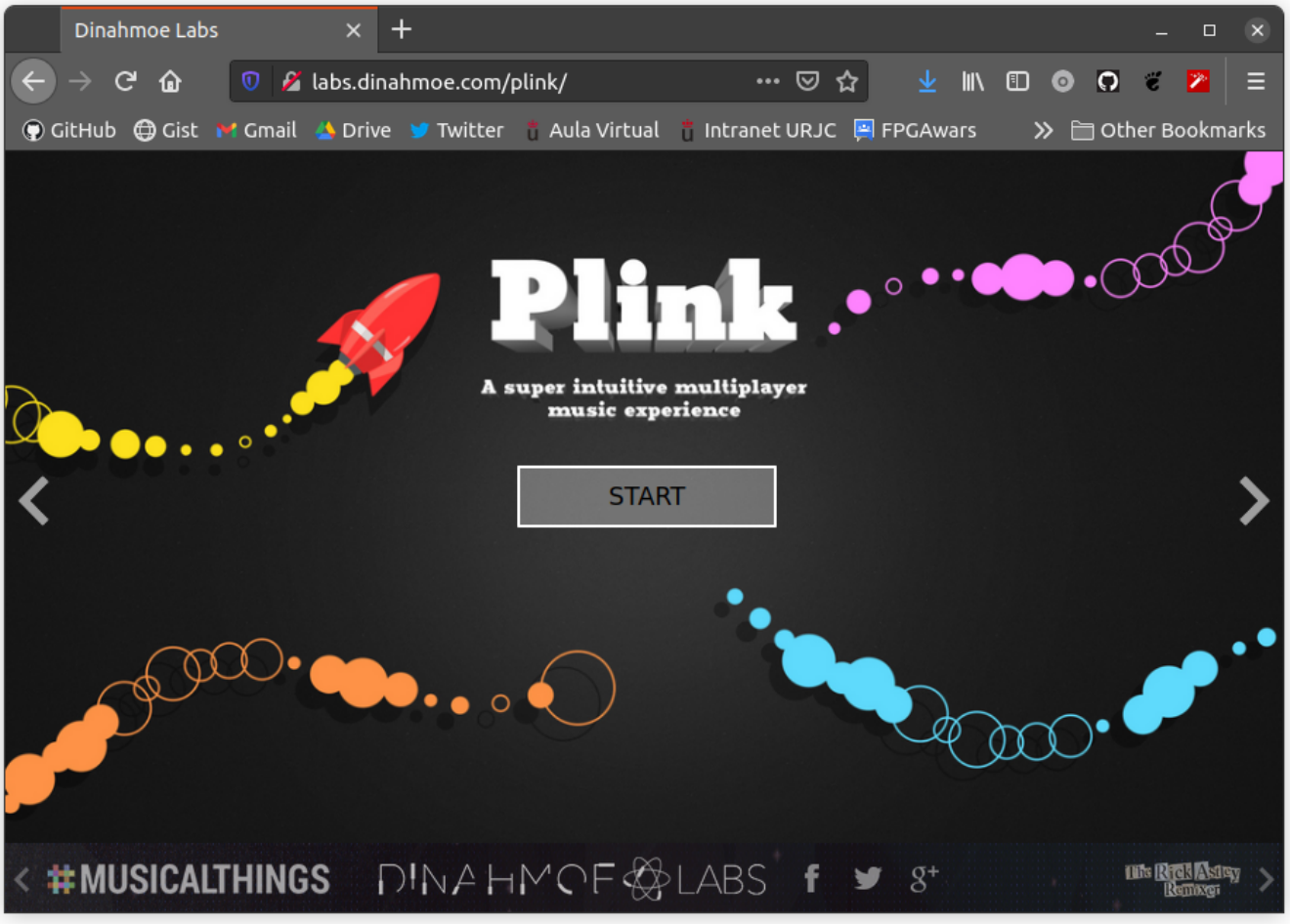
(Fuente: collectdViewer)

- **Chat Web**: Múltiples usuarios enviándose mensajes entre ellos, desde sus navegadores



(Fuente: Socket.io)

- **Juegos multijugador**: Todos los juegadores ven en tiempo real lo que hacen los demás



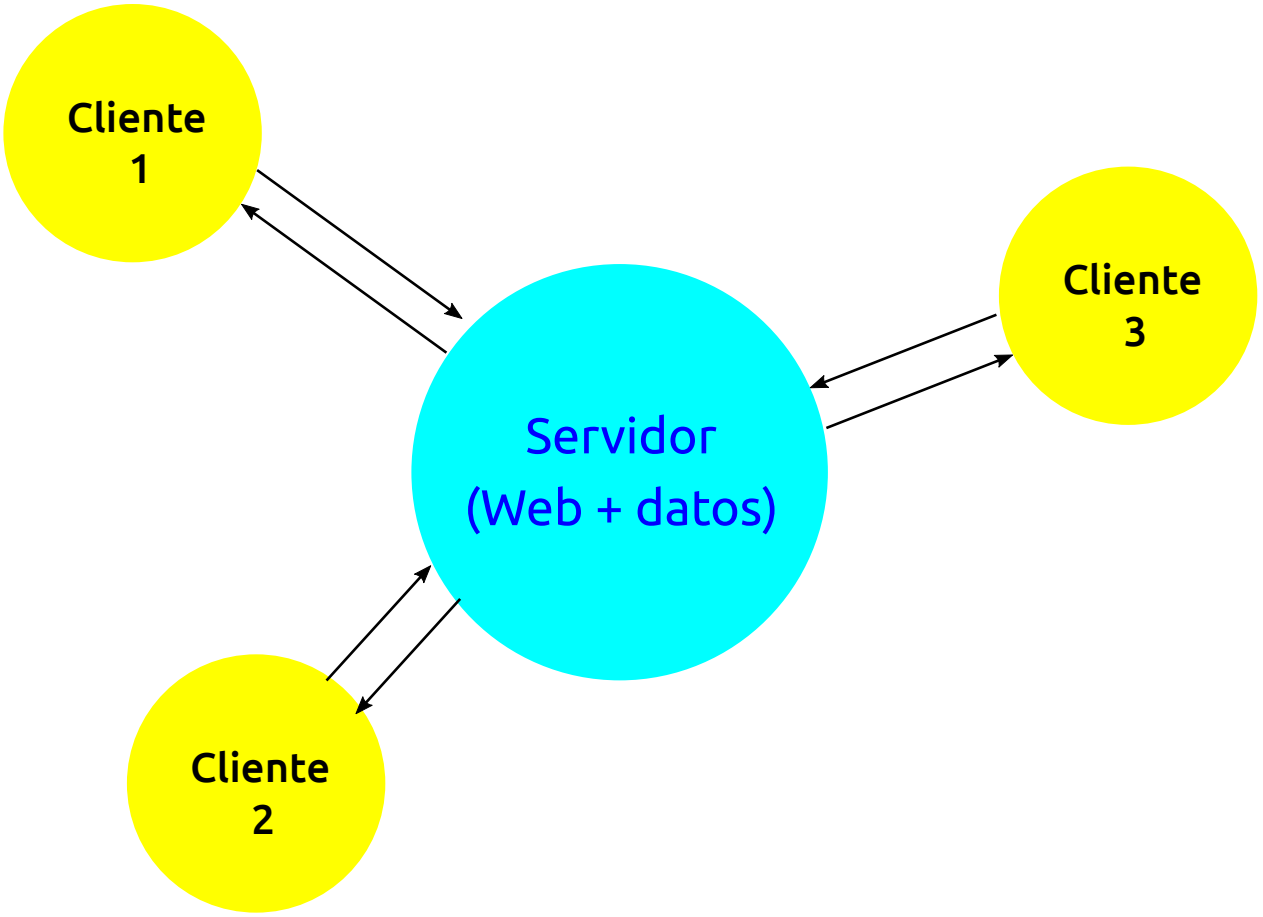
(Fuente: Dinahmoe-Plink)

- **Pizarra interactiva multiusuario**: Todos los usuarios pueden pintar en la pizarra, y ven lo que el resto está dibujando

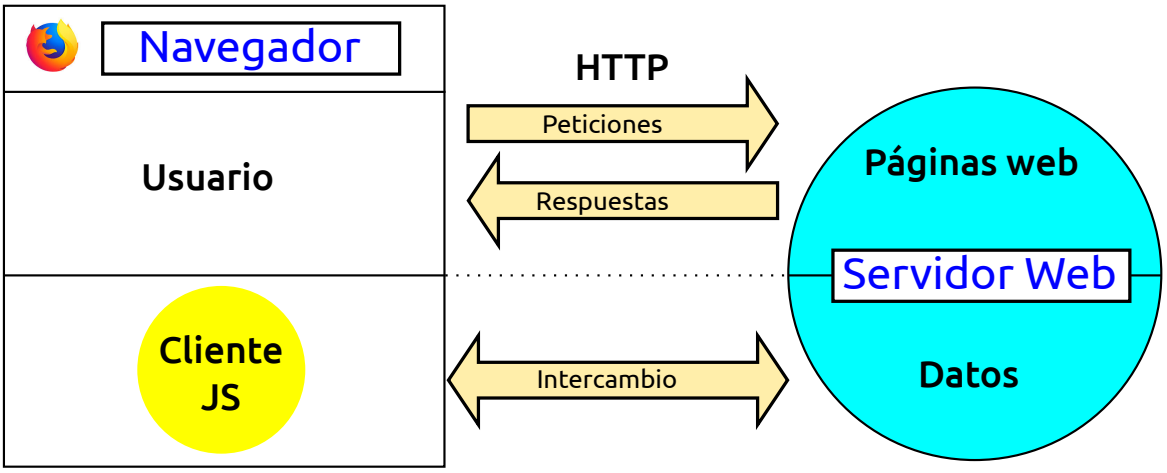
(Fuente: Socket.io)

Arquitectura

Todas estas aplicaciones tienen una **arquitectura Cliente-servidor**. Hay un **servidor web** al que se conectan todos los clientes. Este servidor envía la **interfaz web** (html, css, imágenes, javascript...) a todos los clientes, mediante **HTTP**. Además, recopila la información que le envían los clientes y se la re-envía al resto de clientes



Es decir, este servidor contiene las **páginas** para los navegadores (humanos) y ofrece **datos** a las aplicaciones **javascript** que se ejecutan en los clientes. En cada cliente al menos habrá **dos agentes** intercambiando información con el servidor: El **navegador** y la **aplicación Javascript** (que se ejecuta dentro del navegador a su vez)



La clave está en el **intercambio de datos** entre la **aplicación JS** y el **servidor**

Intercambio de datos entre Cliente JS y servidor

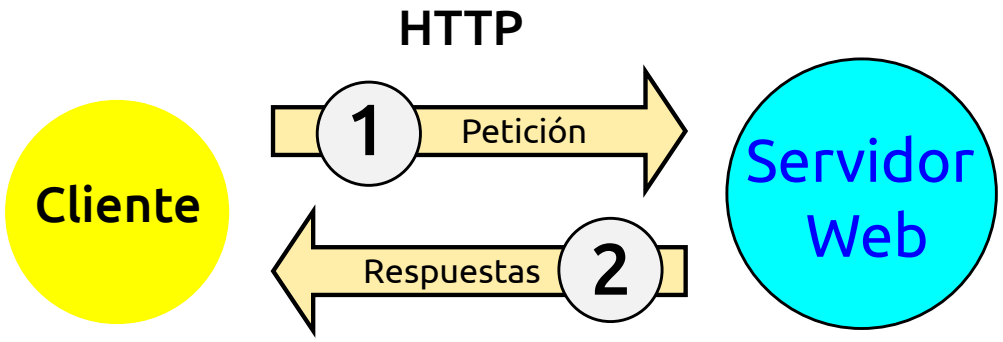
Con lo que conocemos hasta ahora, ¿Cómo podemos implementar este intercambio de datos entre la aplicación JS y el servidor?. Vamos a analizar las difernetes posibilidades. Como **ejemplo de aplicación** pensaremos en un **sistema web de Monitorización**. Queremos conocer periódicamente los datos de un sistema (temperatura, alarma, estado...)

Las **aplicaciones WEB** funcionan con el **protocolo HTTP**, que ya conocemos. Por debajo se usan **conexiones TCP**, y se usan los **puertos** estándares **80** y **443** (TLS)

Limitaciones de HTTP

El **protocolo HTTP** estaba inicialmente pensado para solicitar **páginas web**. Si lo queremos utilizar para aplicaciones en tiempo real, como las indicadas anteriormente (monitorización, juegos multijugador...), presenta una serie de **limitaciones**:

- **Ciclos de Respuesta/Solicitud**. Es un protocolo basado en Respuesta/Solicitud. **El servidor no hace nada activamente**. Espera a recibir peticiones y devolver respuestas. Si el servidor dispone de nueva información, **No se la puede entregar al cliente** hasta que éste no le haga **una petición**



- **HTTP es Half-dúplex**. El tráfico sólo fluye en una dirección cada vez. Bien del cliente al servidor (petición) o del servidor al cliente (respuesta). Pero **NO hay comunicación bidireccional simultánea** (Full-Duplex)
- **HTTP es un protocolo sin estado**: Cada petición es una **transacción independiente**, que no tiene relación con las anteriores. NO se **recuerda** nada de lo que ocurrió en el pasado. Por ello, se envía muchísima información redundante en cada respuesta y solicitud (Por ejemplo: las cookies). Se envían todo el rato del cliente al servidor para indicar información de estado (quién soy, qué he comprado, cuándo me conecté por última vez...)
-

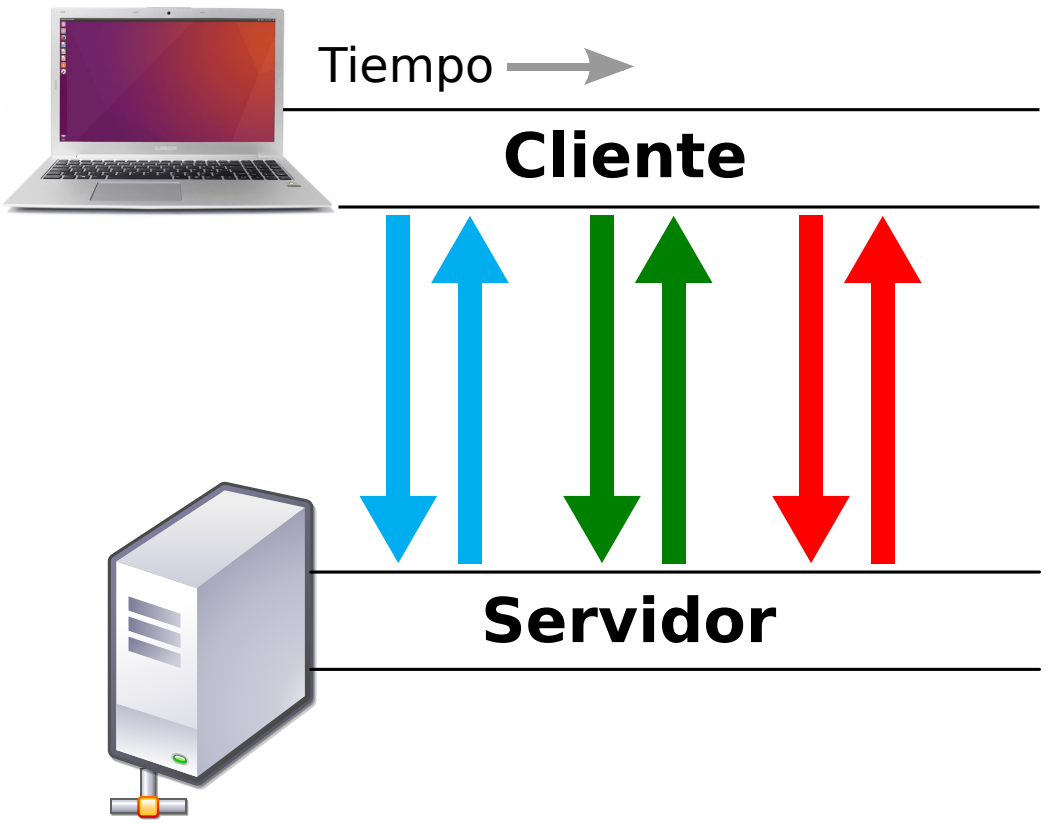
Alta sobrecarga: En las **cabeceras de HTTP** va mucha información. Mucha más que los **datos útiles**. Ejemplo: Recepción de un dato para lectura de la temperatura (4 bytes): Las cabeceras puedes ocupar unos 800 bytes. Y típicamente con las cookies pueden llegar a los 2Kbytes. ¡¡Para recibir la temperatura (4 bytes) en total se intercambian más de 800 bytes!!

Implementando las aplicaciones web en tiempo real con HTTP

OK. HTTP tiene muchas limitaciones. Aún así, ¿Qué opciones tenemos para implementar las aplicaciones web en tiempo real? ¿Lo podemos hacer?
Sí, aunque no es ni mucho menos lo óptimo

Sondeo con AJAX (Ajax polling)

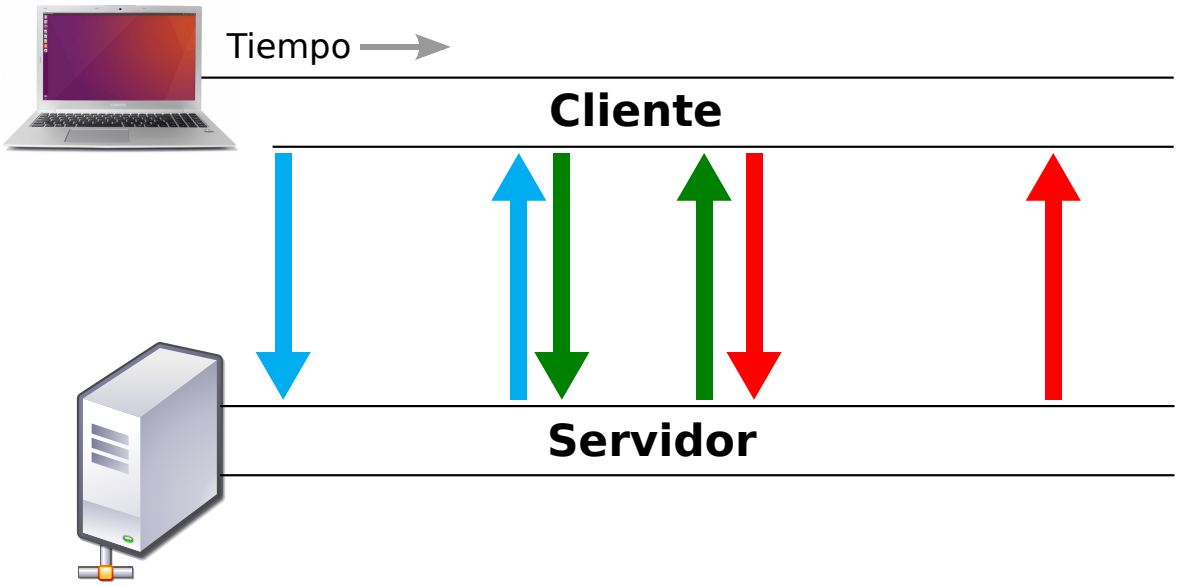
Ya conocemos una manera en la que nuestros programas Javascript puede realizar **peticiones HTTP** de manera paralela a las que hace el navegador, utilizando **AJAX**. La técnica del **sondeo periódico** consiste en enviar peticiones HTTP regularmente del cliente al servidor



Con esto se consigue "casi tiempo real"... pero a costa de **incrementar mucho** el **ancho de banda** usado

Sondeo Asíncrono

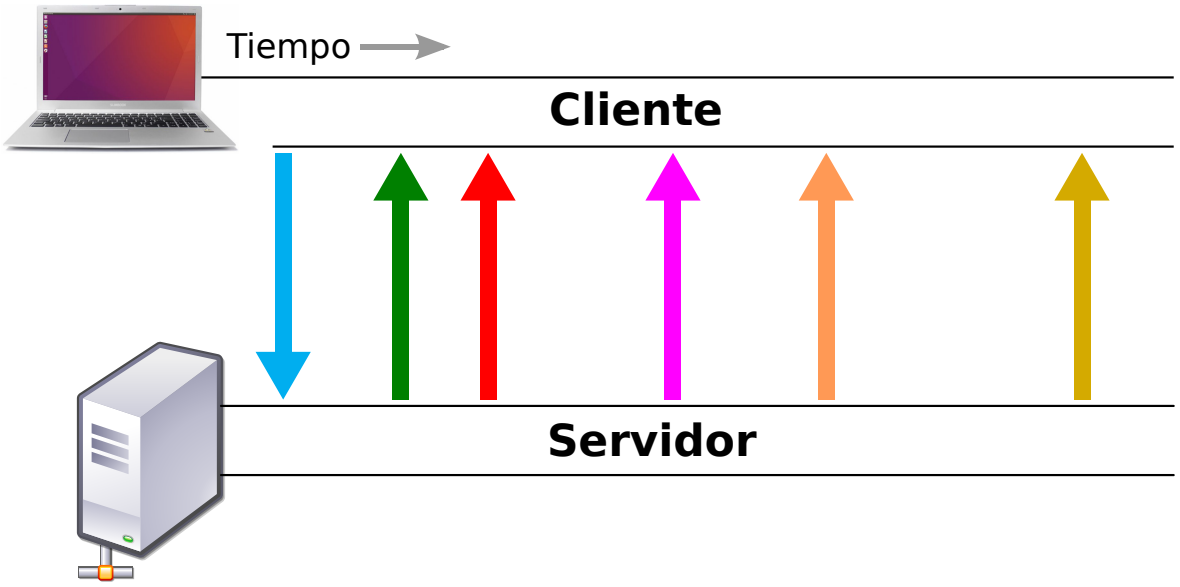
El **cliente** envía una **petición AJAX** al servidor, y se mantiene **abierta** hasta que **haya datos disponibles**. El **servidor** envía la **respuesta** al cliente. El cliente envía otra petición **inmeditamente**



Se gasta menos ancho de banda ya que el servidor **sólo envía respuestas** cuando hay nuevos **datos disponibles**. La velocidad está **limitada** por el ciclo respuesta-petición-respuesta

Eventos enviados por el servidor (SSE: Server Sent Events)

El estándar Server-Sent Events describe cómo los **servidores** pueden **enviar datos directamente al cliente**, una vez que se ha establecido la **conexión inicial**. El cliente envía un mensaje para establecer la conexión y a partir de ahí el servidor envía respuestas cada vez que tiene datos disponibles. Esta conexión permancece abierta.



Se utiliza para hacer *streaming*. El inconveniente es que la comunicación es **unidireccional**: del Servidor al Cliente. Nos servidría para **aplicaciones de monitorización**, donde el servidor envía constantemente el estado del objeto a monitorizar, pero **NO** para **aplicaciones interactivas** como *chats* o *juegos multijugador*, donde los clientes deben enviar información al servidor

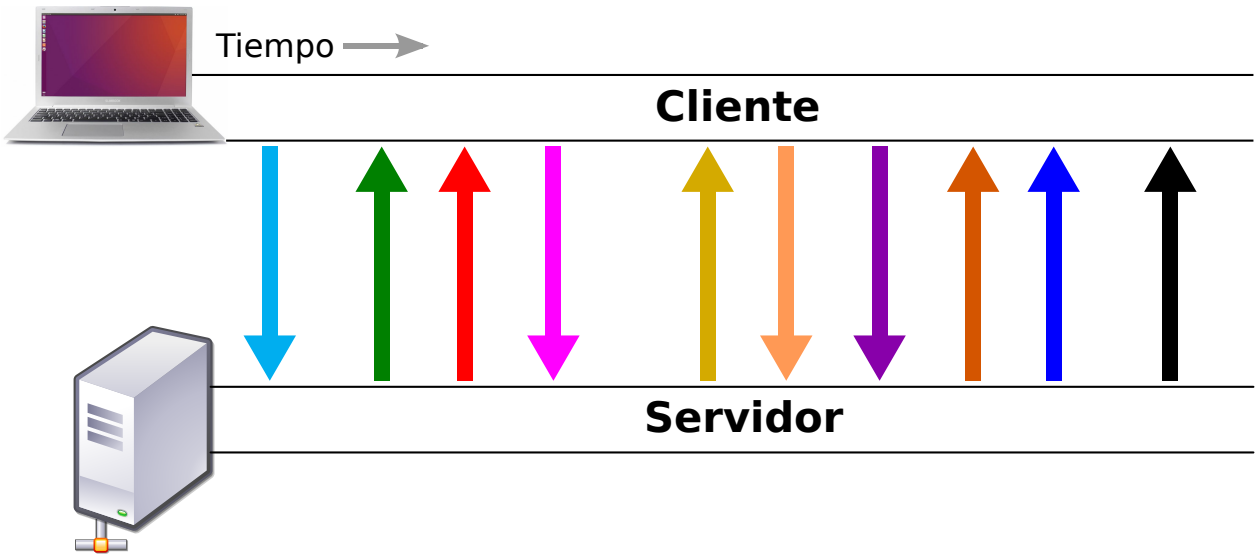
Conexión directa por TCP

TCP crea **canales de datos** entre dos **aplicaciones** situadas en **máquinas diferente**s en internet. Estos canales son **Full-duplex**. Una posibilidad es que los clientes se conecten al servidor por estos canales para el intercambio de datos. Habría que programarlo todo desde cero, pero es viable

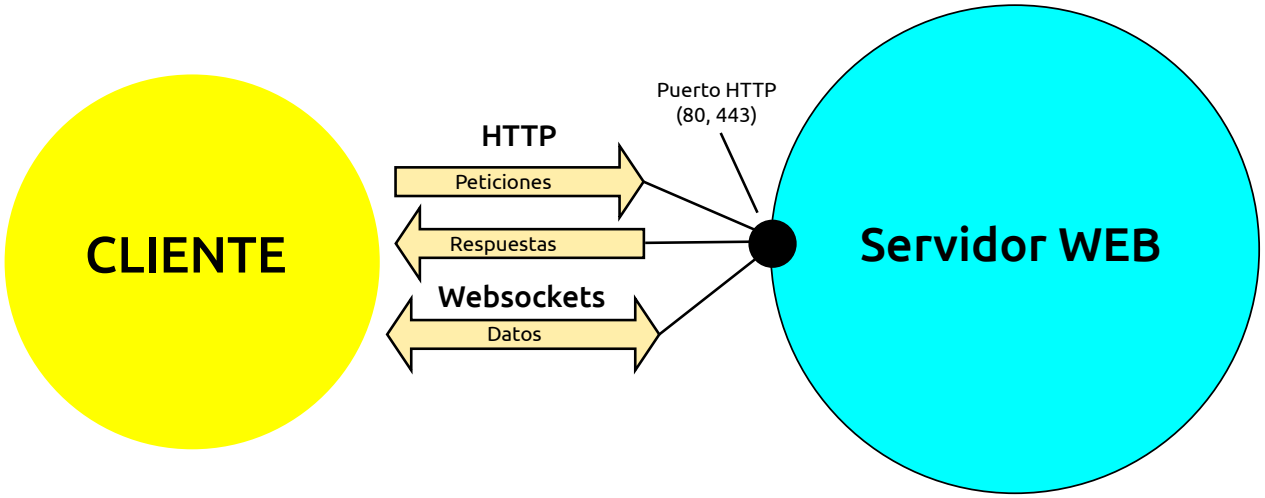
¿Cuál es el inconveniente? Si bien esta solución funcionaría bien si estamos en nuestra propia red privada, no funcionaría en internet, ya que estas conexiones TCP se harían por **puertos NO estandarizados**. La web funciona con los **puertos 80** y **443** y por ello están siempre abiertos, sin embargo el resto de puertos, por razones de seguridad, están cerrados en los **proxies** y **firewalls**

Websockets

Todos los problemas anteriores se solucionan utilizando WebSockets. Nos proporcionan un **canal de comunicación bidireccional** y **full-duplex** entre cliente y servidor



PERO sobre un **único Socket TCP**, que se **comparte** con el usado para el **contenido HTTP**. Por ello, **atraviesa** sin problemas los **Firewalls** y **proxies**



WebSockets ES UN **PROTOCOLO INDEPENDIENTE**, que utiliza el **mismo puerto HTTP** ya existente. El **servidor web** es el que diferencia entre **HTTP** o **websockets**, atendiendo los mensajes acorde al tipo. Pero para el resto de elementos (proxies, firewalls) son accesos Web normales

Protocolo

WebSockets es un **protocolo independiente** que está **encima de TCP**. La **negociación inicial** (HandShake) se hace a través de **HTTP**. El **cliente** envía un **mensaje de Solicitud** en el que incorpora las cabeceras definidas en la negociación de WebSockets:

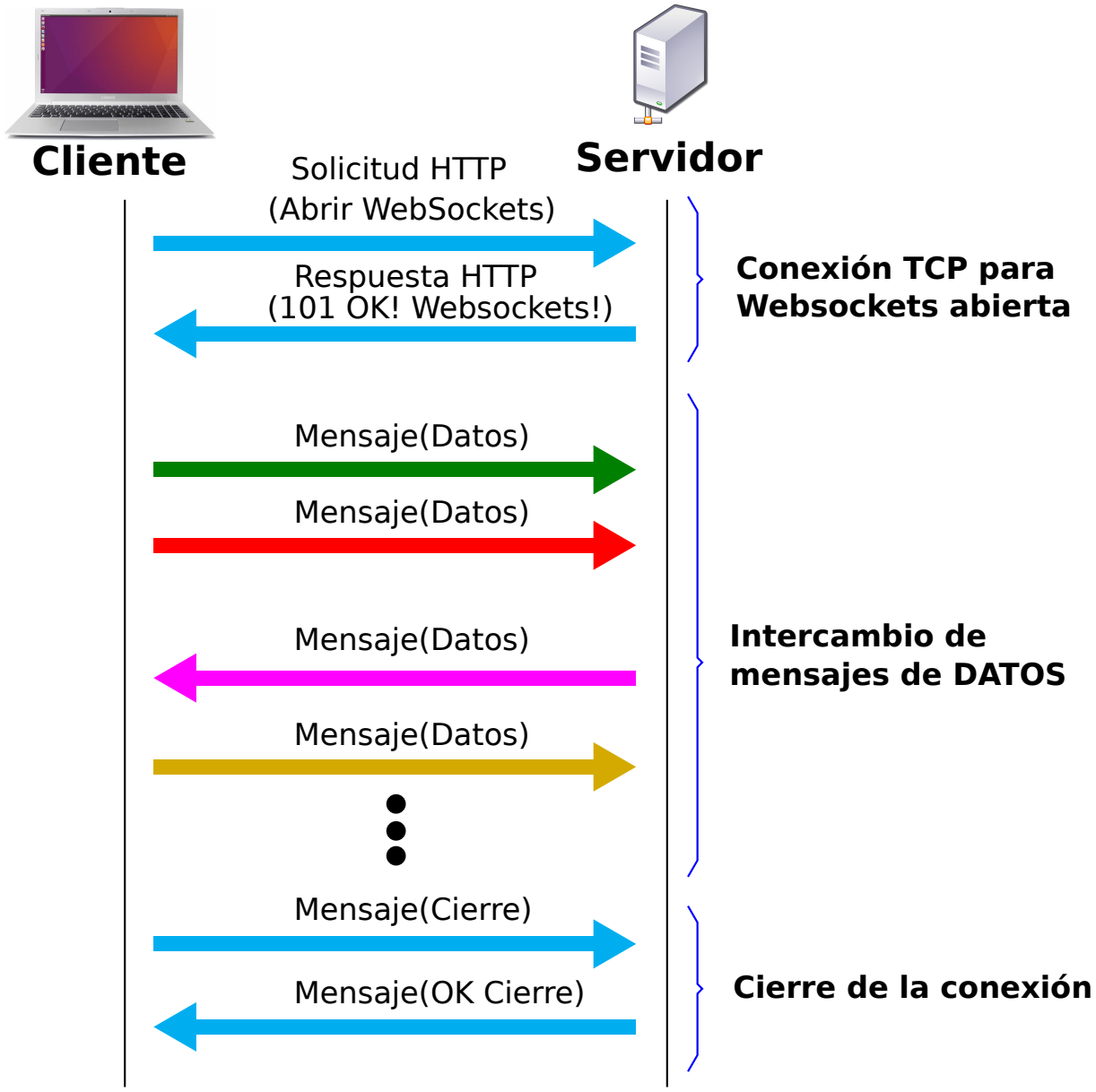
```
GET /mychat HTTP/1.1
Host: server.example.com
```

```
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

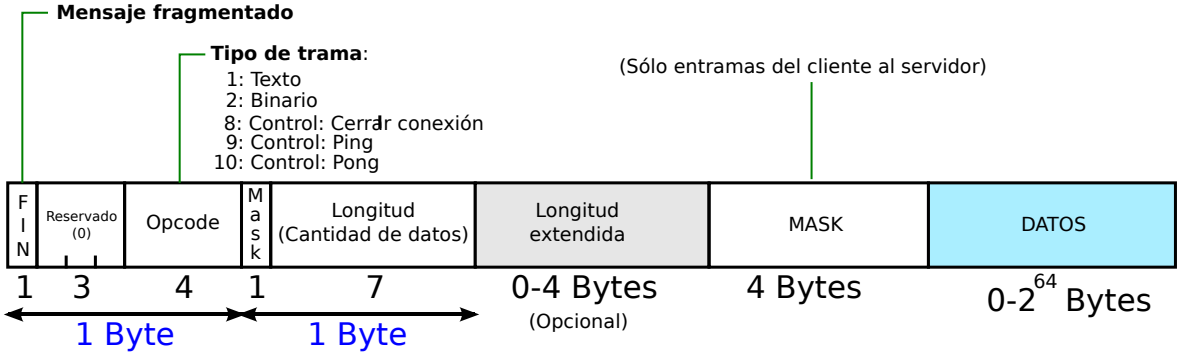
Si el **servidor** soporta WebSockets, devolverá un **mensaje de respuesta** como el siguiente:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

El código de respuesta es **101**, que indica que se ha **actualizado** el protocolo a **Websockets**. Es decir, que a partir de ese momento se pueden ya se pueden **intercambiar datos** a través de **Websockets**. En este dibujo se muestra el funcionamiento



Las **tramas del protocolo** WebSockets tienen esta estructura. ¡Sólo tienen una cabecera obligatoria de 2 bytes!



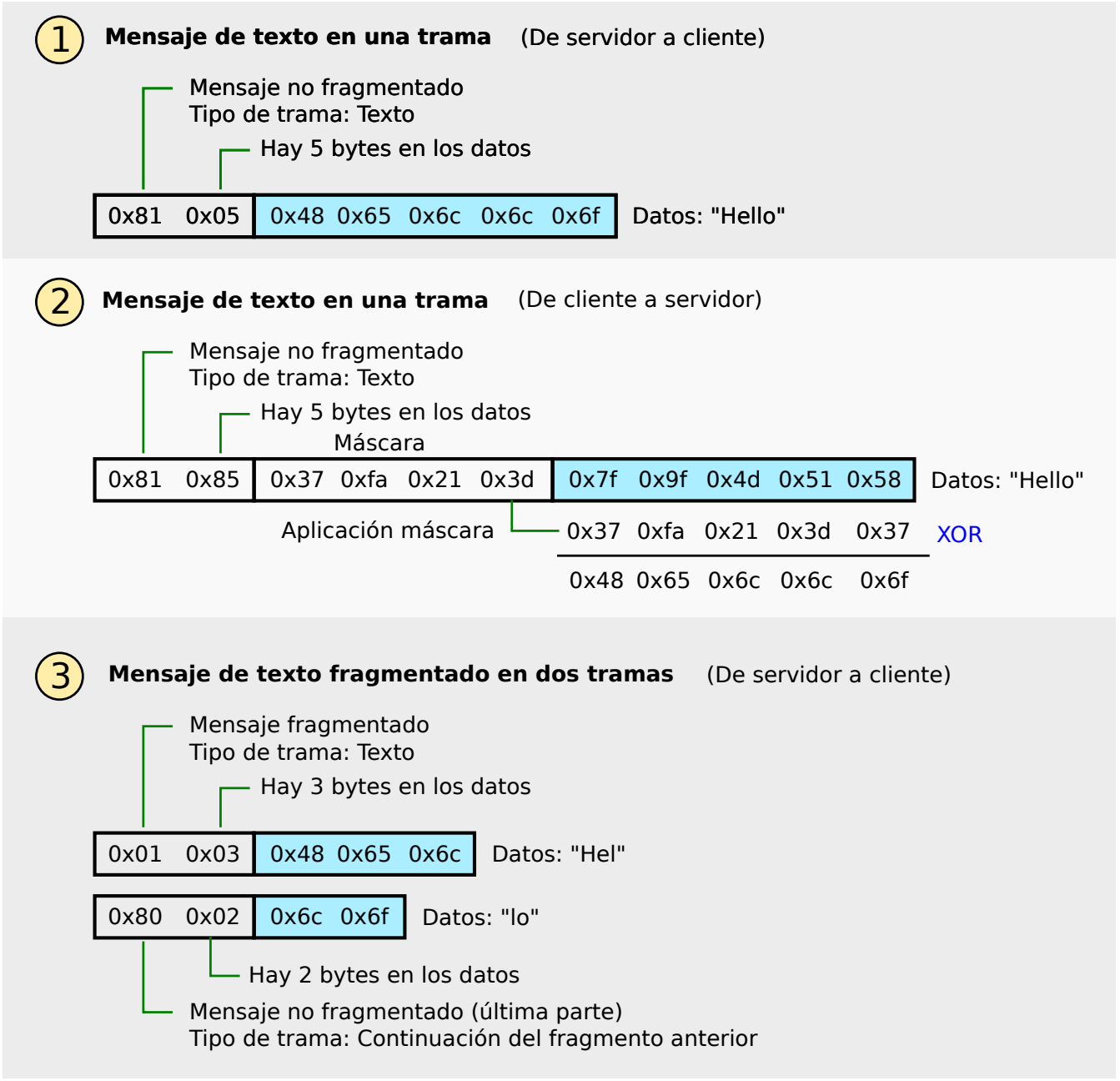
La **longitud máxima** de datos a enviar es de **64GBytes**

Ejemplos de tramas Websockets

En esta figura vemos algunos **ejemplos de tramas** en las que se envían mensajes de texto con los datos "Hello" (Ejemplos sacados del RFC6455). La **primera trama** representa un **mensaje de texto** enviado del **Servidor** al **cliente**. El tamaño es de 5 bytes y como no está fragmentando el bit FIN está a '1' (es el último trozo)

La **segunda trama** es el mismo mensaje pero enviado desde el cliente al servidor. En esa dirección hay que aplicar una máscara de 32 bits (operador XOR) a los datos. Por eso los 5 bytes de los datos son diferentes a los de la trama anterior. Pero si se aplica la máscara se obtieene el mensaje "Hello"

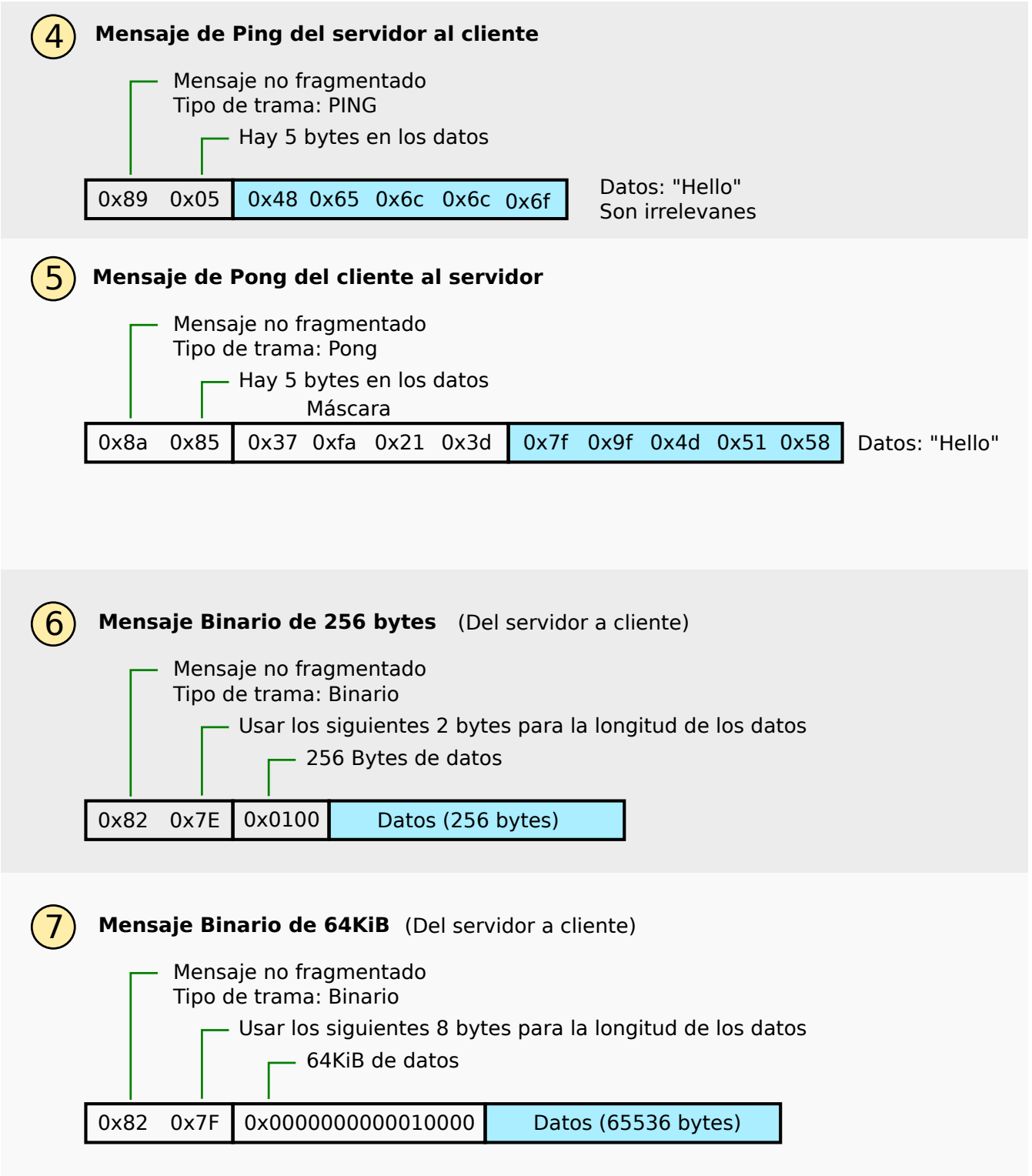
La **última trama** es el mismo mensaje del ejemplo 1 pero fragmentado en dos tramas. En el primer fragmento se envía "Hel" y en el segundo "lo". Como va del servidor al cliente no hay máscara



Con estos ejemplos nos damos cuenta de lo **cortas** que pueden ser las tramas. Esto mismo con el protocolo HTTP implicaría enviar muchísimos más bytes. Websockets es un protocolo **muy eficiente**

Aquí vemos más ejemplos. Las **tramas 4** y **5** se corresponden con tramas de tipo **Ping** y **Pong** respectivamente. La primera del servidor al cliente y la segunda es la respuesta del cliente al servidor, por lo que se usa la máscara

Las tramas **6** y **7** son mensajes Binarios de envío de datos desde el servidor al Cliente. En la primera se envían 256 bytes. El campo de longitud vale **0x7E** (126) lo que indica que la longitud se debe leer de los 2 siguientes bytes. La la segunda se envían 64KiB. Para ello hay que poner **0x7F** en el campo de la longitud y en los **8** siguientes bytes el **tamaño de los datos**



URI para Websockets

Para **denominar** los Websockets se utilizan dos protocolos nuevos: **ws://** y **wss://** (TSL). Esto nos permite nombrar los recursos para establecer las conexiones con Websockets. Por ejemplo:

```
ws://subdomain.examnple.com/some-endpoint
```

API Javascript

La API de WebSocket de los clientes está siendo normalizada por el **W3C**.

Los **navegadores** que implementan *WebSockets** tiene definido el objeto `window.WebSocket` que nos da **acceso a la API**

Para comprobar si el navegador lo tiene implementado usamos este código:

```
//-- Comprobar si existe el objeto WebSocket
if (window.WebSocket)
  console.log("Websockets DISPONIBLES");
else
  console.log("Error: Websockets NO soportados en este navegador");
```

Se definen los siguiente **eventos** para establecer las **funciones de retrollamada**

- `open` : Apertura de conexión. El cliente se ha conectado a un servidor de Websockets correctamente
- `close` : Cierre de conexión. La conexión ha finalizado
- `error` : Se ha producido algún error
- `message` : Se he recibido un mensaje nuevo

Para **trabajar** con los websockets primero es necesario crear un objeto de tipo `WebSocket` y especificar la **URL de conexión** como argumento. Por ejemplo:

```
cont mySocket = new WebSocket('ws://www.WebSocket.org');
```

Las funciones de retrollamada reciben como argumento un evento `e` que tiene los atributos `data` y `code` para indicar los datos recibidos y el código de cierre en caso de cerrarse la conexión. Este es un ejemplo de plantilla del código:

```
//-- Crear el Websocket
const mySocket = new WebSocket("ws://www.WebSocket.org");

// Establecer las funciones de retrollamada para TODOS los eventos
mySocket.onopen = (e) => {
  console.log("Conexión establecida");
};

mySocket.onclose = (e) => {
  console.log("Conexion cerrada. Código: " + e.code};
};

mySocket.onmessage = (e) => {
  console.log("Mensaje recibido: " + e.data);
```

```
};

mySocket.onerror = (e) => {
  console.log("Error);
};
```

Se definen los métodos `send` y `close` para enviar datos y cerrar la conexión:

```
//-- Envío de datos
mySocket.send("Chuck Norris approved!!");

//-- Cerrar el Websocket
mySocket.close();
```

Además estas son algunas de la **propiedades** que se definen (son de sólo lectura):

- **WebSocket.bufferedAmount**: Cantidad de bytes recibidos y que están encolados
- **WebSocket.protocol**: El protocolo elegido por el servidor
- **WebSocket.readyState**: Estado de la conexión
- **WebSocket.url**: La URL del websocket

Pruebas desde la linea de comandos

Antes de empezar a hacer **programas** con los **websockets** vamos a realizar pruebas desde la **línea de comandos**. Esto nos servirá más adelante para **probar** nuestros **servidores de WebSockets** fácilmente

Utilidad wscat

El comando que utilizaremos para las pruebas es wscat, que nos permite enviar datos usando Websockets, así como recibirlos. Es un **paquete de node** que se puede instalar a través de **npm**:

```
npm install -g wscat
```

Hay que instalarlo con el flag `-g` para que se instale **globalmente** y poder usarlo como cualquier otro comando

Ejemplo 1: Conexión al websocket ws://echo.websocket.org

Para probar que funciona correctamente el comando `wscat` vamos a conectarnos a este Websocket: `ws://echo.websocket.org` . Es un **servidor de Eco** alojado en la página WebSocket.org. Todos los datos que reciba los enviará de vuelta. Este tipo de servidores son muy útiles para hacer pruebas

Nos **conectamos** con el siguiente **comando**:

```
wscat --slash -P -c ws://echo.websocket.org
```

Hemos usado los siguientes parámetros:

- `-P` : Imprimir una notificación cuando se recibe un mensaje **Ping** o **Pong**
- `--slash` : Permite enviar tramas de control de Websocket con `/ping`, `/pong` y `/close`
- `-c` : Configurar como cliente

Al ejecutarlo se **conecta al Websocket** y nos permite **enviar** los **mensajes de texto** que escribamos. Las notificaciones y mensajes recibidos se muestran en azul. En este **pantallazo** vemos un ejemplo de uso:

Ejemplo 2: Cliente y servidor Websocket con wscat

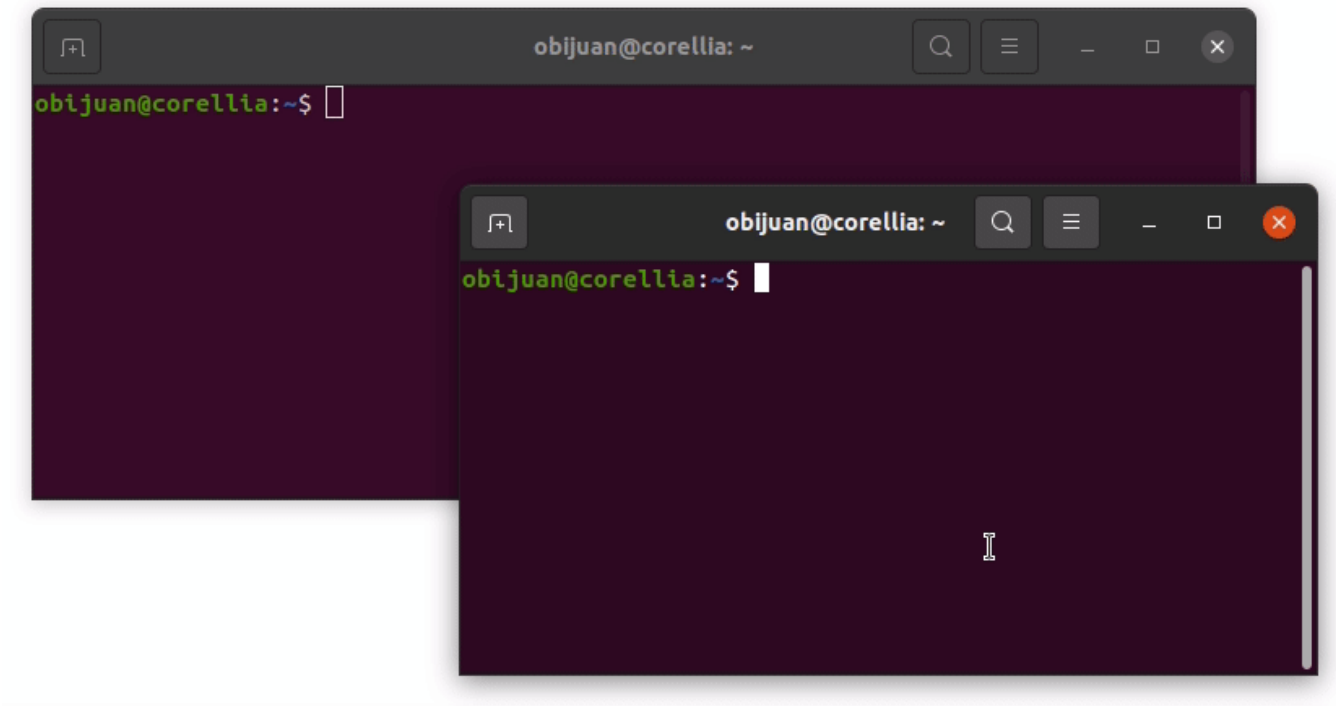
Desde un terminal lazamos un servidor de Websocket, llamando a `wscat` con el parámetro -l y el número de puerto:

```
wscat -l 8000
```

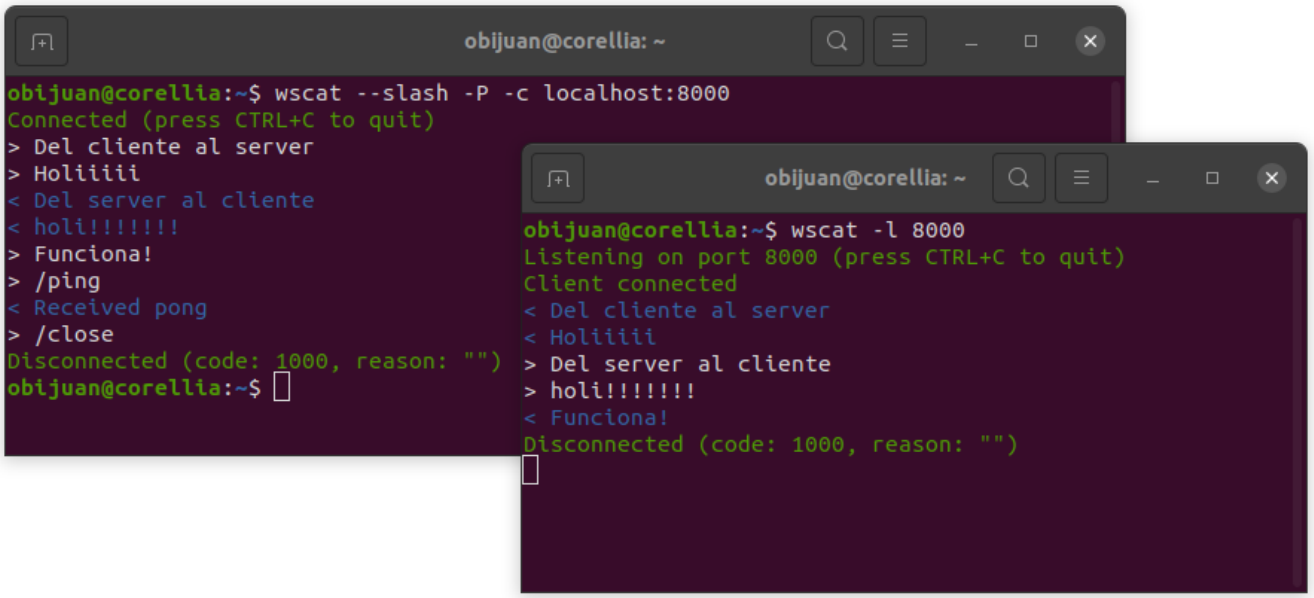
Desde otro terminal lanzamos el cliente:

```
wscat --slash -P -c localhost:8000
```

Al lanzaro se conecta al servidor y nos lo notifica. Ahora todo lo que escribamos en el cliente aparecerá en el servidor, y vice-versa. En esta animación lo vemos en funcionamiento:



En este **pantallazo** se muestra toda la sesión de pruebas:



Probando la API para Websockets del W3C

Ahora implementaremos nuestros **propios clientes** y **servidores de websockets** para familiarzarnos con la **API**. Utilizaremos el paquete websocket. Lo primero que hacemos es **instalarlo**:

```
npm i websocket
```

También utilizaremos el paquete colors para darle un poquito de color a la vida. Recuerda que se instala así:

```
npm i colors
```

Ejemplo 3: Servidor Websockets de Eco

Implementaremos nuestra propia versión del servidor Websokets de Eco. Necesitamos crear primero un **servidor HTTP**, utilizando el **módulo http**, pero no hace falta implementar ningún recurso: no lo usaremos como servidor web. Cualquier petición devolverá el **error 404** y una **página en blanco**

Primero importamos el objeto `server` del paquete `websocket`:

```
const WebSocketServer = require('websocket').server;
```

Creamos el servidor http como ya sabemos (y lo asignamos al objeto server). Ahora creamos el servidor de Websockets, pasándole como argumento el servidor http:

```
const wsServer = new WebSocketServer({httpServer: server});
```

Establecemos la función de retrollamada para la recepción de conexiones. Cada vez que se establezca una conexión válida se ejecutará nuestra función:

```
//-- Conexión al websocket
wsServer.on('request', (req) => {
    console.log("Conexión establecida");
    //....
});
```

Dentro de esta **función de retrollamada** configuramos a su vez **dos retrollamadas** más: una asociada al evento `message` que se produce cada vez que se recibe un **mensaje nuevo** y otra al evento `close` cuando se reciba la **trama de cierre** de la conexión del websocket

```
    //-- Retrollamada de mensaje recibido
    connection.on('message', (message) => {
        console.log("MENSAGE RECIBIDO");
        //...
    };

    //-- Retrollamada de cierre de conexión
    connection.on('close', (reasonCode, description) => {
        console.log("Conexión cerrada");
        //....
    });
```

El mensaje recibido se encuentra en el objeto `message`. Para saber de qué tipo es utilizamos la propiedad `type` tiene estos valores:

- `message.utf8` : Se trata de un mensaje de texto
- `message.binary` : Es un mensaje en binario

Según que el mensaje sea de **texto** o **binario** utilizamos los siguientes atributos para obtener los datos:

- `message.utf8Data` : Datos como cadena de texto
- `message.binaryData` : Datos binarios

Por último, para **enviar un mensaje** al cliente usamos los siguientes **métodos**:

- `connection.sendUTF(data)` : Envío de un mensaje de texto
- `sendBytes(message.binaryData)` : Envío de un mensaje en binario;

El **código completo** se muestra a continuación. Está en el **fichero**: 03-Websocket-API-server.js. Sólo se procesan los **mensajes de texto** (por simplicidad)

```
const WebSocketServer = require('websocket').server;
const http = require('http');

//-- Vamos a dar un poquito de color...
const colors = require('colors');

const PUERTO = 8080;

//-- Crear servidor. No tiene recursos implementados
//-- (siempre devuelve error)
const server = http.createServer( (req, res) => {
    console.log('Solicitud http: ' + req.url);
    res.writeHead(404);
    res.end();
});

//-- Crear el servidor de websockets, asociado al servidor http
const wsServer = new WebSocketServer({httpServer: server});

//-- Conexión al websocket
wsServer.on('request', (req) => {
    console.log("Conexión establecida".yellow);

    //-- Esperar a recibir mensajes
    const connection = req.accept();

    //-- Retrollamada de mensaje recibido
    connection.on('message', (message) => {
        console.log("MENSAGE RECIBIDO");
```

```
            console.log("  Tipo de mensaje: " + message.type);
            if (message.type === 'utf8') {
                console.log("  Mensaje: " + message.utf8Data.green);

                //-- Enviar el eco
                connection.sendUTF(message.utf8Data);
            }
        });

        //-- Retrollamada de cierre de conexión
        connection.on('close', (reasonCode, description) => {
            console.log("Conexión cerrada".yellow + ". Código: " + reasonCode + ". Ra
        });

    });

    //-- Lanzar el servidor
    //-- ¡Que empiecen los juegos de los WebSockets!
    server.listen(PUERTO);
    console.log("Escuchando en puerto: " + PUERTO);
```

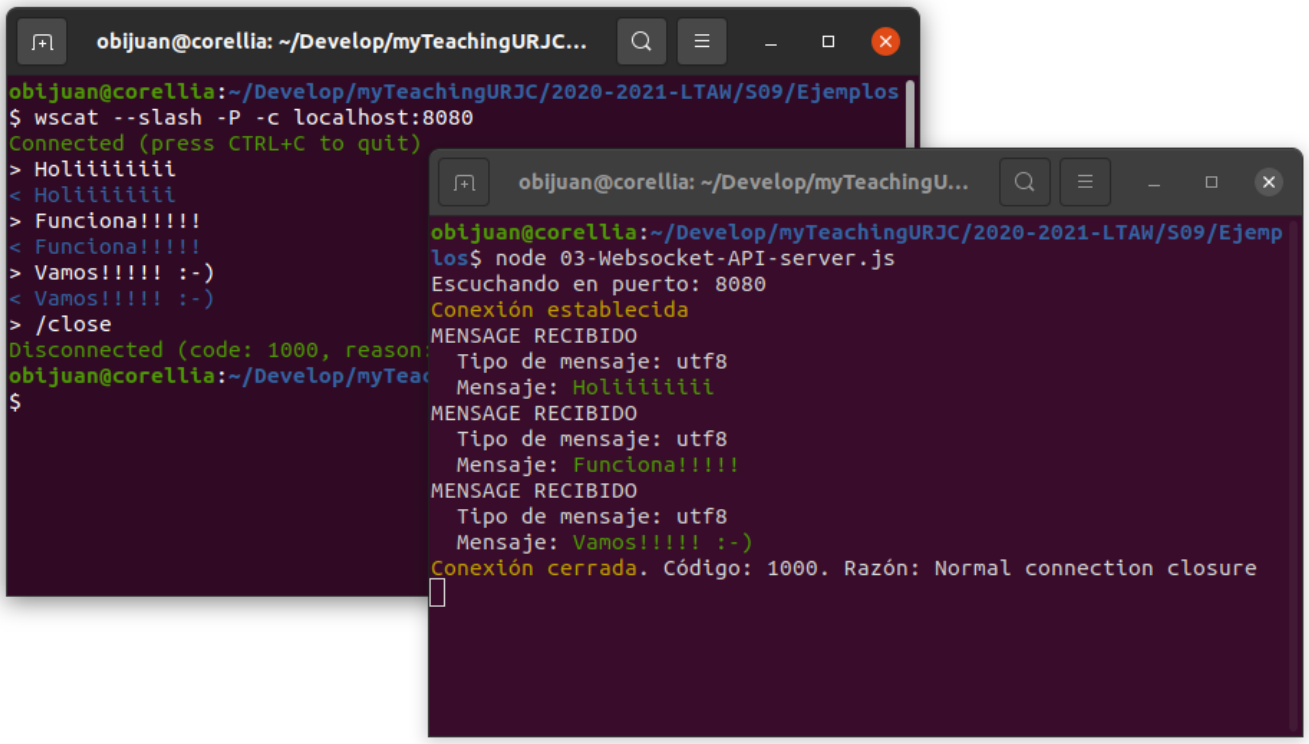
Lanzamos primero el **servidor**, que se queda a la espera de recibir conexiones

```
$ node 03-Websocket-API-server.js
Escuchando en puerto: 8080
```

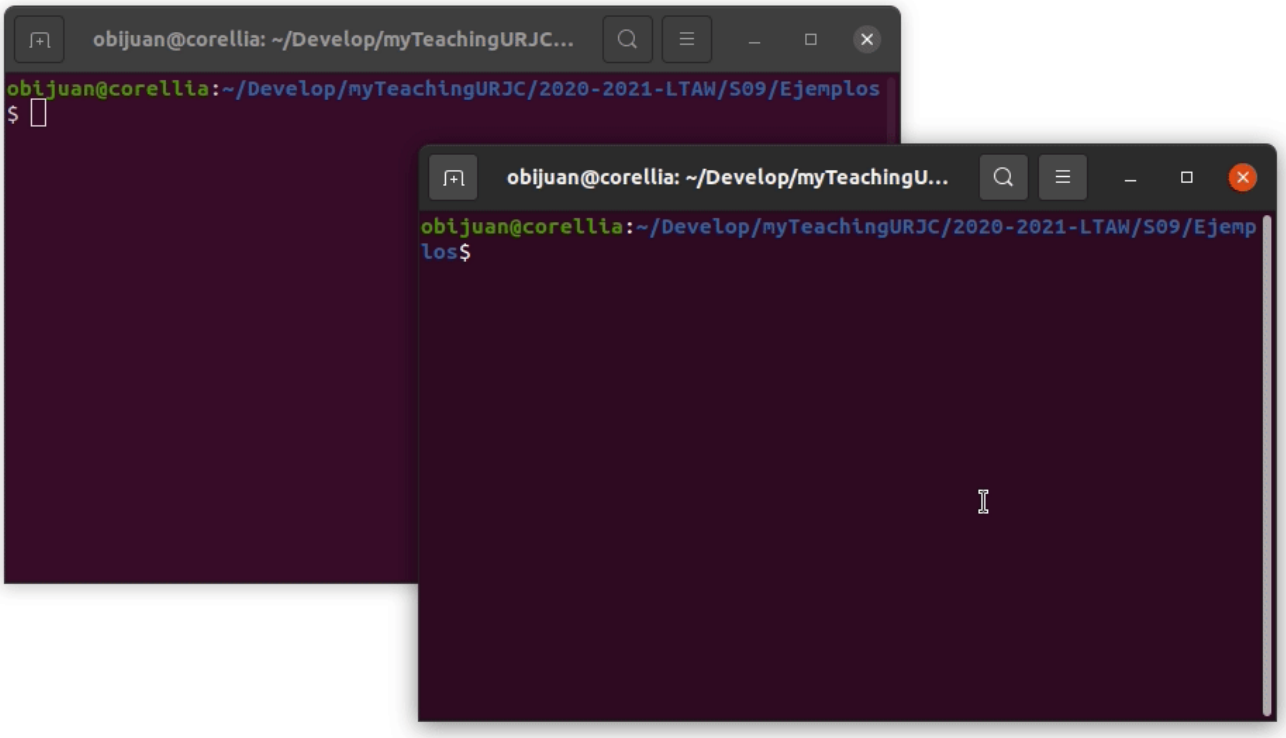
Y luego el **cliente** con wscat:

```
$ wscat --slash -P -c localhost:8080
Connected (press CTRL+C to quit)
>
```

Al lanzarlo vemos en el servidor el mensaje de **Conexión establecida**. Escribimos unos **mensajes de prueba** y terminamos escribiendo `/close`



En esta **animación** lo vemos en funcionamiento



Ejemplo 4: Cliente en Node.js (línea de comandos)

Vamos a crear nuestro propio **cliente** en la **línea de comandos** que se conectará al **Servidor de Eco**, utilizando la **API Websocket** estandarizada por el **W3C+*. El módulo websocket de Node tiene varias APIs implementadas. Para usar la del W3C hay que importar el objeto `w3cwebsocket` :

```
const W3CWebSocket = require('websocket').w3cwebsocket;
```

En este ejemplo se envía un mensaje inicial al establecerse la conexión y luego se envían mensajes periódicamente cada 2 segundos. Cada mensaje se numera utilizando un contador que empieza a contar desde 1

Este es el código completo, que está en el fichero: **04-Websocket-API-client.js**:

```
//-- Módulo para crear clientes con la API websockets del W3C
const W3CWebSocket = require('websocket').w3cwebsocket;

const colors = require('colors');

//-- Crear el objeto cliente con la URL a la que conectarse
const client = new W3CWebSocket('ws://localhost:8080/');

//-- Función de retrollamada al establecerse la conexión
client.onopen = () => {
    console.log('CLIENTE. Conectado al Servidor'.yellow);

    const MSG = "Holiiii-";
    let cont = 1;

    //-- Enviar mensaje inicial
    client.send("Mensaje inicial");

    //-- Enviar mensajes cada 2 segundos...
    setInterval(()=>{
      //-- Solo enviamos el mensaje si la conexión está abierta
      if (client.readyState == client.OPEN) {
        console.log("Enviado: " + (MSG + cont).blue)
        client.send(MSG + cont);
        cont = cont + 1;
      }
    }, 2000);

};

//-- Retrollamada de mensaje recibido
client.onmessage = (e) => {

    //-- Solo imprimir los mensajes de texto
    if (typeof e.data === 'string') {
        console.log("Mensaje recibido: " + e.data.green);
    }
};

//-- Retrollamada de conexión terminada
client.onclose = () => {
    console.log('CLIENTE: Conexión terminada'.yellow);
```

```
};
```

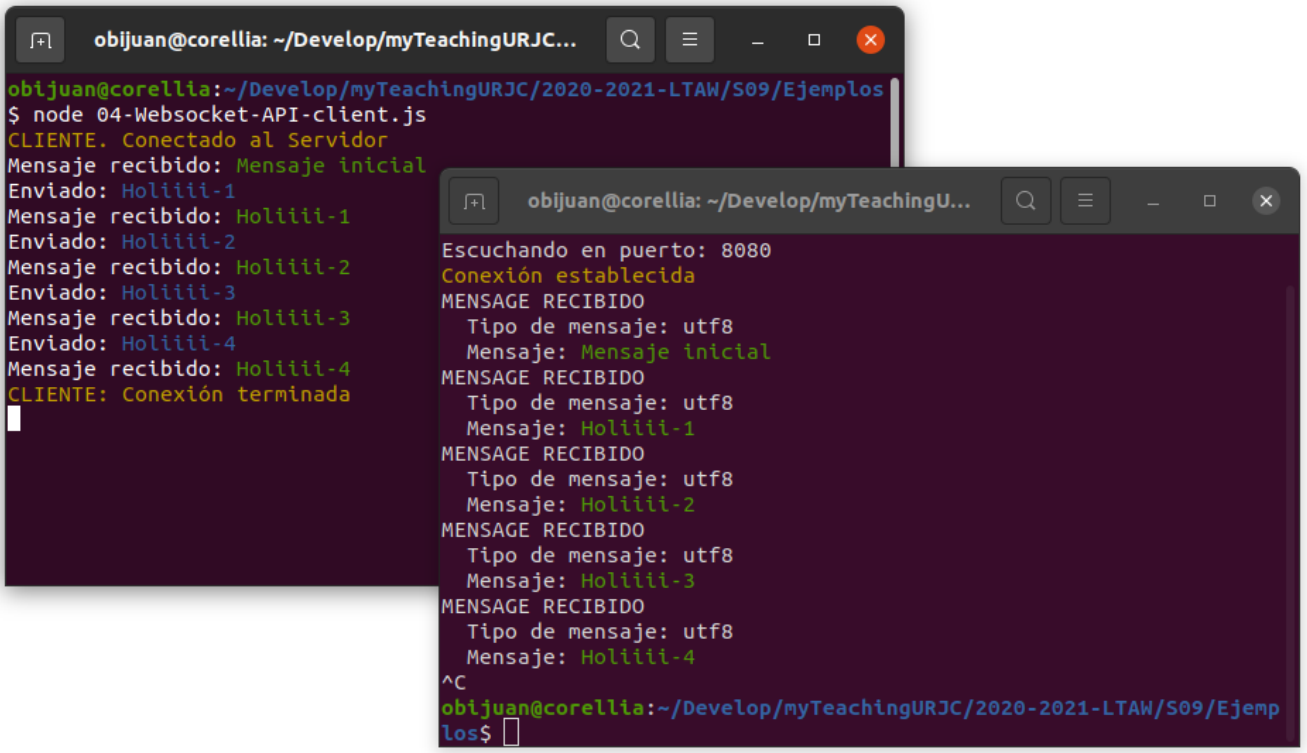
Lanzamos primero el **servidor de Eco**, que queda a la escucha en el **puerto 8080**

```
$ node 03-Websocket-API-server.js
Escuchando en puerto: 8080
```

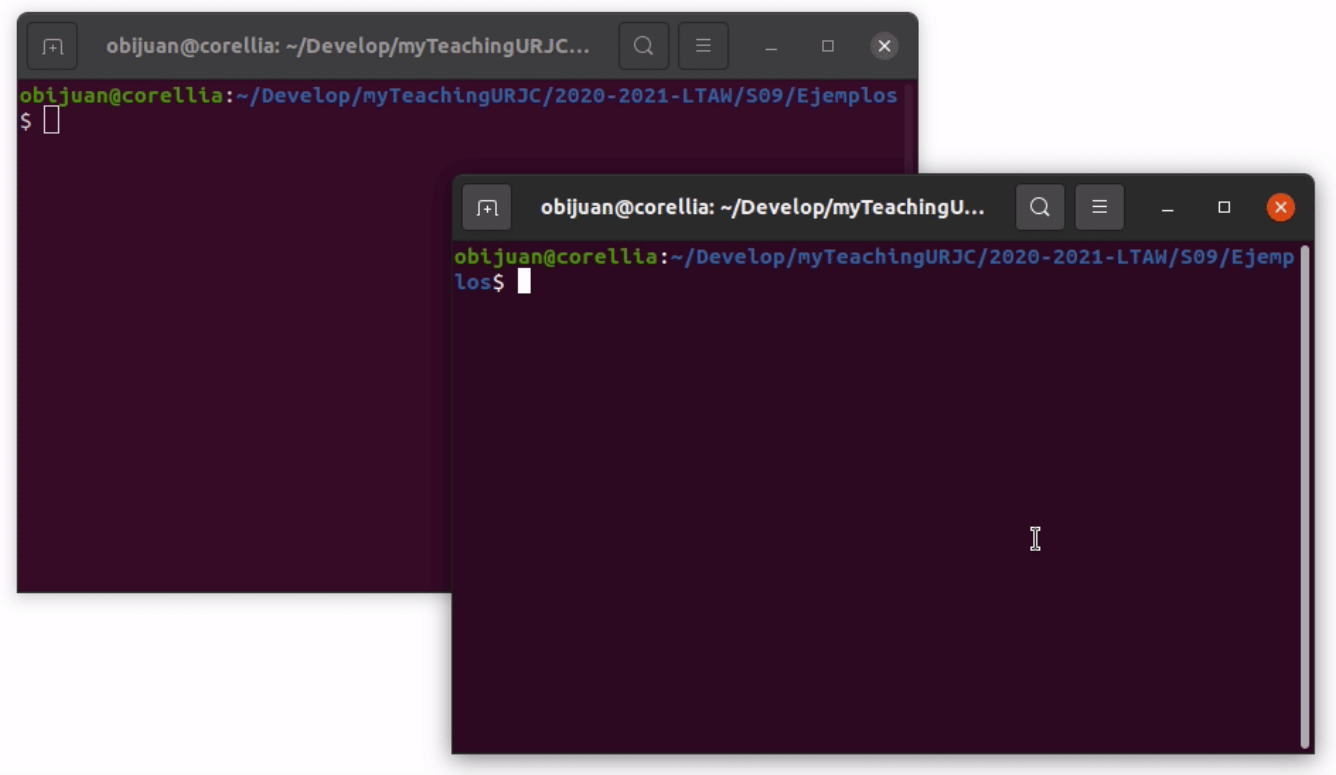
Y ahora ejecutamos el cliente, que nada más conectar envía un mensaje inicial del que el servidor hace ECO:

```
$ node 04-Websocket-API-client.js
CLIENTE. Conectado al Servidor
Mensaje recibido: Mensaje inicial
```

En este **pantallazo** lo que ocurre. Tras algunos segundos se **cierra el servidor** con **Control-C**



En esta **animación** lo vemos en funcionamiento:



Ejemplo 5: Servidor Web + Websockets con express

Vamos a hacer una primera **aplicación WEB** que una ambos mundo: que sirva **páginas web** utilizando `express` y que tenga también un **Websocket** al que se puedan **conectar clientes**

El **servidor HTTP** tiene un comportamiento dual: por un lado **recibir peticiones HTTP**, que deben ser procesadas por **express**. Por otro lado debe recibir **peticiones de conexiónes al Websocket**, que se gestionan por el **servidor de Websockets**

Primero se crea una **aplicación Web vacía** con express. Después se crea el **objeto http server** al que se le pasa como parámetro la aplicación de express. Por último se crea el objeto servidor **wsServer** pasándole como argumento el servidor http:

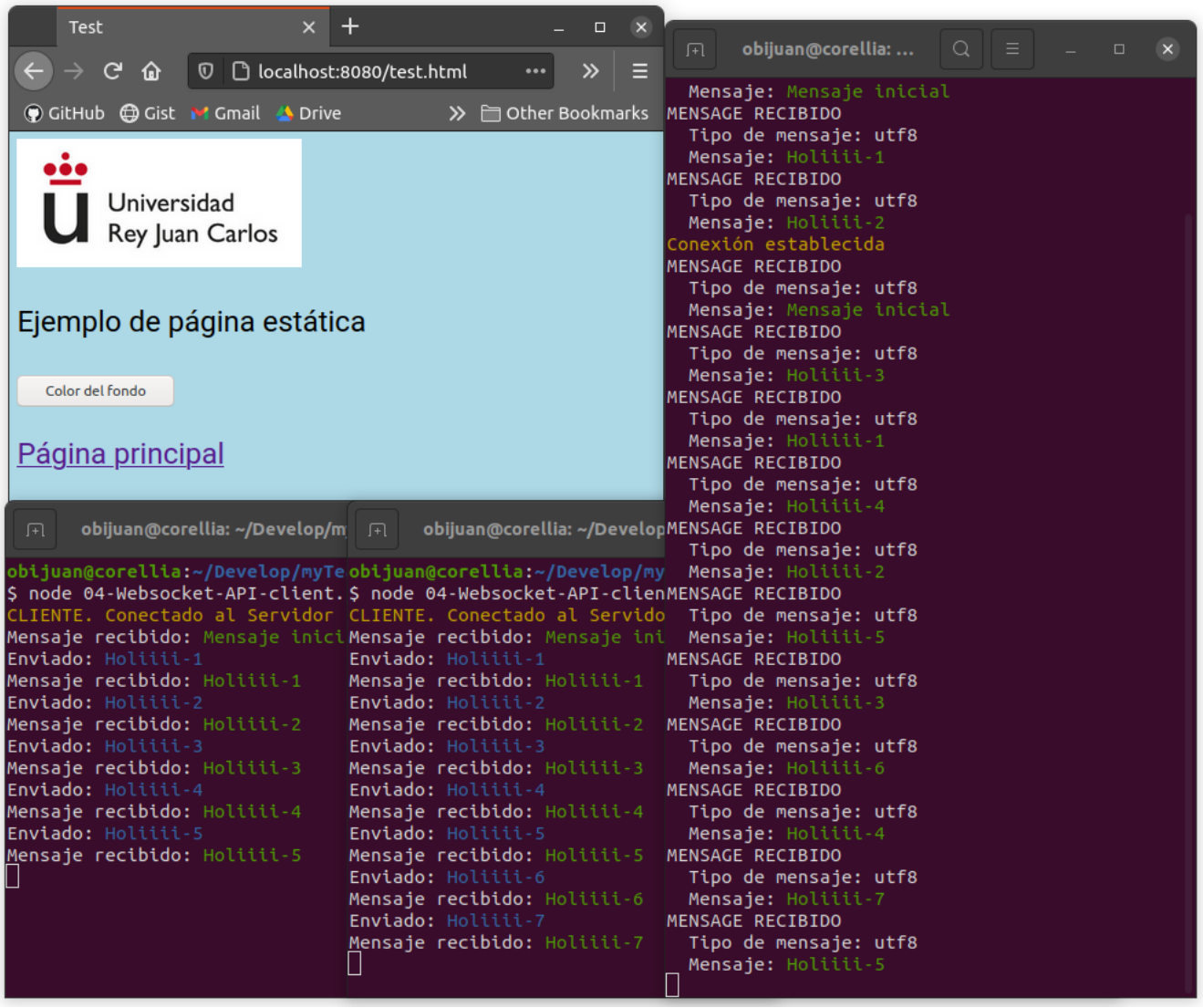
```
//-- Crear una aplicación web vacia
const app = express();

//-- Asociar el servidor http con la app de express
const server = http.Server(app);

//-- Crear el servidor de websockets, asociado al servidor http
const wsServer = new WebSocketServer({httpServer: server});
```

Y ahora ya sólo hay que establecer las **funciones de retrollamada** asociadas a **express** y al **servidor de WebSockets**. El servidor se debe lanzar como siempre (y NO con app.listen()):

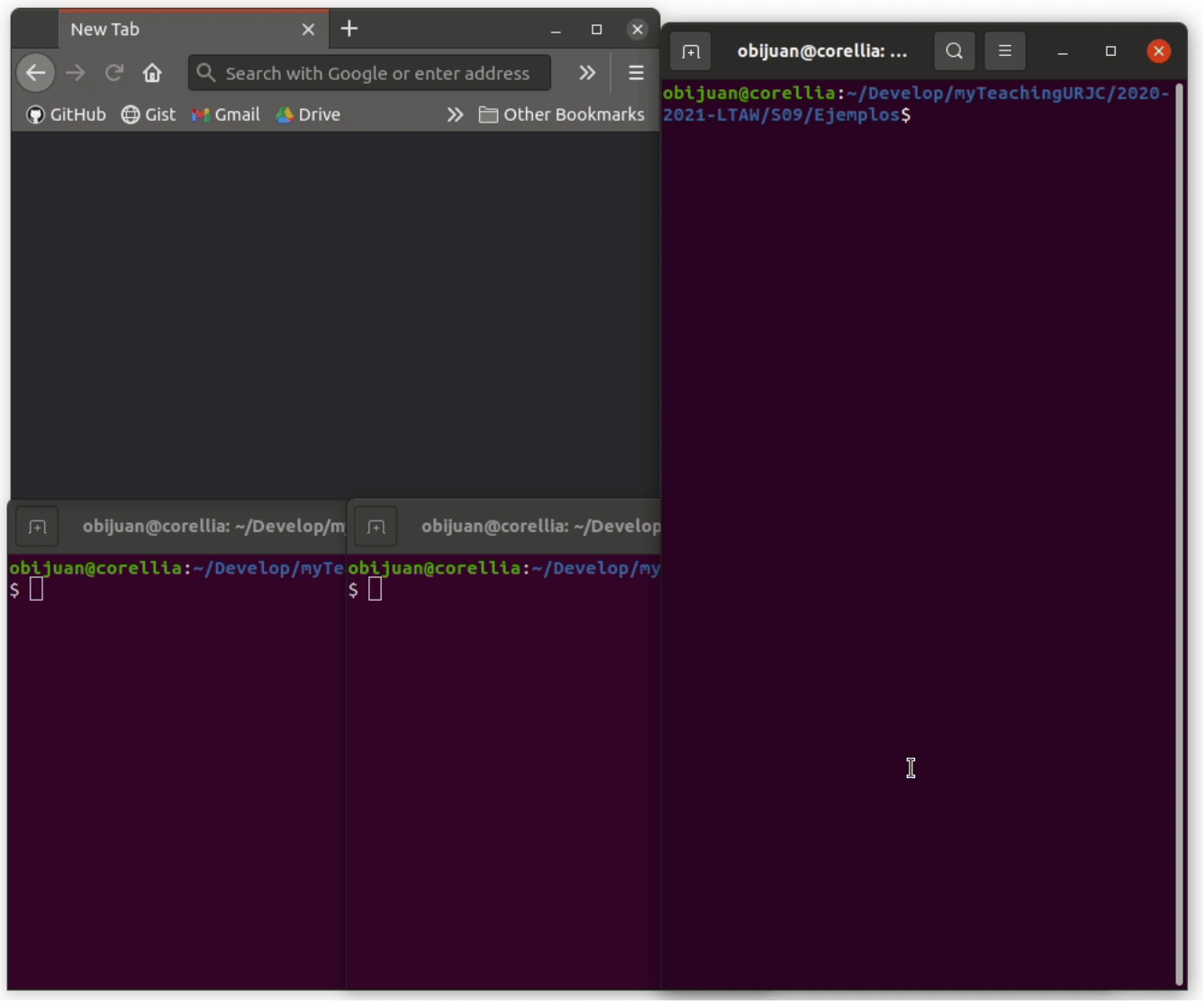
```
//-- Lanzar el servidor HTTP
//-- ¡Que empiecen los juegos de los WebSockets!
```

```
server.listen(PUERTO);
console.log("Escuchando en puerto: " + PUERTO);
```

En este **pantallazo** lo vemos en funcionamieto. En el navegador se está accediendo a la página **/test.html**, que incluye código javascript, css y una imagen. Desde dos terminales independientes se tienen **dos clientes** conectados al websocket, enviando mensajes cada 2 segundos



En esta **animación** vemos su evolución



Ejemplo 6: Cliente de Websockets en el Navegador

Ahora haremos un **cliente Javascript** que se ejecute en el **navegador** y que **envíe mensajes** por un **websocket** al servidor. Al arrancar se envía un **mensaje inicial**. Cada vez que pulsamos un botón se envía un mensaje que lleva asociado un contador, que se va incrementando. Todos los mensajes recibidos del servidor se muestran en el navegador

En el **fichero Ej-06.html** está definido el HTML del ejemplo, que situaremos dentro del directori **public**

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="Ej-06.js" defer></script>
    <title>Ejemplo 6</title>
</head>
<body>
    <h2>Ejemplo 6: Prueba de Websockets</h2>
```

```
    <input id="button" type="button" value="Enviar mensaje por Websocket">
    <h3>Mensajes recibidos</h3>
    <p id="display"></p>
</body>
</html>
```

El javascript que se ejecuta en el navegador está en el fichero Ej-06.js, también en el directorio **public**:

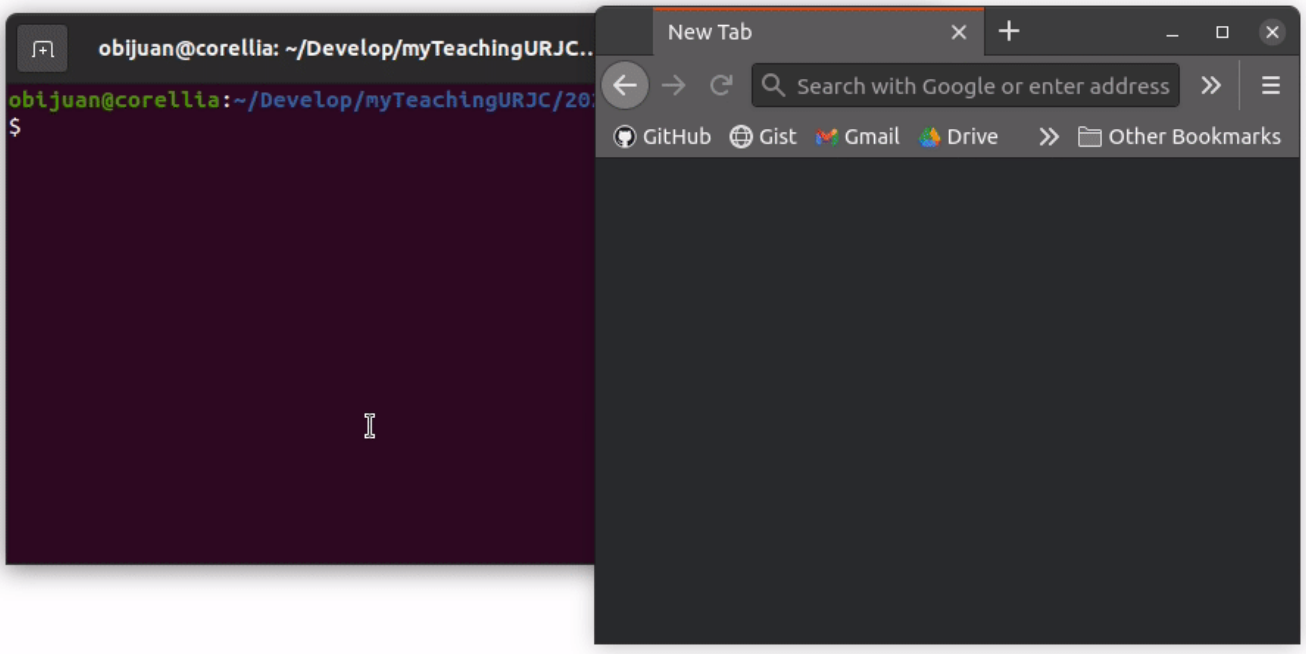
```
//-- Elementos del interfaz
const button = document.getElementById("button");
const display = document.getElementById("display");

//-- Crear el Websocket
const websocket = new WebSocket("ws://localhost:8080");

let contador = 1;

//-- Enviar mensaje inicial al establecerse la conexión
websocket.onopen = () => {
    console.log("Conexión establecida!")

    //-- Enviar mensaje inicial
    websocket.send("Mensaje inicial del Cliente!!!");
}

websocket.onclose = () => {
    console.log("Conexión cerrada!");
}

//-- Mensaje recibido!
websocket.onmessage = (e) => {
  display.innerHTML += '<p style="color: blue">' + e.data + '</p>';
}

//-- Al apretar el botón se envía un mensaje al servidor
button.onclick = () => {
  websocket.send("Holiii-" + contador);
  contador += 1;
}
```

Primero lanzamos el servidor, que es el mismo del ejemplo 5. Se queda a la escucha

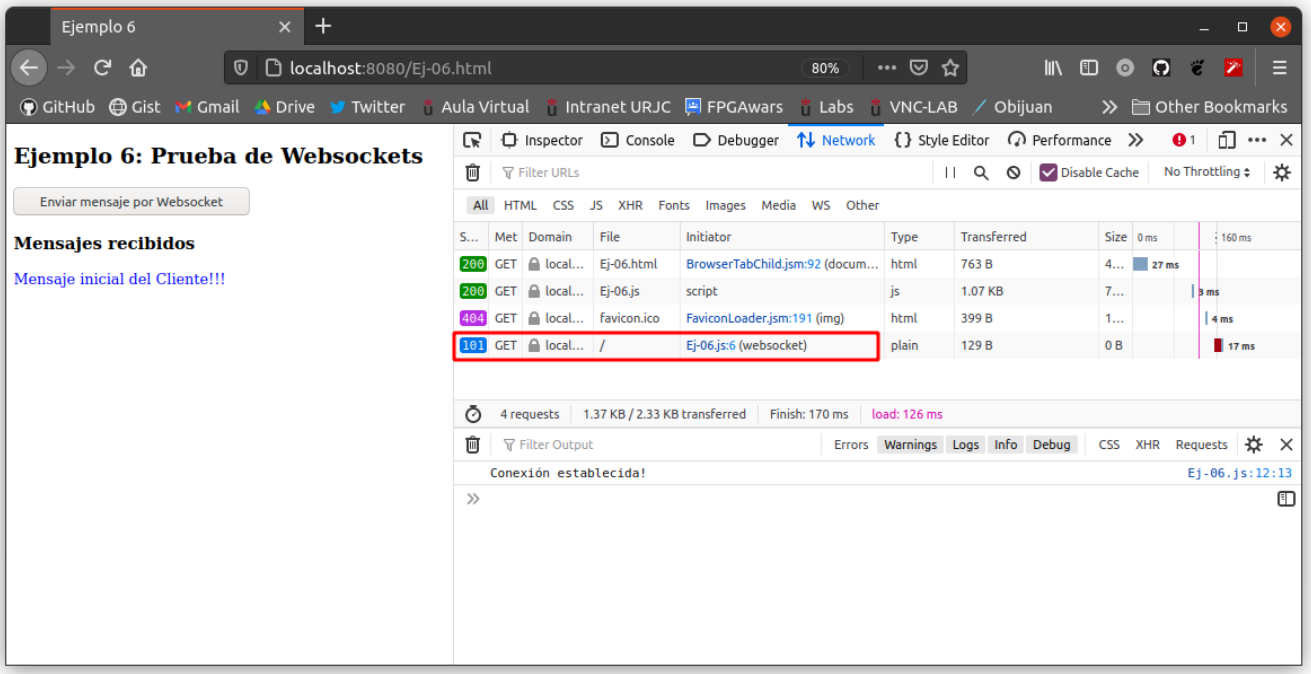
```
$ node 05-Websocket-API-server.js
Escuchando en puerto: 8080
```

Ahora desde el navegador nos conectamos a la URL: `localhost:8080/Ej-06.html` . Se envía el mensaje inicial, y el servidor hace eco. Apretamos el botón 3 veces, para enviar tres mensajes adicionales. Esto es lo que vemos:

En esta **animación** lo vemos en funcionamiento:



Si abrimos la pestaña **Network** de las herramienas del desarrollador para ver todas las peticiones realizadas, observamos que efectivamente ha habido una petición realizada desde Ej-06.js para conectarse a los websockets. El código devuelto es **101**, que indica que la conexión es correcta



Sin embargo, ya **NO** veremos las peticiones que se realizan al pulsar el botón, ya que van por la **nueva conexión TCP** creada y que ya no está relacioanda con el protocolo HTTP (el navegador NO la ve)

Biblioteca Socket.io

Para trabajar con los websockets podemos utilizar el API estandarizada. Pero existen muchas **bibliotecas** que nos facilitan más la labor, tanto en la **implementación** de los **servidores** como de los **clientes**. Una de ellas es la Socket.IO, que es la que usaremos en las prácticas para **implementar un Chat**

Instalación

La instalación es muy fácil. Sólo hay ejecutar el siguiente **comando**:

```
npm i socket.io
```

Socket.io API

Para trabajar con **Socket.io** tenemos que crear primero el objeto **io**:

- En el lado del **servidor** (node.js):

```
//-- Importar el módulo socket.io
const socket = require('socket.io');

//-- Crear el servidor de websockets asociado a un servidor http (previamente cre
```

```
const io = socket(server);
```

- En el lado del **cliente** (Navegador):

```
//-- Crear un websocket. Se establece la conexión con el servidor
const socket = io();
```

Desde el HTML, previamente, debemos de haber cargado la librería socket.io (Esta biblioteca se ha instalado cuando hemos hecho el `npm i socket.io`)

```
<script src="/socket.io/socket.io.js"></script>
```

Los **eventos disponibles** son:

- `connect` : Nueva conexión establecida
- `disconnect` : Conexión terminada
- `message` : Mensaje recibido (enviado con socket.send())
- `xxxx` : Cualquier otra cadena se usa para recibir mensajes emitidos con socket.emit()

Los **Métodos** disponibles son:

- `socket.send(data)` : Enviar un mensaje genérico
- `socket.emit(Nombre_evento, data)` : Enviar datos asociados al evento "Nombre_evento" (es una cadena)

Ejemplo 7: Servidor de Eco y cliente en navegador

Este ejemplo funciona igual que el **Ejemplo 6** pero utilizando la biblioteca **Socket.io**. Sin embargo, como **Socket.io** no está incluida en el navegador, tendremos que indicar explícitamente en el HTML que se cargue

- Fichero **Ej-07.html** (en el directorio public)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Ejemplo 7</title>

  <!-- Cargar la biblioteca Socket.IO. Nos la debe entregar el servidor -->
  <script src="/socket.io/socket.io.js"></script>
```

```
  <!-- Cliente que usa los Sockets.io-->
  <script src="Ej-07-client.js" defer></script>
</head>

<body>
  <h2>Ejemplo 7: Prueba de Socket.io</h2>
    <input id="button" type="button" value="Enviar mensaje por Websocket">
    <h3>Mensajes recibidos</h3>
    <p id="display"></p>
</body>
</html>
```

- Fichero: **Ej-07-client.js** (en el directorio public)

Es el cliente javascript que se carga en el navegador, y que se lo entrega al navegador el propio servidor

```
//-- Elementos del interfaz
const button = document.getElementById("button");
const display = document.getElementById("display");

//-- Crear un websocket. Se establece la conexión con el servidor
const socket = io();

let contador = 1;

socket.on("connect", () => {
  //-- Enviar mensaje inicial
  socket.send("Mensaje inicial del Cliente!!!");
});

socket.on("disconnect", ()=> {
  display.innerHTML="¡¡DESCONECTADO!!"
})

socket.on("message", (msg)=>{
  display.innerHTML += '<p style="color:blue">' + msg + '</p>';
});

//-- Al apretar el botón se envía un mensaje al servidor
button.onclick = () => {
  socket.send("Holiii-" + contador);
  contador += 1;
}
```

- Fichero: **Ej-07-server.js**. Es el servidor de eco

```
//-- Cargar las dependencias
const socket = require('socket.io');
```

```
const http = require('http');
const express = require('express');
const colors = require('colors');

const PUERTO = 8080;

//-- Crear una nueva aplciacion web
const app = express();

//-- Crear un servidor, asosiaco a la App de express
const server = http.Server(app);

//-- Crear el servidor de websockets, asociado al servidor http
const io = socket(server);

//-------- PUNTOS DE ENTRADA DE LA APLICACION WEB
//-- Definir el punto de entrada principal de mi aplicación web
app.get('/', (req, res) => {
  res.send('Bienvenido a mi aplicación Web!!!' + '<p><a href="/Ej-07.html">Test</
});

//-- Esto es necesario para que el servidor le envíe al cliente la
//-- biblioteca socket.io para el cliente
app.use('/', express.static(__dirname +'/'));

//-- El directorio publico contiene ficheros estáticos
app.use(express.static('public'));

//------------------- GESTION SOCKETS IO
//-- Evento: Nueva conexion recibida
io.on('connect', (socket) => {

  console.log('** NUEVA CONEXIÓN **'.yellow);

  //-- Evento de desconexión
  socket.on('disconnect', function(){
    console.log('** CONEXIÓN TERMINADA **'.yellow);
  });

  //-- Mensaje recibido: Hacer eco
  socket.on("message", (msg)=> {
    console.log("Mensaje Recibido!: " + msg.blue);

    //-- Hacer eco
    socket.send(msg);
  });

});

//-- Lanzar el servidor HTTP
//-- ¡Que empiecen los juegos de los WebSockets!
server.listen(PUERTO);
```

```
  console.log("Escuchando en puerto: " + PUERTO);
```

Además del directorio **public** donde están las **páginas estáticas** (.html y .js), hay que hacer que el directorio desde el que se lance node también se incluya entre las páginas estáticas. Esto es necesario para que pueda servir la **biblioteca socket.io** al cliente. Por ello hemos puesto esta línea:

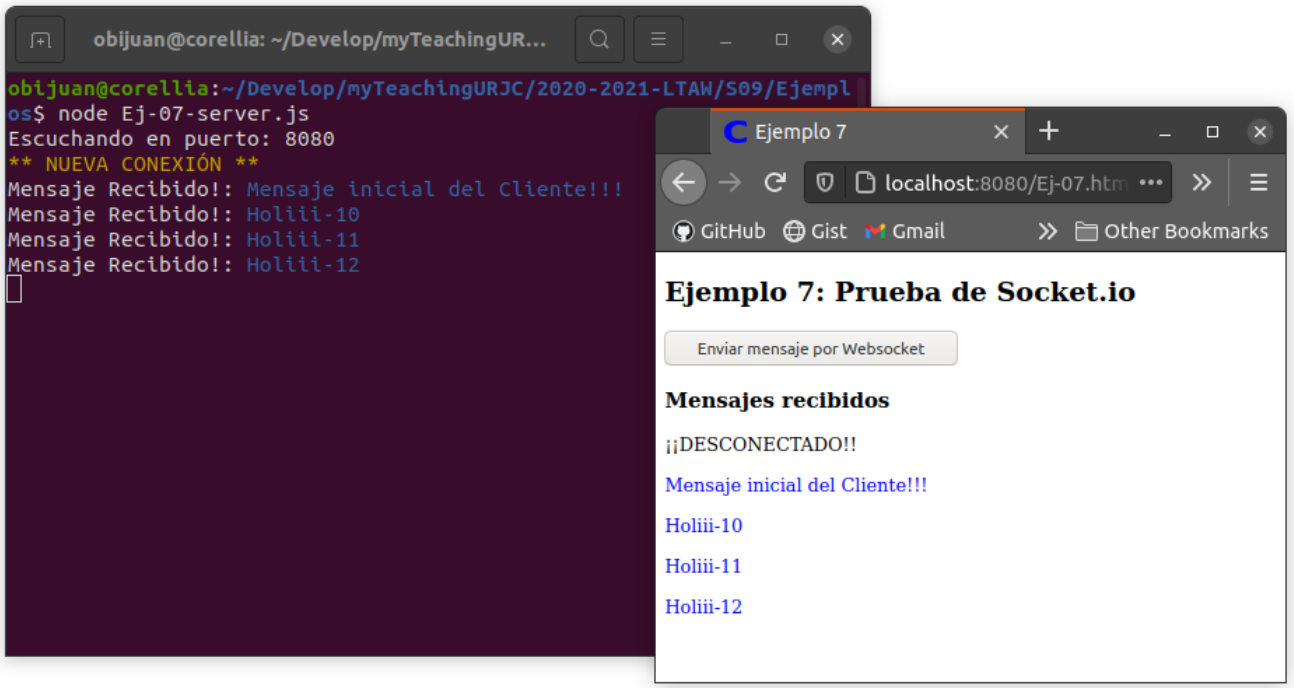
```
  app.use('/', express.static(__dirname +'/'));
```

Para probarlo primero lanzamos el servidor:

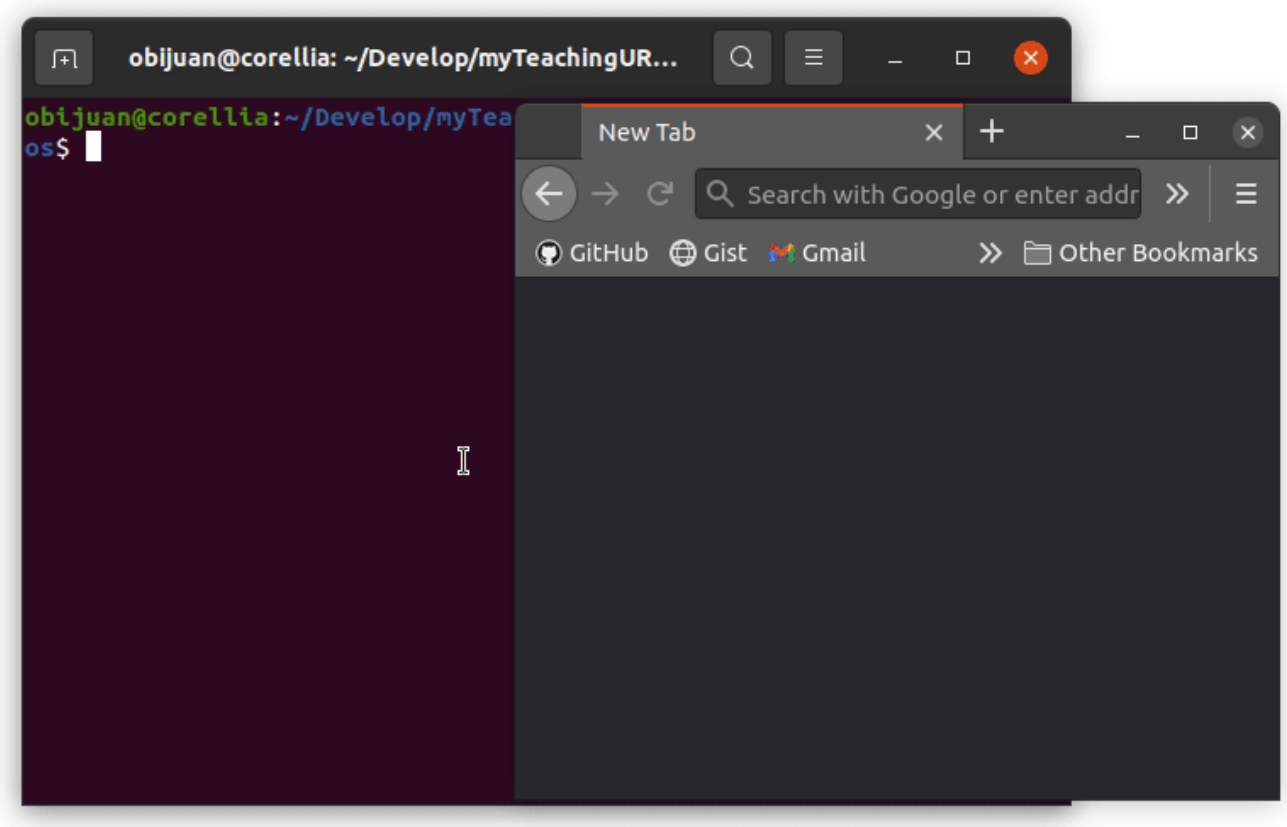
```
  $ node Ej-07-server.js
  Escuchando en puerto: 8080
```

Desde el navegador nos conectamos al recurso `localhost:8080/Ej-07.html` . Se envía el mensaje de bienvenida al servidor y éste hace eco, por lo que lo veremos en azul en el navegador. Cada vez que apretemos el botón enviamos un nuevo mensaje, que nos devolverá el servidor

En este **pantallazo** vemos el resultado:



Y en esta **animación** lo vemos en acción. Aquí sólo estamos utilizando un cliente, pero te puedes conectar desde varios clientes a la vez (otras pestañas, el móvil...)



Ejemplo 8: Mensajes y eventos

La biblioteca Socket.io permite enviar **mensajes asociados** a un **evento**. Esto nos permite tener muchos **canales** independientes de comunicación. Este ejemplo es similar al anterior, pero se emite periódicamente el mensaje "TIC-" + otro contador por el evento denominado 'tic' (podemos poner el nombre que queramos)

El **servidor hace eco** de los **mensajes recibidos**, pero **NO** de los mensajes que le llegan por el evento `tic` . Estos simplemente los sacará por la consola. Los mensajes asociados a un evento se envía utilizando el método `socket.emit(Nombre-evento, mensaje)`

- Fichero **Ej-08.html** (en la carpeta public)

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Ejemplo 8</title>

  <!-- Cargar la biblioteca Socket.IO. Nos la debe entregar el servidor -->
  <script src="/socket.io/socket.io.js"></script>

  <!-- Cliente que usa los Sockets.io-->
  <script src="Ej-08-client.js" defer></script>
</head>
```

```
<body>
  <h2>Ejemplo 8: Prueba de Socket.io</h2>
    <input id="button" type="button" value="Enviar mensaje por Websocket">
    <h3>Mensajes recibidos</h3>
    <p id="display"></p>
</body>
</html>
```

- Fichero: "Ej-08.js" (En la carpeta public)

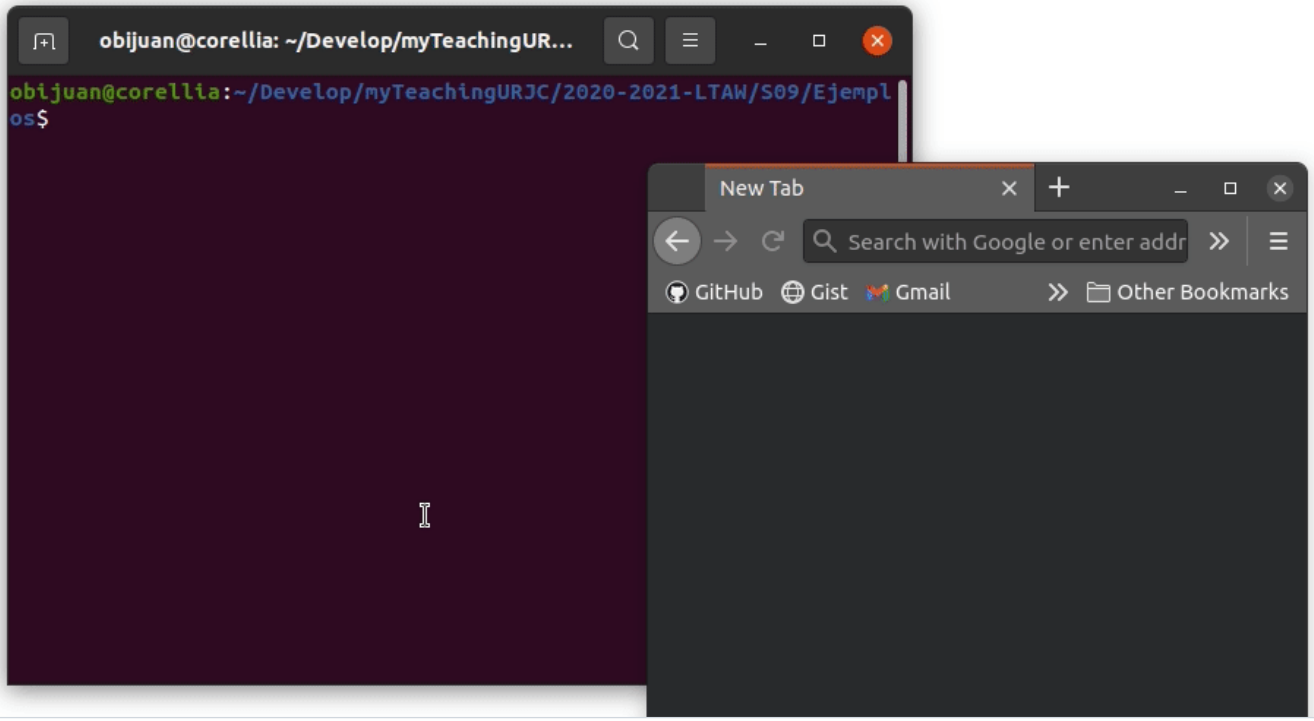
```
//-- Elementos del interfaz
const button = document.getElementById("button");
const display = document.getElementById("display");

//-- Crear un websocket. Se establece la conexión con el servidor
const socket = io();

let contador = 1;

socket.on("connect", () => {
  //-- Enviar mensaje inicial
  socket.send("Mensaje inicial del Cliente!!!");
});

socket.on("disconnect", ()=> {
  display.innerHTML="¡¡DESCONECTADO!!"
})

socket.on("message", (msg)=>{
  display.innerHTML += '<p style="color:blue">' + msg + '</p>';
});

//-- Al apretar el botón se envía un mensaje al servidor
button.onclick = () => {
  socket.send("Holiii-" + contador);
  contador += 1;
}

let tic = 1;

//-- Enviar un mensaje periódico ("TIC")
setInterval(() => {
    socket.emit('tic', "TIC-" + tic);
    tic += 1;
}, 1000);
```

- Fichero **Ej-08-server.js**: Servidor

```
//-- Cargar las dependencias
const socket = require('socket.io');
const http = require('http');
const express = require('express');
const colors = require('colors');

const PUERTO = 8080;

//-- Crear una nueva aplciacion web
const app = express();

//-- Crear un servidor, asosiaco a la App de express
const server = http.Server(app);

//-- Crear el servidor de websockets, asociado al servidor http
const io = socket(server);

//-------- PUNTOS DE ENTRADA DE LA APLICACION WEB
//-- Definir el punto de entrada principal de mi aplicación web
app.get('/', (req, res) => {
  res.send('Bienvenido a mi aplicación Web!!!' + '<p><a href="/Ej-08.html">Test</
});

//-- Esto es necesario para que el servidor le envíe al cliente la
//-- biblioteca socket.io para el cliente
app.use('/', express.static(__dirname +'/'));

//-- El directorio publico contiene ficheros estáticos
app.use(express.static('public'));

//------------------- GESTION SOCKETS IO
//-- Evento: Nueva conexion recibida
io.on('connect', (socket) => {

  console.log('** NUEVA CONEXIÓN **'.yellow);

  //-- Evento de desconexión
  socket.on('disconnect', function(){
    console.log('** CONEXIÓN TERMINADA **'.yellow);
  });

  //-- Mensaje recibido: Hacer eco
  socket.on("message", (msg)=> {
    console.log("Mensaje Recibido!: " + msg.blue);

    //-- Hacer eco
    socket.send(msg);
  });

  //-- Atender mensajes de TIC
  socket.on('tic', (msg)=> {
```

```
    //-- Los mensajes de tic se sacan por la consola,
    //-- pero no se hace eco de ellos
    console.log(msg.green);
  });

});

//-- Lanzar el servidor HTTP
//-- ¡Que empiecen los juegos de los WebSockets!
server.listen(PUERTO);
console.log("Escuchando en puerto: " + PUERTO);
```

En esta **Animación** lo vemos en funcionamiento



▸ **Pages** 26

- Página Inicial

TEORIA

- S0: Presentación
- S1: Lenguajes de marcado. Markdown
- S2: Node.js
- S3: Node.js. Módulos
- S4: XML
- S5: JSON
- S6: Formularios y Cookies

```
<!DOCTYPE html>
```

```
<html>
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Ejemplo 9</title>

  <!-- Cargar la biblioteca Socket.IO. Nos la debe entregar el servidor -->
  <script src="/socket.io/socket.io.js"></script>

  <!-- Cliente que usa los Sockets.io-->
  <script src="Ej-09-client.js" defer></script>
</head>

<body>
  <h2>Ejemplo 9: Mini-chat</h2>
  <input type="text" placeholder="Escribe aquí los mensajes..." autocomplete="off
  <h3>Mensajes recibidos</h3>
  <p id="display"></p>
</body>
</html>
```

- L3: Puesta en marcha
- L4: Módulos

```
//-- Elementos del interfaz
const display = document.getElementById("display");
const msg_entry = document.getElementById("msg_entry");

//-- Crear un websocket. Se establece la conexión con el servidor
const socket = io();


socket.on("message", (msg)=>{
  display.innerHTML += '<p style="color:blue">' + msg + '</p>';
});

//-- Al apretar el botón se envía un mensaje al servidor
msg_entry.onchange = () => {
  if (msg_entry.value)
    socket.send(msg_entry.value);

  //-- Borrar el mensaje actual
  msg_entry.value = "";
}
```

- 2021-03-23-Parcial-1
- 2021-05-19-Parcial-2

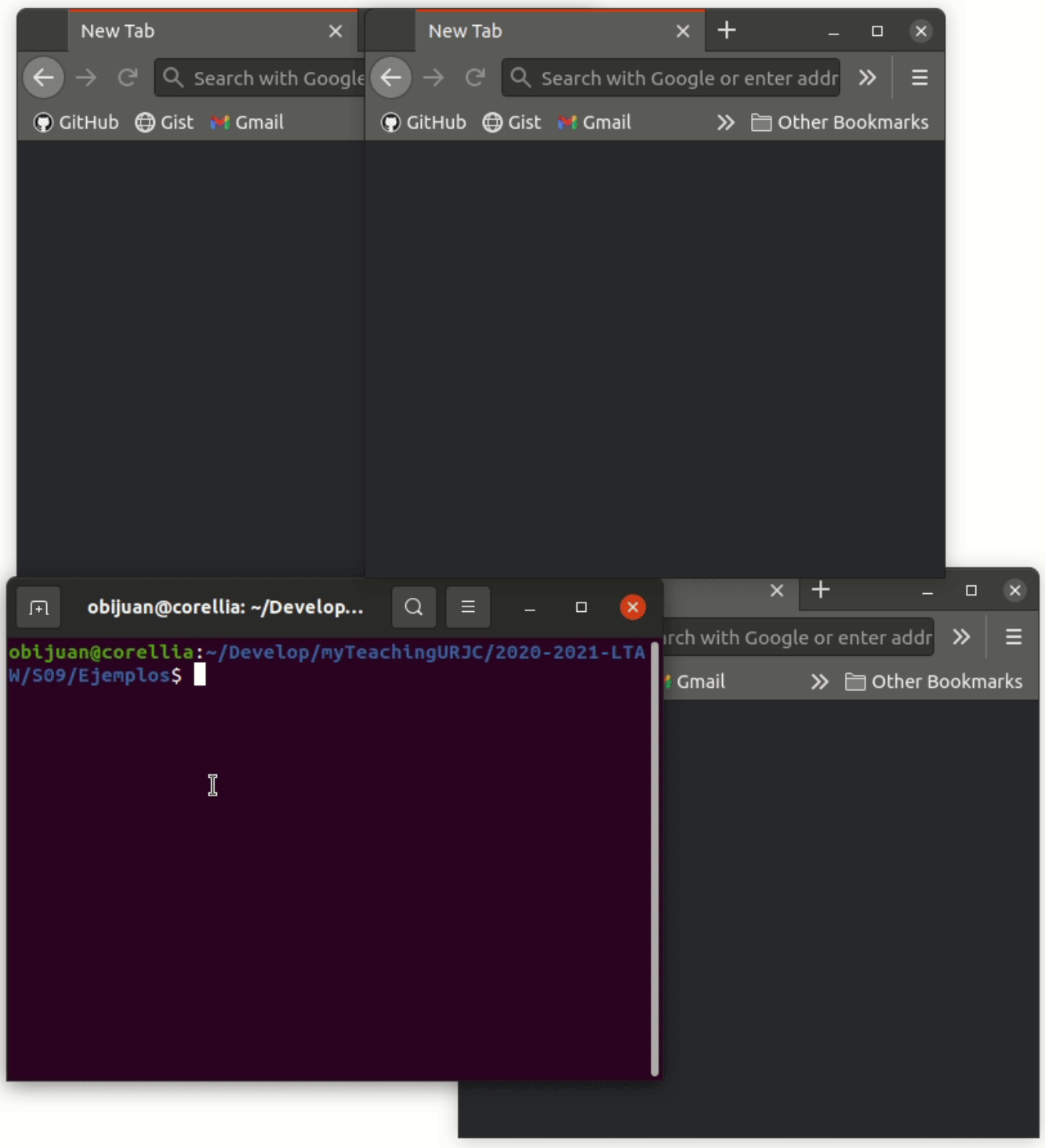
```
//-- Cargar las dependencias
const socket = require('socket.io');
```

https://github.com/myTeachingURJC/2020-2021-LTAW.wiki.git

```
const express = require('express');
const colors = require('colors');

const PUERTO = 8080;

//-- Crear una nueva aplciacion web
const app = express();

//-- Crear un servidor, asosiaco a la App de express
const server = http.Server(app);

//-- Crear el servidor de websockets, asociado al servidor http
const io = socket(server);

//-------- PUNTOS DE ENTRADA DE LA APLICACION WEB
//-- Definir el punto de entrada principal de mi aplicación web
app.get('/', (req, res) => {
  res.send('Bienvenido a mi aplicación Web!!!' + '<p><a href="/Ej-09.html">Test</
});

//-- Esto es necesario para que el servidor le envíe al cliente la
//-- biblioteca socket.io para el cliente
app.use('/', express.static(__dirname +'/'));

//-- El directorio publico contiene ficheros estáticos
app.use(express.static('public'));

//------------------- GESTION SOCKETS IO
//-- Evento: Nueva conexion recibida
io.on('connect', (socket) => {

  console.log('** NUEVA CONEXIÓN **'.yellow);

  //-- Evento de desconexión
  socket.on('disconnect', function(){
    console.log('** CONEXIÓN TERMINADA **'.yellow);
  });

  //-- Mensaje recibido: Reenviarlo a todos los clientes conectados
  socket.on("message", (msg)=> {
    console.log("Mensaje Recibido!: " + msg.blue);

    //-- Reenviarlo a todos los clientes conectados
    io.send(msg);
  });

});

//-- Lanzar el servidor HTTP
//-- ¡Que empiecen los juegos de los WebSockets!
server.listen(PUERTO);
```

```
console.log("Escuchando en puerto: " + PUERTO);
```

En esta **Animación** vemos la **consola del servidor** y **tres clientes** en los navegadores. Lo que escribe uno lo reciben todos los demás



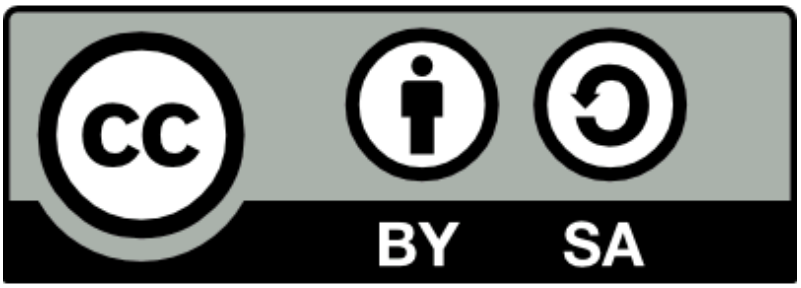
¡¡Tenemos un mini-chat!!

Autor

- Juan González-Gómez (Obijuan)


Créditos

Licencia



Enlaces

- Websocket.org
- Websocket Node
- Universidad Rey Juan Carlos de Madrid
- Escuela Técnica Superior de Ingeniería de Telecomunicaciones (URJC)