WebSockets

Juegos en Red - Grado en Desarrollo de Videojuegos

Ruben Rodríguez
ruben.rodriguez@urjc.es
URJC

Natalia Madrueño
natalia.madrueno@urjc.es
URJC

2025-09-09

Universidad Rey Juan Carlos

DSLAB
DATA SCIENCE LABORATORY

Tabla de contenidos

Introducción a Sockets

Concepto Básico de Socket

Socket: extremo de comunicación bidireccional

- Identificado por **IP + Puerto**
- Permite comunicación entre dos programas
- Bidireccional: ambos pueden enviar y recibir
- Simultáneo: no hay que esperar turnos

Ejemplo:

- Servidor: `212.128.240.50:10000`
- Cliente: `1XX.XXX.XXX.XX:YYYY`

Proceso de Comunicación con Sockets

Flujo típico:

1. **Servidor** escucha en puerto específico
2. **Cliente** inicia petición de conexión
3. **Servidor** acepta la conexión
4. Se establece **canal bidireccional**
5. Ambos pueden enviar/recibir datos

> **Importante**
>
> No es solicitud-respuesta: ambos pueden iniciar comunicación

WebSockets

¿Qué son los WebSockets?

Protocolo de comunicación full-duplex

- Canal permanente entre cliente y servidor
- Comunicación **bidireccional** simultánea
- No necesita abrir nueva conexión cada vez
- Ideal para aplicaciones en tiempo real

> **Nota**
>
> Full-duplex: ambas partes pueden enviar mensajes independientemente

Funcionamiento de WebSockets

Proceso:

1. Cliente solicita conexión al servidor
2. Servidor acepta la conexión
3. Se genera **conexión permanente**
4. Procesos en ambos lados escuchan mensajes

> **Tip**
>
> El servidor puede enviar sin que el cliente pregunte constantemente

WebSockets como Estándar

RFC 6455 sobre socket TCP

Ventajas:

- Usa **puerto 80** (mismo que HTTP)
- Multiplexado: múltiples comunicaciones por puerto
- Evita problemas con firewalls
- Pensado para navegadores y servidores web

> **Nota**
>
> Funciona sobre HTTP pero puede usar protocolo propio

Protocolos WebSocket

Dos esquemas disponibles:

- **ws://** - WebSocket en claro
  - Equivalente a `http://`
- **wss://** - WebSocket seguro (TLS)
  - Equivalente a `https://`

```
1 new WebSocket('ws://example.com/demo');
2 new WebSocket('wss://example.com/demo');
```

Negociación de Conexión

Handshake Inicial

La negociación comienza sobre HTTP

Petición del cliente:

```
1 GET /demo HTTP/1.1
2 Host: example.com
3 Connection: Upgrade
4 Upgrade: WebSocket
5 Origin: http://example.com
```

- `Connection: Upgrade` → queremos cambiar de protocolo
- `Upgrade: WebSocket` → especifica WebSocket
- `Origin` → origen de la petición (seguridad)

Respuesta del Servidor

Si acepta la conexión:

```
1 HTTP/1.1 101 WebSocket Protocol Handshake
2 Upgrade: WebSocket
3 Connection: Upgrade
4 Sec-WebSocket-Origin: http://example.com
5 Sec-WebSocket-Location: ws://example.com/demo
```

- Código **101**: cambio de protocolo aceptado
- `Sec-WebSocket-Location` → nueva ubicación con `ws://`

> **Importante**
>
> Después del handshake, HTTP se transforma en WebSocket

Uso Práctico

La negociación es automática

```
1 // Esto maneja todo el handshake automáticamente
2 const connection = new WebSocket('ws://example.com/demo');
```

Tip

No necesitas implementar manualmente el handshake HTTP

WebSockets en Juegos

Ventajas para Juegos en Red

Por qué usar WebSockets:

- **Tiempo real**
  - Actualizaciones instantáneas
  - Sin demoras
- **Bidireccional**
  - Servidor puede enviar eventos
  - Sin polling constante
- **Eficiente**
  - Una conexión persistente
  - Menos recursos
- **Baja latencia**
  - Ideal para respuestas rápidas
  - Crítico en juegos competitivos

WebSockets vs REST en Juegos

Cuándo usar cada uno:

REST	WebSockets
Registro/autenticación	Movimiento de jugadores
Puntuaciones	Estado del juego
Rankings	Eventos en tiempo real
Perfiles	Chat del juego
Configuración	Notificaciones instantáneas

> **Nota**
>
> Ambas tecnologías se complementan en juegos modernos

API WebSocket en JavaScript

Crear Conexión

Instanciar el objeto WebSocket

```
1  const connection = new WebSocket('ws://IP:PUERTO/RUTA');
```

Componentes:

- `ws://` o `wss://` → protocolo
- `IP` → dirección o dominio del servidor
- `PUERTO` → puerto donde escucha el servidor
- `RUTA` → endpoint específico

Ejemplo:

```
1  const connection = new WebSocket('ws://127.0.0.1:8080/echo');
```

Tip

La conexión se establece automáticamente

Métodos Principales

send() - Enviar datos al servidor

```
// Texto simple
connection.send('Hola servidor');

// JSON (debe convertirse a string)
const data = {
  type: 'player_move',
  x: 150,
  y: 200
};
connection.send(JSON.stringify(data));
```

close() - Cerrar conexión

```
connection.close();
```

Nota

`close()` inicia cierre ordenado de la conexión

Event Listeners - onopen

Se ejecuta cuando la conexión se abre

```
connection.onopen = function() {
  console.log('¡Conectado al servidor!');

  // Enviar información inicial
  connection.send(JSON.stringify({
    type: 'player_join',
    username: 'Jugador1'
  }));
};
```

Tip

Momento perfecto para enviar mensaje inicial

Event Listeners - onmessage

Se ejecuta al recibir mensaje del servidor

```
connection.onmessage = function(msg) {
  console.log("Mensaje recibido: " + msg.data);

  // Si es JSON, parsearlo
  const data = JSON.parse(msg.data);

  // Procesar según tipo
  if (data.type === 'player_position') {
    actualizarPosicionJugador(data.playerId, data.x, data.y);
  } else if (data.type === 'game_over') {
    mostrarPantallaFinJuego(data.winner);
  }
};
```

Importante

El dato real está en `msg.data`

Event Listeners - onerror

Se ejecuta cuando hay un error

```
connection.onerror = function(error) {
  console.log("Error en WebSocket: ", error);

  // Informar al usuario
  alert('No se pudo conectar al servidor. Verifica tu conexión.');
};
```

Situaciones que disparan error:

- No se puede conectar al servidor
- Problema de red
- Servidor rechaza la conexión

Event Listeners - onclose

Se ejecuta cuando la conexión se cierra

```
connection.onclose = function(event) {
  console.log('Desconectado del servidor');
  console.log('Código de cierre:', event.code);

  // Notificar al usuario
  alert('Se ha perdido la conexión');

  // Intentar reconectar después de 3 segundos
  setTimeout(function() {
    connection = new WebSocket('ws://127.0.0.1:8080/echo');
    configurarListeners(connection);
  }, 3000);
};
```

Razones del cierre:

- Llamamos a `connection.close()`
- El servidor cierra la conexión
- Se pierde la conexión de red

Ejemplo Completo - Cliente

```javascript
// Crear conexión
const ws = new WebSocket('ws://127.0.0.1:8080/game');

// Configurar listeners
ws.onopen = function() {
  console.log('Conectado');
  ws.send(JSON.stringify({ type: 'join', name: 'Alice' }));
};

ws.onmessage = function(msg) {
  const data = JSON.parse(msg.data);
  console.log('Recibido:', data);
};

ws.onerror = function(error) {
  console.error('Error:', error);
};

ws.onclose = function() {
  console.log('Desconectado');
};

// Enviar movimiento
function moverJugador(x, y) {
  ws.send(JSON.stringify({ type: 'move', x, y }));
```

Servidor WebSocket con Node.js

Librería ws

Implementación robusta y eficiente

Instalación:

```
npm init -y
npm install ws express
```

Configurar package.json:

```
{
  "name": "websocket-server",
  "version": "1.0.0",
  "type": "module",
  "dependencies": {
    "ws": "^8.0.0",
    "express": "^4.18.0"
  }
}
```

Servidor Básico

```
1  import express from 'express';
2  import { WebSocketServer } from 'ws';
3  import { createServer } from 'http';
4
5  const app = express();
6  const server = createServer(app);
7
8  // Servir archivos estáticos
9  app.use(express.static('public'));
10
11 // Crear servidor WebSocket
12 const wss = new WebSocketServer({ server });
13
14 // Iniciar servidor
15 const PORT = 8080;
16 server.listen(PORT, () => {
17   console.log(`Servidor en http://localhost:${PORT}`);
18 });
```

Estructura del Servidor

Componentes:

1. **Servidor HTTP** (`createServer`)
    - Sirve archivos estáticos
    - Comparte puerto con WebSocket
2. **Express** para archivos estáticos
    - Carpeta `public` para cliente HTML/CSS/JS
3. **WebSocketServer** asociado al servidor HTTP
    - Ambos protocolos en el mismo puerto

Evento connection

Se dispara cuando un cliente se conecta

```
wss.on('connection', (ws) => {
  console.log('Nuevo cliente conectado');

  // ws representa la conexión con este cliente específico
});
```

> **Nota**
>
> Cada cliente tiene su propio objeto `ws`

Recibir Mensajes del Cliente

```
wss.on('connection', (ws) => {
  console.log('Nuevo cliente conectado');

  ws.on('message', (message) => {
    console.log('Mensaje recibido:', message.toString());

    // Si es JSON, parsearlo
    const data = JSON.parse(message.toString());
    console.log('Nombre:', data.nombre);
    console.log('Mensaje:', data.mensaje);
  });
});
```

Importante

Los mensajes llegan como `Buffer`, usar `.toString()`

Enviar Mensajes al Cliente

A un cliente específico:

```
ws.on('message', (message) => {
  const received = message.toString();

  // Enviar respuesta
  ws.send(`Echo: ${received}`);
});
```

Enviar JSON:

```
ws.on('message', (message) => {
  const respuesta = {
    tipo: 'confirmacion',
    timestamp: new Date().toISOString(),
    mensaje: 'Recibido correctamente'
  };

  ws.send(JSON.stringify(respuesta));
});
```

Eventos close y error

Limpiar recursos al desconectar:

```
wss.on('connection', (ws) => {
  console.log('Cliente conectado');

  ws.on('close', () => {
    console.log('Cliente desconectado');
    // Limpiar datos asociados a este cliente
  });

  ws.on('error', (error) => {
    console.error('Error en la conexión:', error);
  });

  ws.on('message', (message) => {
    // Manejar mensajes
  });
});
```

Ejemplo: Servidor Echo

Devuelve cualquier mensaje recibido

```
import express from 'express';
import { WebSocketServer } from 'ws';
import { createServer } from 'http';

const app = express();
const server = createServer(app);

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

const wss = new WebSocketServer({ server });

wss.on('connection', (ws) => {
  console.log('Nuevo cliente');

  ws.on('message', (message) => {
    const data = message.toString();
    console.log('Recibido:', data);

    // Echo: devolver al cliente
    ws.send(`Echo: ${data}`);
  });

  ws.on('close', () => console.log('Cliente desconectado'));
  ws.on('error', (error) => console.error('Error:', error));
});
```

Broadcast: Enviar a Todos

Enviar mensaje a todos los clientes conectados

```
wss.on('connection', (ws) => {
  ws.on('message', (message) => {
    const data = JSON.parse(message.toString());

    // Enviar a todos los clientes
    wss.clients.forEach((client) => {
      if (client.readyState === ws.OPEN) {
        client.send(JSON.stringify(data));
      }
    });
  });
});
```

Tip

Verificar `readyState === ws.OPEN` antes de enviar

Ejemplo: Servidor de Chat

```
import express from 'express';
import { WebSocketServer } from 'ws';
import { createServer } from 'http';

const app = express();
const server = createServer(app);
app.use(express.static('public'));

const wss = new WebSocketServer({ server });

wss.on('connection', (ws) => {
  console.log('Cliente conectado al chat');

  ws.on('message', (message) => {
    try {
      const datos = JSON.parse(message.toString());
      console.log(`${datos.nombre}: ${datos.mensaje}`);

      const respuesta = {
        nombre: datos.nombre,
        mensaje: datos.mensaje,
        timestamp: new Date().toISOString()
      };

      // Broadcast a todos
```

Gestionar Estado de Clientes

Asociar datos a cada conexión

```
const clientes = new Map();

wss.on('connection', (ws) => {
  // Asignar ID único
  const clientId = Date.now();
  clientes.set(ws, { id: clientId, nombre: null });

  console.log(`Cliente ${clientId} conectado`);

  ws.on('message', (message) => {
    const datos = JSON.parse(message.toString());

    // Guardar nombre del cliente
    const clienteInfo = clientes.get(ws);
    clienteInfo.nombre = datos.nombre;

    // Procesar mensaje...
  });

  ws.on('close', () => {
    const clienteInfo = clientes.get(ws);
    console.log(`Cliente ${clienteInfo.id} desconectado`);
    clientes.delete(ws);
  });
});
```

Integración REST + WebSockets

Arquitectura Híbrida

Combinar lo mejor de ambos mundos

```
import express from 'express';
import { WebSocketServer } from 'ws';
import { createServer } from 'http';

const app = express();
const server = createServer(app);

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

// Rutas REST
app.get('/api/status', (req, res) => {
  res.json({
    clientesConectados: wss.clients.size,
    estado: 'activo'
  });
});

// Servidor WebSocket
const wss = new WebSocketServer({ server });

wss.on('connection', (ws) => {
  // Manejar conexiones WebSocket
});

```

División de Responsabilidades

REST para:

- Autenticación
- Registro de usuarios
- Consulta de rankings
- Subida de archivos
- Operaciones puntuales

WebSocket para:

- Movimiento de jugadores
- Estado del juego en tiempo real
- Notificaciones instantáneas
- Chat del juego
- Eventos en vivo

Importante

Ambos protocolos comparten servidor y puerto

Ventajas de Arquitectura Híbrida

Lo mejor de ambos mundos:

- **Simplicidad REST** para operaciones CRUD
- **Tiempo real WebSocket** para interactividad
- **Mismo servidor** simplifica despliegue
- **Mismo puerto** evita problemas de firewall
- **Código organizado** por tipo de comunicación

Resumen

Conceptos Clave de WebSockets

Fundamentos:

- Socket: extremo de comunicación bidireccional
- WebSocket: protocolo full-duplex sobre TCP
- Conexión permanente entre cliente y servidor
- Handshake inicial sobre HTTP (código 101)
- Protocolos `ws://` y `wss://`

Ventajas:

- Comunicación en tiempo real
- Bidireccional y simultánea
- Eficiente (una conexión persistente)
- Baja latencia para juegos

API JavaScript

Objeto WebSocket:

```
1  const ws = new WebSocket('ws://host:port/path');
```

Métodos:

- `send(data)` - enviar al servidor
- `close()` - cerrar conexión

Event Listeners:

- `onopen` - conexión establecida
- `onmessage` - mensaje recibido
- `onerror` - error en conexión
- `onclose` - conexión cerrada

Servidor Node.js

Librería ws:

```
1 import { WebSocketServer } from 'ws';
2 const wss = new WebSocketServer({ server });
```

Eventos del servidor:

- `connection` - nuevo cliente conectado
- `message` - mensaje del cliente
- `close` - cliente desconectado
- `error` - error en conexión

Broadcast:

```
1 wss.clients.forEach(client => {
2   if (client.readyState === ws.OPEN) {
3     client.send(data);
4   }
5 });
```

REST vs WebSockets

Cuándo usar cada tecnología:

Criterio	REST	WebSockets
Frecuencia	Ocasional	Continua
Latencia	Aceptable	Crítica
Dirección	Cliente inicia	Ambos inician
Casos de uso	CRUD, auth	Tiempo real, chat

Tip

En juegos modernos: ambas tecnologías se complementan