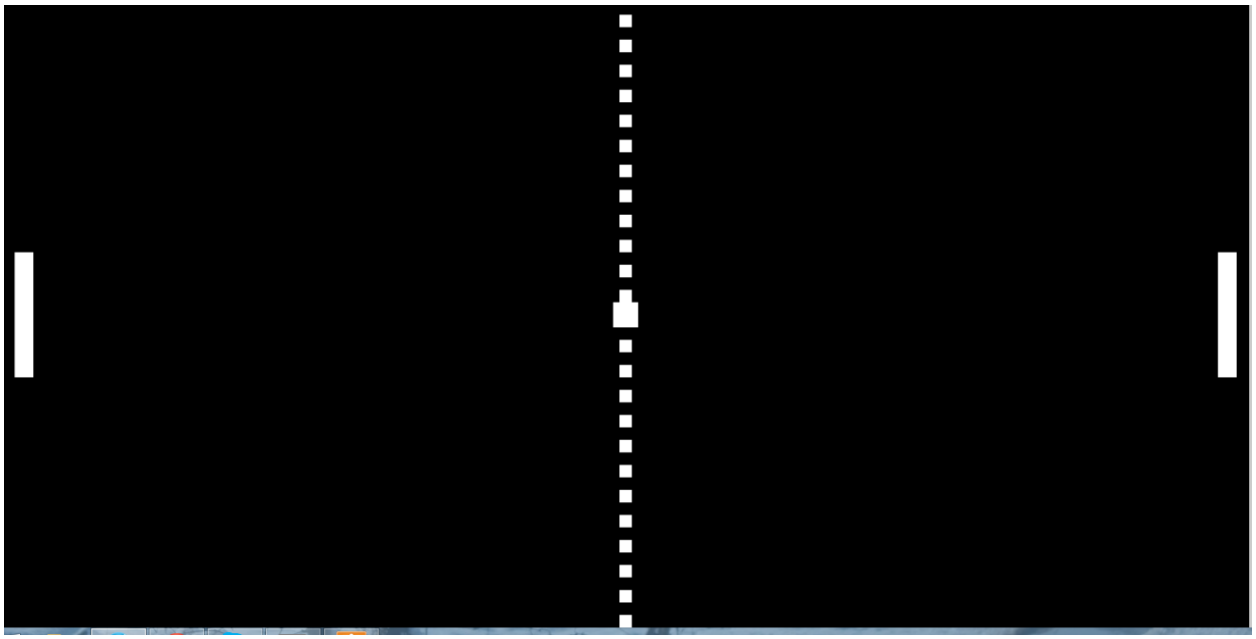
DOCUMENTACIÓN TÉCNICA - TALLER NODE.JS
PONG CON NODE.JS EN 2 HORAS



Francisco García Díaz
Jesús Rodríguez Pérez

Introducción

Se realizará un taller de node.js en el que se guiará paso a paso al alumno en la realización de un pong multijugador en tiempo real a través de la red.
En el taller se explicarán los conceptos básicos para comprender el funcionamiento de las tecnologías utilizadas, principalmente node.js y la etiqueta canvas de HTML5

Breve descripción del trabajo

En este taller cada alumno realizará los pasos indicados en su ordenador, estos pasos principalmente serán la creación del servidor en node.js y la creación del juego en HTML5. El proyecto completo el cual incluirá la presentación paso a paso se podrá descargar de github.

Lenguajes de Programación/Tecnologías utilizados

Lenguajes:

HTML5, JavaScript.

Tecnologías y frameworks:

- Node.js
  Es una plataforma para construir aplicaciones de red escalables usando javascript.
- Jade
  Sistema de plantillas para node.js. Nos permite escribir pa rápidamente codigo que después se renderiza como HTML gracias a que no tenemos que usar cierres de etiquetas y usamos la indentacion para indicar que etiqueta va dentro de otra.
- Socket.io
  Librería que ayuda a implementar un sistema de eventos en tiempo real con node.js
- Express
  Es framework MVC bastante ligero que ayuda a construir aplicaciones webs.

Descripción de cada elemento desarrollado

1. Creación de una presentación de la aplicación con ejemplos explicativos de código.
2. Creación de la aplicación servidor utilizando la plataforma node.js
3. Creación de la web que se conectará al servidor y que mostrará al usuario el videojuego pong jugable como resultado final.

Objetivos

Investigar novedosos métodos cliente/servidor.
Poder hacer que otros aprendan de manera sencilla las bases del modelo cliente/servidor gracias a node.js.
Realizar un software que sea libre y didáctico.

Licencia
La licencia elegida para este proyecto ha sido MIT, se considera software libre, no tiene copyleft y es compatible con la GPL.

Tutorial

Paso 1.- Instalación y estructura de ficheros

Para empezar a programar se usará el directorio paso1 que está a su vez en la carpeta de versiones, esta versión del software.

Lo primero será ejecutar el comando npm install para instalar todas las dependencias, este se ejecutará en la raiz del directorio.

El directorio se compone de los siguientes ficheros:

La estructura de directorios y ficheros es la siguiente:

- node_modules: Contiene todos los modulos de node necesarios para trabajar.
- public: En este directorio tenemos el cliente de juego y la hoja de estilos
- views: Donde tenemos el index de la web que contendrá la aplicación
- nodepong.js: Donde tenemos el servidor y la lógica de juego.
- package.json: Que contiene información de la aplicación

Por ejemplo, el package.json de esta aplicación es el siguiente:

```
{
    "name": "nodepong",
    "version": "0.0.1",
    "description": "Pong en red",
    "dependencies": {
        "socket.io": "0.9.16",
        "express": "3.5.1",
        "jade": "1.3.0"
    },
    "author": "Jesusnoseq & PacoPGD"
}
```

Paso 2.- El juego

En este paso se explicará como realizar la clase del juego “pong.js” para que se pinte el tablero de juego en el cliente y se ejecute un servidor básico.

- function init(data): Esta función actua como "main" de nuestro cliente.

- function setFullScreen(data): Esta función nos permite poner el juego a pantalla completa .
- function paint(data): Esta función se encarga de pintar los distintos elementos en pantalla
- function paintLet(): Pintado de red de juego "Usada para simplificar paint"
- function paintData(data): Pintado de datos de juego "Usada para simplificar paint"

Función init

Esta función actuará como main de nuestra aplicación javascript, contendrá las siguientes lineas:

```
function init(data) {
	canvas = document.getElementById('game');
	ctx = canvas.getContext('2d');

	setFullScreen(data);

	paint(data,false);
}
```

Las dos primeras lineas son las que preparan nuestro canvas, siendo la primera la que obtiene el id del canvas que está creado en el main y la segunda linea la que especifica que nuestro canvas será 2d.
Las dos siguientes lineas llaman a nuestras funciones que nos harán tener el juego a pantalla completa y pintarán el juego.

Función setFullScreen

La función setFullScreen nos permite tener el juego a pantalla completa haciendo que el canvas se adapte al tamaño de la pantalla del dispositivo que lo ejecute.

```
function setFullScreen(data) {
	canvas.width = data.width;
	canvas.height = data.height;

	var w = window.innerWidth / canvas.width;
	var h = window.innerHeight / canvas.height;
	var scale = Math.min(h, w);

	//DEFINIR ANCHO Y ALTO DEL CANVAS
	canvas.style.width = (canvas.width * scale) + 'px';
	canvas.style.height = (canvas.height * scale) + 'px';

	//CENTRAR PANTALLA
	canvas.style.position = 'fixed';
	canvas.style.left = '50%';
	canvas.style.top = '50%';
```

```
        canvas.style.marginLeft = -(canvas.width * scale) / 2 + 'px';
        canvas.style.marginTop = -(canvas.height * scale) / 2 + 'px';
}
```

Las dos primeras líneas recogen del servidor el tamaño que va a tener nuestro canvas, si queremos darle un tamaño fijo podemos usar variables enteras para ello, con las 3 siguientes líneas definimos la escala que tiene nuestro juego y las últimas líneas nos permiten centrar la pantalla fijando la posición de nuestro canvas y dejando unos márgenes identicos a sus lados.

Función Paint

La función paint es la que pinta los elementos en pantalla, simplemente usamos ctx.fillStyle para definir el color de los elementos que pintaremos y ctx.fillRect para pintar rectangulos, siendo cada uno el que se indica en el comentario, los 4 parametros que se le pasan a fillRect son la posición de inicio del rectangulo respecto a X y a Y además de su ancho y largo.

```
function paint(data,panels) {
        //FONDO
        ctx.fillStyle = '#000';
        ctx.fillRect(0, 0, canvas.width, canvas.height);

        //BARRA DEL JUGADOR 1
        ctx.fillStyle = '#fff';
        ctx.fillRect(data.p1.x, data.p1.y, data.p1.w, data.p1.h);

        //BARRA DEL JUGADOR 2
        ctx.fillRect(data.p2.x, data.p2.y, data.p2.w, data.p2.h);

        //BOLITA
        ctx.fillRect(data.ball.x, data.ball.y, data.ball.w, data.ball.h);

        //RED
        paintLet();

        //DATOS
        if(panels){
                paintData(data);
        }

}
```

Las funciones paintLet y paintData son las que pinta la red central del tablero y los goles que han marcado los jugadores.

NOTA: Si no se ha configurado la parte servidor no llegará nada de data por lo que tendremos que poner valores numéricos para que estos rectángulos puedan ser representados.

Las funciones axiliares paintLet y paintData funcionan de forma similar:

```
function paintLet()
{
	var i;
	for(i=10;i<500;i+=20)
	{
		ctx.fillRect(495,i,10,10);
	}
}
```

La unica función nueva es la de fillText que nos permite introducir un texto, tiene 3 parametros de entrada, el primero es el texto que se verá en pantalla y los otros dos la posición que esté ocupara en el canvas, se ha usado ctx.font para definir la fuente y el tamaño de la misma.

```
function paintData(data)
{
	//GOLES J1
	ctx.font="75px Arial";
	ctx.fillText(data.p1.points,390,75);

	//GOLES J2
	ctx.fillText(data.p2.points,560,75);
}
```

Servidor

El servidor se va a encargar en este caso de llevar la lógica del juego, manejar los eventos que envían los clientes, enviar datos a todos lo clientes y manejar la cola de usuarios para permitir que cuando se salga un usuario que esté jugando se pueda continuar jugando.

A continuación se va a explicar cada parte del servidor que se encuentra en el fichero nodepong.js.

```
var express = require("express");
var app = express();
var port = 8080;

// Fijando el sistema de plantillas
app.set('view engine', "jade");
app.engine('jade', require('jade').__express);

// Indicando las rutas de ficheros de vistas y estáticos
app.set('views', __dirname + '/views');
app.use(express.static(__dirname + '/public'));
```

```

```

En estas primeras líneas se carga el framework express y se fija como sistema de plantillas Jade, además se especifica en qué directorio se encontrarán las vistas y los ficheros estáticos.

```
// Definiendo rutas
app.get("/", function(req, res) {
    res.render("index");
});
```

Siguiendo este ejemplo se pueden definir rutas que cargarán una vista previamente creada. En este caso se crea una ruta para mostrar la vista index que renderizará el fichero index.jade.

```
app.listen(port);
console.log("Listening on port " + port);
```

Con estas líneas se le indica a aplicación en qué puerto debe escuchar. Además usando console.log lo mostramos por la terminal del servidor.

Paso 3 - Comunicación con socket.io

Lógica del juego

En este paso se codificará la lógica que llevará el juego. Se programará el sistema de colisiones y las clases necesarias.

```
// Constantes del juego
var FPS = 30;
var W = 1000;
var H = 500;
var KEY_UP = 38;
var KEY_DOWN = 40;
var TOP = 0;
var LEFT = 1;
var BOT = 2;
var RIGHT = 3;
var STARTED = 1;
var STOPPED = 0;
```

Estas son las constantes para controlar el juego

```
var Actor = (function(x, y, w, h, vx, vy) {
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
    this.vx = vx;
    this.vy = vy;
});
```

Este objeto es el de la paleta de jugador, que hereda de actor, para realizar la herencia hay que usar las lineas Actor.call y la ultima linea Paddle.prototype = new Actor().

Las funciones que tiene nuestra paleta son las de lastKey que recoge la última pulsación realizada por el jugador la cual es enviada desde el cliente y la función move que mueve la paleta en función de la tecla pulsada por el jugador,

```
var Paddle = (function(x, y, w, h, vx, vy) {
    Actor.call(this, x, y, w, h, vx, vy);
    this.points = 0;
    this.dir=0;

    this.lastKey=(function(key){
        this.dir=key;
    });

    this.move = (function() {
        switch(this.dir) {
                case KEY_UP:
                        if (this.y > 0) {
                                this.y -= this.vy;
                        }
                        break;
                case KEY_DOWN:
                        if ((this.y + this.h) < H) {
                                this.y += this.vy;
                        }
                        break;
                default:

        }
        this.dir=0;
    });
});
Paddle.prototype = new Actor();
```

El objeto de Ball es el que nos permitirá representar a la bola en el juego, también hereda de actor.

Su función move es la que permite que la bola esté en constante movimiento haciendo que el ángulo afecte a la velocidad de la bola.

La función collide es la función que nos permite controlar las colisiones con las barras de los jugadores o con las paredes, haciendo que si esta rebota invierta su ángulo para que por lo tanto invierta su dirección y vuelva al campo.

La función de resetball resetea la bola después de un gol, simplemente la sitúa en su posición inicial de juego.

```
var Ball = (function(x, y, w, h, vx, vy, angle) {
    Actor.call(this, x, y, w, h, vx, vy);
    // en radianes
    this.angle = angle;


    this.move = (function() {
        var vyt = (this.vy * Math.sin(this.angle));
        var vxt = (this.vx * Math.cos(this.angle));
        this.x += vxt;
        this.y -= vyt;
    });

    this.collide = (function(pos) {
        switch(pos) {
                case TOP:
                        this.angle = -this.angle;
                        break;
                case LEFT:
                        if (this.angle>0){
                                this.angle -= Math.PI / 2;
                        }else{
                                this.angle += Math.PI / 2;
                        }
                        break;
                case BOT:
                        this.angle= -this.angle;
                        break;
                case RIGHT:
                        if (this.angle>0){
                                this.angle += Math.PI / 2;
                        }else{
                                this.angle -= Math.PI / 2;
                        }
                        break;
        }
    });

    this.resetBall= (function (dir) {
        b.x = W / 2 - b.w / 2;
        b.y = H / 2 - b.h / 2;
        if(dir==LEFT){
                b.angle=-Math.PI/4;
        }else{
                b.angle=(Math.PI/4)*3;
        }
    });
});
Ball.prototype = new Actor();
```

La función collides nos devuelven las colisiones que pueden ocasionarse entre los objetos geneardos y la función checkCollisions chequea si se ha metido un gol “Al producirse colisión con la pared izquierda o derecha del campo o si esta colisión se ha realizado con la pared superior o inferior “llamando a la función collide anteriormente mencionada” o si se colisiona con una de las palas de jugador.

```
function collides(a, b) {
        return !(
        ((a.y + a.h) < (b.y)) ||
        (a.y > (b.y + b.h)) ||
        ((a.x + a.w) < b.x) ||
        (a.x > (b.x + b.w))
        );
}

function checkCollisions() {
    //goles
    if ((b.x + b.w) < 0) {
         b.resetBall(LEFT);
         p2.points++;
    }

    if (b.x > W) {
         b.resetBall(RIGHT);
         p1.points++;
    }

    //colisiones con paredes
    if (b.y <= 0) {
         b.collide(TOP);
    }
    if ((b.y + b.h) >= H) {
         b.collide(BOT);
    }

    //colision con palas
    if (collides(p1, b)) {
         b.collide(LEFT);
    }

    if (collides(p2, b)) {
         b.collide(RIGHT);
    }
}
```

Las funciones gameStart y gameStop nos permiten inician y para el juego, gameReset hace que comience una nueva partida haciendo que los goles vuelvan a 0 y que la bola vuelva a su posición inicial.

```
function gameStart() {
```

```
    interval = setInterval(update, 1000 / FPS);
    gameStatus=STARTED;
}

function gameStop() {
    clearInterval(interval);
    gameStatus=STOPPED;
}

function gameReset() {
    b.resetBall();
    p1.points = 0;
    p2.points = 0;
}
```

La función chagePlayer para que si uno de los dos jugadores sale de partida este puesto se ocupe con el siguiente jugador en la cola de jugadores en espera.

```
function chagePlayer(){
    for (var i = 0; i < users.length; ++i){
        if(users[i]!=player1 && users[i]!=player2){
                if(player1==null){
                        player1=users[i];
                        return true;
                }else if(player2==null){
                        player2=users[i];
                        return true;
                }
        }
    }
    return false;
}
```

Paso 4 - Todo junto

En este paso se programará la comunicación entre el cliente y el servidor con socket.io. Principalmente en este paso crearemos y manejaremos eventos.

```
// variables del juego
var playerCounter = 0;
var gameStatus=STOPPED;
var interval;

var paddleW = 15;
var paddleH = 100;
var ballDiam = 20;
```

```
var p1 = new Paddle(10                              , H/2-paddleH/2 , paddleW , paddleH ,
0 , 20 );
var p2 = new Paddle(W-10-paddleW             , H/2-paddleH/2 , paddleW , paddleH , 0 , 20
);
var b  = new Ball  (W / 2 - ballDiam/2     , H/2-ballDiam/2, ballDiam, ballDiam, 10, 10,
-Math.PI/4);

var player1 = null;
var player2 = null;
var users = [];
```

Estas son las variables utilizadas para controlar el juego.
Se crea la paleta de la izquierda (p1), la de la derecha(p2), las variables player1 y player2 que guardaran el ID del jugador y por último un array donde se guardaran los IDs de los usuarios conectados.

```
function update() {
    p1.move();
    p2.move();
    b.move();
    checkCollisions();
    io.sockets.emit('draw', {
        'p1' : p1,
        'p2' : p2,
        'ball' : b
    });
}
```

Esta función se encarga de actualizar el estado del juego y mandar los datos actualizados a todos los clientes usando la función io.sockets.emit para ello.

Servidor

```
// Parte para cada cliente
io.sockets.on('connection', function(socket) {
    //socket.set('id',playerCounter);
    //console.log(socket);
    var myid = playerCounter;

    playerCounter++;

    console.log("ID del nuevo cliente: " + myid);
    socket.on('adduser', function() {
        users.push(myid);
        if(gameStatus==STOPPED /*player1==null || player2==null*/){
                chagePlayer();
        }

        if (gameStatus==STOPPED && player1!=null && player2!=null) {
                gameStart();
        }
```

```
    });

    // Datos iniciales para el cliente
    socket.emit('initData', {
        'width' : W,
        'height' : H,
        'p1' : p1,
        'p2' : p2,
        'ball' : b
    });

    // cada vez que el cliente pulsa una tecla
    socket.on('keypress', function(data) {
        if (myid == player1) {
                p1.lastKey((data.key));
        } else if (myid == player2) {
                p2.lastKey((data.key));
        }
    });


    socket.on('disconnect', function() {
        // Borro el ID del cliente desconectado del array de clientes
        users.splice(users.indexOf(myid), 1);
        // Si el cliente desconectado es un juegador se para el juego
        // y se busca otro jugador
        if(myid==player1 || myid==player2){
                gameStop();
                gameReset();

                if (myid == player1) {
                        player1=null;
                } else if (myid == player2) {
                        player2=null;
                }
                // Si se consigue reemplazar al jugador se pone el juego de
                // de nuevo en marcha
                if(chagePlayer()){
                        gameStart();
                }
        }
    });

});
```

Este es el código que se ejecutará individualmente para cada cliente.
En el se empieza asignando un ID al nuevo cliente o socket y se aumenta el contador de usuarios.
En el evento *adduser* que se recibe del cliente, el servidor se encarga de introducirlo en el array de usuarios, hacerlo jugador si no los hay y si el juego no está empezado lo empieza si hay dos usuarios conectados al servidor.
El evento del cliente keypress se encarga de que en caso de que ese cliente sea un jugador, actualizar la dirección que debe de tomar la paleta al actualizar el juego.
El servidor emite el evento initData con los datos iniciales que el cliente necesita para dibujar el juego.

En evento disconnected se activa cuando dicho usuario se desconecta. En este caso cuando un usuario se desconecta se borra su ID del array de usuarios y si era un jugador se reiniciara y para el juego y se intentará colocar algún cliente en su lugar, si es posible, el juego se reanuda.

Cliente

```
window.onload = (function() {
    //var nick = prompt("Please enter your name:");

    var socket = io.connect(url);

    socket.on('connect', function() {
        socket.emit('adduser');//,{'nick':nick});
    });

    socket.on('draw', function(data) {
        paint(data,true);
        console.log("recibiendo");
    });

    socket.on('disconnect', function() {
        console.log('Disconnected!');
    });

    socket.on('initData', function(data) {
        init(data);
    });

    //Leemos que tecla estamos pulsando
    document.addEventListener('keydown', function(evt) {
        var lastPress = evt.keyCode;
        if (lastPress == KEY_UP || lastPress == KEY_DOWN) {
                socket.emit('keypress', {
                        'key' : lastPress
                });
        }
    }, false);
});
```

Esta parte de código que se encuentra en pong.js así que se ejecutará en el cliente cuando cargue la página.
En primer lugar nos conectamos al servidor con "*var socket = io.connect(url);" u*na vez conectados empezamos a mandar y recibir eventos.
El evento connect se lanza una vez nos hemos conseguido conectar con el servidor, como se puede ver en el código, una vez conectados mandamos al servidor el evento adduser como "callback" para indicar que nos hemos conectado.
Del servidor recibimos también otros eventos ya explicados en el apartado anterior.
Hay que indicar que cada vez que se recibe el evento draw se pinta de nuevo.

MÁS INFORMACIÓN

- https://github.com/learnboost/socket.io/wiki
- http://expressjs.com/guide.html
- http://jade-lang.com/