



**Universidad
Rey Juan Carlos**

**Grado en Ingeniería en Sistemas Audiovisuales y
Multimedia**

Escuela Técnica Superior de Ingeniería de Telecomunicación

Curso académico 2016-2017

Trabajo Fin de Grado

Prácticas docentes de desarrollo web

Autor: Walter Rene Cuenca Guachamin

Tutor: José María Cañas Plaza

Madrid 2017

Dedicatoria

*'El genio es 1 % de inspiración y
99 % de transpiración'.*

By Tomas Edison

Agradecimientos

*A todo mi familia pero en especial a mi mama,
por ser madre y padre para mi,
y apoyarme incondicionalmente todo este tiempo.
y por creer en mi.*

*Y a mis amigos, por ser
amigos de verdad.*

Gracias.

Resumen

La utilización de tecnologías y aplicaciones Web presenta un gran crecimiento debido a sus características logrando remplazar en muchos casos a las aplicaciones convencionales de escritorio. Dentro de sus características destaca su fácil acceso, la no necesidad de instalar ni actualizar componentes en el cliente y ser independiente del sistema operativo.

Este TFG sirve como apoyo en las prácticas de la asignatura de LTAW del grado de Ingeniería de sistemas audiovisuales y multimedia. Consta de cuatro prácticas que utilizan diferentes tecnologías Web, herramientas y bibliotecas muy acentuadas en este campo. Algunas de las tecnologías empleadas son JavaScript en el cliente, NodeJS y Django en el servidor, Base de datos como MySQL y por último WebSockets y WebRTC como tecnologías de comunicación fluida. Para cada una de las prácticas se han realizado vídeos¹ demostrativos de su utilización por terceras personas.

¹<https://www.youtube.com/channel/UCCHPTA6ztkPFhwvUbDTR5uQ>

Índice general

Índice de figuras	7
1. Introducción	1
1.1. Tecnologías Web	1
1.2. Tecnologías cliente	6
1.3. Tecnologías servidor	7
1.4. Antecedentes	8
2. Objetivos	12
2.1. Metodología	12
2.2. Plan de trabajo	14
3. Infraestructura	15
3.1. HTML5	15
3.1.1. Canvas	15
3.1.2. Media	20
3.1.3. WebSockets	21
3.1.4. API File	24
3.2. Lenguaje JavaScript	25
3.3. Biblioteca JQuery	25
3.4. Biblioteca Bootstrap	26
3.5. Entorno de servidor NodeJS	26
3.6. Base de datos en aplicaciones web	28
3.6.1. MySQL	28
3.7. Entorno de servidor Django	29
3.8. WebRTC	32
3.9. Web Services	40
3.9.1. WebServices Google Maps	42
4. Comecocos Web	43
4.1. Enunciado	43
4.2. Diseño	44
4.3. Desarrollo	45
4.3.1. Game Area	45
4.3.2. Pacman	50

4.3.3. Fantasmas	54
4.3.4. Movimiento	57
4.4. Pruebas	59
5. Comecocos Multijugador	62
5.1. Enunciado	62
5.2. Diseño	63
5.3. Desarrollo Servidor	64
5.4. Desarrollo Cliente	72
6. Tienda Web	83
6.1. Enunciado	83
6.2. Desarrollo Back-End	84
6.3. Desarrollo Front-End	88
7. VideoConferencia usando WebRTC	102
7.1. Enunciado	102
7.2. Diseño	103
7.3. Desarrollo Servidor Señalización	104
7.4. Desarrollo Cliente Peer-to-Peer	106
8. Conclusiones	117
8.1. Conclusiones	117
8.2. Trabajos futuros	118
Bibliografía	119

Índice de figuras

1.1. Esquema Cliente-Servidor	1
1.2. Evolución de las tecnologías web.	4
1.3. Ejemplo Web 1 ^a generación	4
1.4. Spotify.	5
1.5. YouTube.	6
1.6. Netflix.	6
1.7. Comparativa uso de los navegadores	7
1.8. Esquema Surveillance 5.1	8
1.9. Interfaz Web Surveillance 5.1.	9
1.10. Esquema JdeRobotWebClients	9
1.11. Interfaz JdeRobotWebClients	10
1.12. Arquitectura de la aplicación web Dron con WebRTC	10
1.13. Interfaz Web Dron con WebRTC	10
2.1. Metodología en cascada	13
3.1. Ejemplo del dibujo de un trazo	16
3.2. Ejemplo rectángulos canvas	17
3.3. Ejemplo texto canvas	18
3.4. Ejemplo escalado de imagen canvas.	19
3.5. Ejemplo recorte imagen canvas.	19
3.6. Comparación Polling-WebSockets	22
3.7. Ejemplo Base de Datos.	29
3.8. Esquema MVC Django.	30
3.9. Proceso del Servidor Señalización	33
3.10. Ejemplo Protocolo SDP	34
3.11. Ejemplo petición-respuesta STUN	35
3.12. Ejemplo petición-respuesta TURN	36
3.13. Comparativa Soap y Rest	41
4.1. Aspecto clásico Pacman	43
4.2. Diseño come cocos web	44
4.3. Apariencia elementos del juego	48
4.4. Apariencia información del juego	49
4.5. Colisión Pacman-Obstáculo	52
4.6. Colisión Pacman-Cocos	53

4.7. Apariencia comportamiento Fantasmas	58
4.8. Inicio Juego	60
4.9. Pause/Start juego.	60
4.10. Game/Loose juego.	61
5.1. Ejemplo Aspecto multijugador Pacman	62
5.2. Esquema Pacman multijugador	63
5.3. Ejecución del servidor	64
5.4. ServidorSeñalizacion	67
5.5. ServidorSeñalizacion	68
5.6. Petición de partida usuario.	68
5.7. Respuesta de partida usuario.	68
5.8. Envío elementos del juego.	69
5.9. Envio actualización del juego.	71
5.10. Envio actualización del juego.	71
5.11. Envío coco comido al usuario.	72
5.12. Pagina Inicio Pacman-Online	73
5.13. Petición/Respuesta sala 1 ^a jugador.	74
5.14. Petición/Respuesta sala 2 ^a jugador.	74
5.15. Envío petición partida.	75
5.16. Recepción petición partida.	76
5.17. Recepción respuesta a la petición de partida.	77
5.18. Recepción parámetros iniciales 1 ^a usuario.	78
5.19. Recepción parámetros iniciales 2 ^a usuario.	78
5.20. Recepción NewPos_Ghost 1 ^a usuario.	81
5.21. Recepción NewPos_Ghost 2 ^a usuario.	81
5.22. Aspecto clásico Pacman	82
6.1. Portada Paginas Web.	83
6.2. Diagrama modelos App.	85
6.3. Interfaz admin Django	86
6.4. Diagrama formularios de la aplicación.	86
6.5. Diagrama URLs de la aplicación.	87
6.6. Diagrama vistas de la aplicación.	88
6.7. Diagrama ficheros Front-End de la aplicación.	89
6.8. Barra de navegación	89
6.9. Buscador de la aplicación	89
6.10. Pagina Contacta	91
6.11. Pagina Registro	91
6.12. Formulario Login	92
6.13. Perfil Usuario	92
6.14. Pagina Principal	93
6.15. Cantante panel Información	94
6.16. Cantante panel Información	94
6.17. Cantante panel Discos	95
6.18. Cantante pestaña imágenes	96

6.19. Evento panel Información	97
6.20. Evento panel ubicación	97
6.21. Evento panel Cantantes	99
6.22. Evento pestaña Entradas	99
6.23. Detalle Carrito	100
6.24. Orden Compra	101
7.1. Ejemplo sitio Web	102
7.2. Creación oferta cliente.	103
7.3. Conexión Peer-to-Peer final.	103
7.4. Request/Replay Salas Servidor	104
7.5. Request/Replay conexión Servidor	105
7.6. Mensaje señalización Answer.	106
7.7. Mensaje señalización Offer.	106
7.8. Mensaje señalización Icecandidate.	107
7.9. Inicio de conexión.	108
7.10. Creación oferta cliente.	110
7.11. Creación de la respuesta.	111
7.12. Envió de caracteres del chat por RTCDataChannel.	113
7.13. Envió información del fichero con RTCDataChannel.	115
7.14. Recepción y reconstrucción del fichero	116

Capítulo 1

Introducción

Este Trabajo Fin de Grado se centra en el campo de las aplicaciones Web. En este capítulo ponemos el contexto a las aplicaciones Web en la actualidad y muestra trabajos previos en los que se aplica tecnologías web en su desarrollo.

1.1. Tecnologías Web

En ingeniería de software se denomina aplicación web a aquellas aplicaciones que los usuarios pueden utilizar accediendo a un servidor web a través de Internet mediante un navegador. Su popularidad se debe al emplear los navegadores webs como clientes ligeros, a la independencia del sistema operativo, así como a la facilidad para actualizar y mantener sin distribuir e instalar software a miles de usuarios potenciales.

Las aplicaciones Webs se basan en la arquitectura cliente-servidor en el que las tareas se reparten entre los servidores y los clientes.

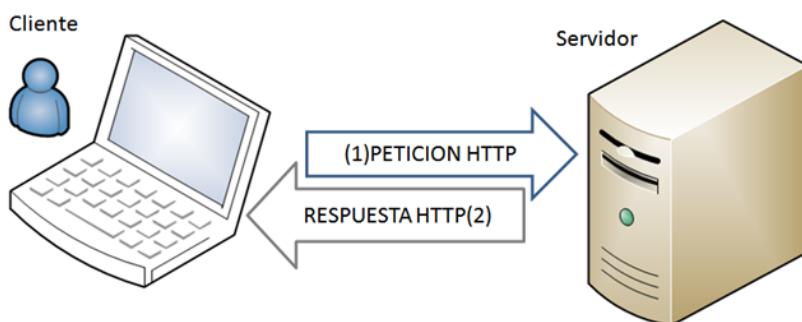


Figura 1.1: Esquema Cliente-Servidor.

Protocolo HTTP

HTTP (*Hyper Text Transfer Protocol*) protocolo desarrollado por el *World Wide Web Consortium* y la *Internet Engineering Task Force*. Funciona a través de la solicitud y

respuesta entre un cliente (navegador) y un servidor, dando lugar a una secuencia de mensajes conocidos como sesión HTTP.

Se trata de un protocolo orientado a transición siguiendo el esquema petición-respuesta. El cliente que realiza la petición es conocido como *user-agent*, la información transmitida se denomina recurso y se la identifica mediante un localizador uniforme de recursos (*URL*). Los recursos pueden ser de varios tipos como archivos, el resultado de la ejecución de un programa, una consulta a base de datos, etc.

La comunicación se realiza en dos etapas, la primera es la solicitud del cliente que consta de una cabecera y opcionalmente de contenido. Dentro de la cabecera se encuentra el método que queremos invocar (GET, POST, PUT,) y en ocasiones información sobre la petición y el cliente. La segunda es la respuesta del servidor sobre la operación solicitada en forma de código numérico que permite conocer el éxito o fracaso de dicha operación.

HTTP es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores. El desarrollo de aplicaciones web necesita frecuentemente mantener estado por lo que se utilizan las *cookies*, que es información que un servidor puede almacenar en el sistema cliente para simular la noción de sesión.

HTML

Son las iniciales de la expresión en inglés *HyperText Markup Language*. Se trata de un conjunto de etiquetas que se van intercalando entre el texto de forma que los navegadores sepan qué es lo que tienen que mostrar cuando accedemos a una página y cómo deben presentarlo en pantalla.

El W3C¹ (World Wide Web Consortium) es el fórum internacional que se encarga desarrollar nuevas tecnologías relacionadas con la Web dictando las normas que constituyen el estándar HTML. A lo largo de sus diferentes versiones, se han incorporado y suprimido diversas características, con el fin de hacerlo más eficiente y facilitar el desarrollo de páginas web. La versión más extendida fue la 4.0 que permitía utilizar textos, sonidos, imágenes y enlaces a otras páginas enriqueciendo de esta forma la información de la página.

A medida que este lenguaje se extendía se vio la necesidad de actualizar la versión del estándar. Es aquí, donde surge HTML 5² incluyendo novedades significativas en diversos áreas, ya que no incorpora solo nuevas etiquetas o elimina otras, sino mejora áreas que estaban fuera del alcance del lenguaje.

- **Estructura del cuerpo:** Los sitios webs tienen varias partes de su estructura común como cabecera, pie, etc. HTML 5 permite agrupar todas estas partes de una web en nuevas etiquetas que representarán cada uno de las partes típicas de una página.
- **Etiquetas para contenido específico:** Hasta ahora se utilizaba una única etiqueta para incorporar diversos tipos de contenido enriquecido, como anima-

¹Web: <http://www.w3.org/>

²<http://www.w3.org/TR/2014/REC-html5-20141028/>.

ciones Flash o vídeo. Ahora se utilizarán etiquetas específicas para cada tipo de contenido como audio, vídeo, etc.

- **Canvas:** Nuevo componente que permitirá dibujar en la página todo tipo de formas, que podrán estar animadas y responder a interacción del usuario por medio de las funciones de un API.
- **Bases de datos locales:** El navegador permitirá el uso de una base de datos local, con la que se podrá trabajar en una página web por medio del cliente y a través de un API. Permiten almacenar grandes cantidades de información, lo que permitirá la creación de aplicaciones web que funcionen sin necesidad de estar conectados a Internet.
- **Web Workers:** Son procesos que requieren bastante tiempo de procesamiento por parte del navegador, pero que se podrán realizar en un segundo plano, para que el usuario no tenga que esperar que se terminen para empezar a usar la página. Para ello se dispondrá también de un API para el trabajo con los Web Workers.
- **Aplicaciones web Offline:** API que permite el trabajo con aplicaciones web, que se podrán desarrollar para que funcionen también en local y sin estar conectados a Internet.
- **Geolocalización:** Permite localizar geográficamente las páginas web por medio de un API de Geolocalización.
- **Nuevas APIs para interfaz de usuario:** Temas tan utilizados como el *drag & drop* en las interfaces de usuario de los programas convencionales, serán incorporadas al HTML 5 por medio de un API.
- **Fin de las etiquetas de presentación:** Todas las etiquetas que tienen que ver con la presentación del documento serán eliminadas. La responsabilidad de definir el aspecto de una web correrá a cargo únicamente de CSS.

Como se puede ver, existirán varios API con los que podremos trabajar para el desarrollo de todo tipo de aplicaciones complejas, que funcionarán online y offline.

Evolución

Las fuentes establecen varias generaciones de sitios Web que se explican mas adelante, como se puede ver en la figura 1.2. En la actualidad conviven distintas generación aunque la mayoría de sitios web sigue el modelo que establece la segunda generación.

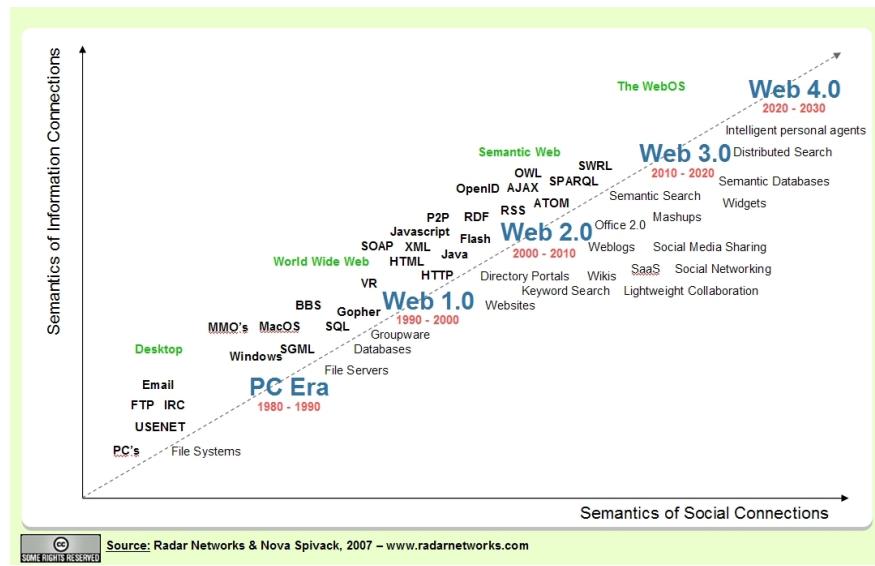
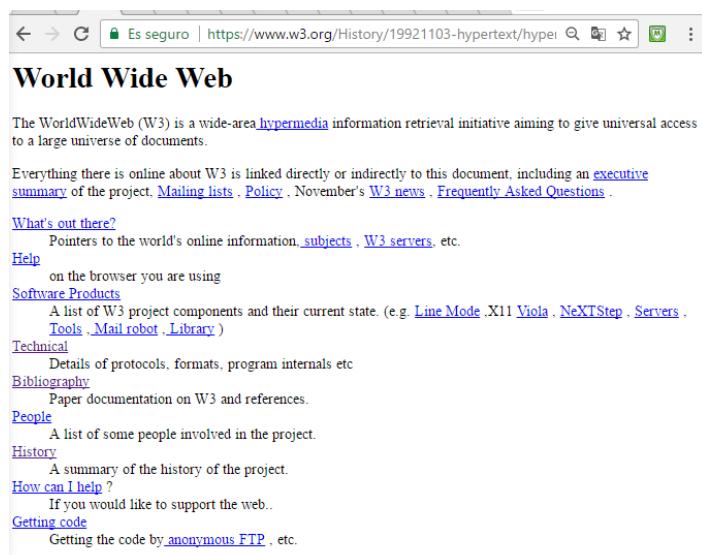


Figura 1.2: Evolución de las tecnologías web.

Primera generación

Esta primera generación abarca desde el nacimiento de la Web (1992) y es conocida como Web 1.0. Sus páginas eran simples documentos en lenguaje HTML constituidos por texto e interpretadas por los navegadores existentes en ese momento. Estos documentos los generaba una única persona que se encargaba del diseño y de obtener los datos necesarios presentando como obstáculo que los documentos solo eran de lectura como la figura 1.3.

Figura 1.3: Ejemplo Web 1^a generación.

Segunda generación

Con la llegada de las “compañías ciberneticas” (2001) Internet da un giro de 180 grados. El éxito de estas compañías dependía de webs mucho más dinámicas y para ello era necesario huir de sitios estáticos y poco actualizados, y servir páginas HTML dinámicas creadas al vuelo con actualizada base de datos.

Esta generación recibe el nombre Web 2.0, termino que aparece por primera vez en el año 2004 por parte Dale Dougherty (vicepresidente de O'Reilly Media) en una conferencia. El éxito de esta filosofía radica en el echo de que el editor del sitio web no se encarga de aportar el contenido sino que es la propia comunidad la que proporciona y promociona determinados contenidos.

Tercera generación

En el intento de comprender la propia Web 2.0 se vislumbran futuras etapas de la Web, sobre todo orientadas a mejorar la interactividad y la movilidad entre/de los usuarios. Apareciendo el término Web 3.0 asociado al concepto de Web Semántica, desarrollado bajo la tutela del creador de la Web Tim Berners Lee.

Básicamente, toda la información publicada en las diferentes páginas web no es entendible por los ordenadores, teniendo únicamente significado para las personas. La idea consiste en añadir información adicional a la información “visible”, de tal manera que pueda ser entendida por los ordenadores.

Aplicaciones

En la actualidad las aplicaciones web se han incorporado en varias áreas para facilitar y acercando el acceso a la información a los usuarios.

Una de las áreas que ha crecido enormemente es el servicio de contenido multimedia como es el caso de Spotify, figura 1.4, que permite a los usuarios escuchar música de forma online o YouTube, figura 1.5, que permite a los usuarios reproducir vídeos publicados por otros usuarios o vídeos que se emiten en directo.

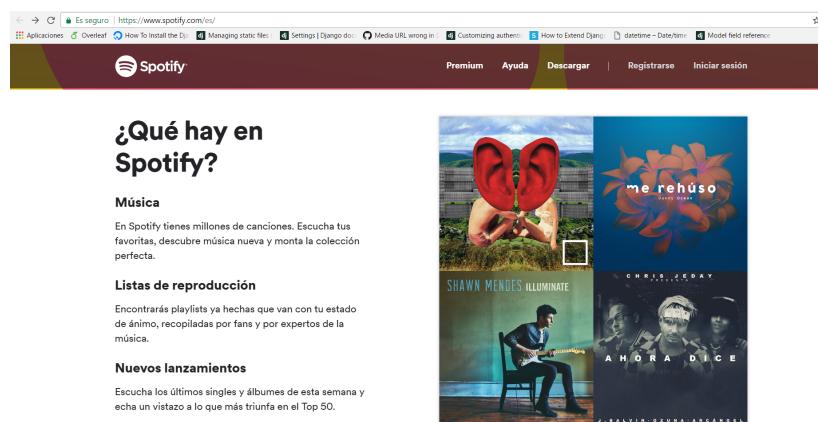


Figura 1.4: Spotify.

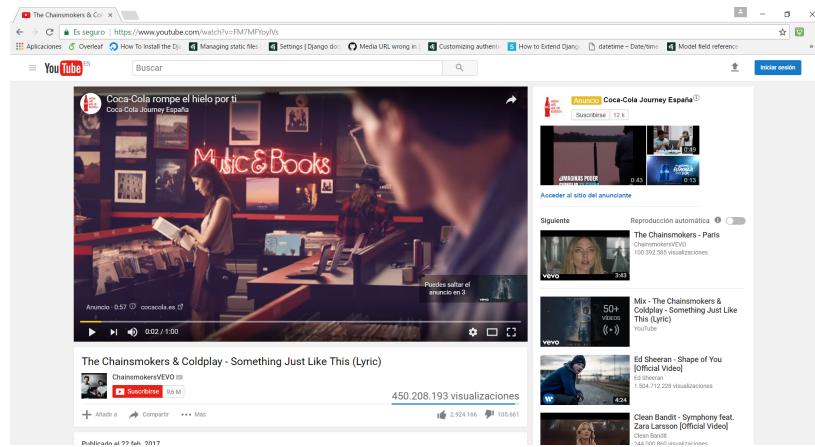


Figura 1.5: YouTube.

Otra área que ha ganado peso en estos últimos años es la creación de plataformas que sirven contenido televisivo a través de una aplicación web como es el caso de Netflix, figura 1.6.

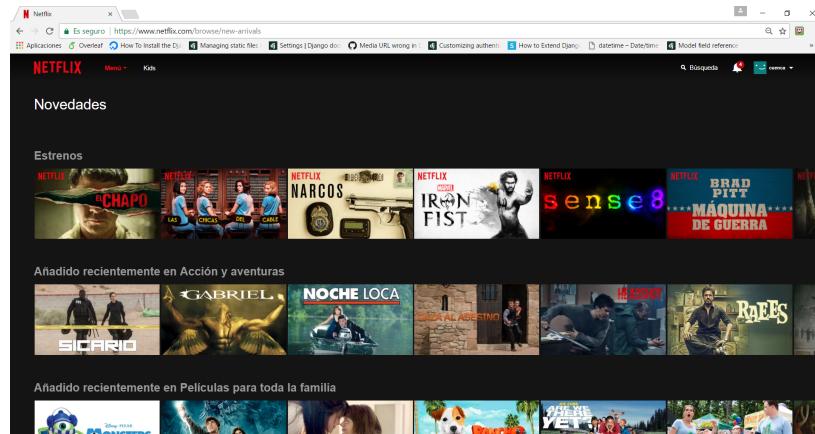


Figura 1.6: Netflix.

Aunque se ha mencionado algunos ejemplos existen muchos otras áreas donde la utilización de aplicaciones web ha mejorado la experiencia y accesibilidad al contenido por parte de los usuarios.

1.2. Tecnologías cliente

Antiguamente la interactividad de los usuarios era limitada provocando limitación en el desarrollo de interfaces y degradando la experiencia del usuario. Ante esto la primera solución fue incluir instrucciones que se ejecuten en el cliente desarrolladas en Java por medio del uso de plug-ins. Al tratarse de un lenguaje pesado, se optó por emplear instrucciones desarrolladas en JavaScript.

Lenguaje mucho mas ligero e interpretado por los navegadores, que al capturar un evento de la pagina ejecuta un trozo de código que generalmente cambia algún atributo de la pagina, comprueba los valores de un formulario antes de enviarlo, etc. A partir de este lenguaje surgen bibliotecas y entornos que permiten crear aplicaciones mas interactivas, como por ejemplo: Ajax, Jquery, Angular.

El rendimiento de los navegadores depende de la calidad de su implementación ya que son los encargados de interpretar el código de las paginas web. En la actualidad disponemos de los siguientes navegadores:

1. Internet Explorer: Creado en 1995 aunque no fue hasta el año 2000 que domino absolutamente el mercado. Pero con la llegada de Firefox su uso global descendió de una forma notable.
2. Firefox: Creado por la fundación Mozilla y es la continuación al navegador Mozilla. Se publico en 2004 con el objetivo de permitir que la web sea publica, abierta y accesible.
3. Safari: Creado en 2003 para el sistema operativo Mac de Apple ya que antes no disponía de un navegador propio.
4. Chrome: Creado en 2008 por Google a partir de WebKit. Destaca por su interfaz minimalista y por la velocidad de ejecución del código JavaScript, lo que obligó a sus competidores a ponerse las pilas en estos aspectos.

La figura 1.7 muestra una comparativa del porcentaje de uso de cada uno de los navegadores.

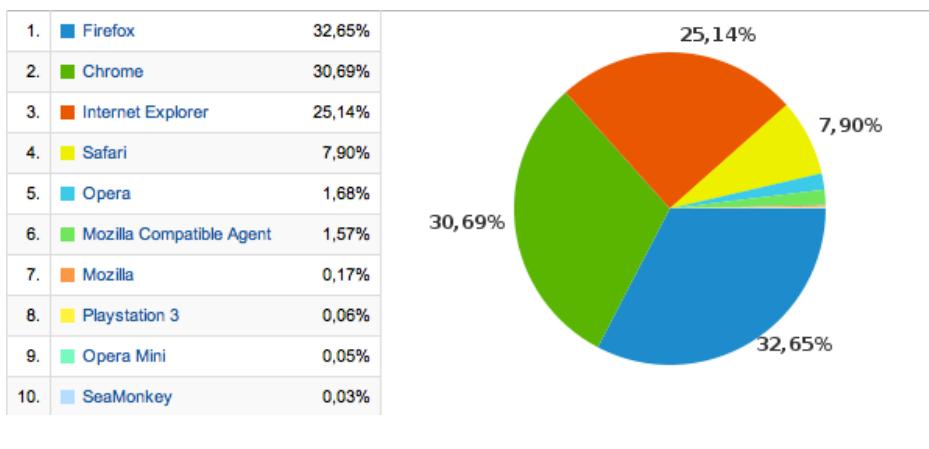


Figura 1.7: Comparativa del uso de los navegadores.

1.3. Tecnologías servidor

Este tipo de tecnologías se basan en el procesamiento de peticiones de los usuarios mediante la interpretación de un *script* en el servidor web para generar páginas HTML dinámicamente como respuesta.

Así pues, podemos hablar de lenguajes de lado servidor que son aquellos lenguajes que son reconocidos, ejecutados e interpretados por el propio servidor, por ejemplo: ASP, PHP, JSP, NodeJS. Estos lenguajes son especialmente útiles en trabajos en los que se tiene que acceder a información centralizada, situada en una base de datos en el servidor, y cuando por razones de seguridad los cálculos no se pueden realizar en el cliente.

Es importante destacar que los lenguajes del lado del servidor son necesarios porque la mayoría de las aplicaciones web necesitan tener acceso a muchos recursos externos al cliente, principalmente bases de datos alojadas en servidores de Internet.

1.4. Antecedentes

El TFG centra su desarrollo en el campo de las tecnologías Web y como a través de ella permiten generar diversas aplicaciones con las que el usuarios pueden interactuar. A continuación, se presentan varios ejemplo de TFGs dentro del Laboratorio de Robótica de la URJC que emplean estas tecnologías y han sido el punto de partida de este TFG.

Surveillance 5.1 (URJC)

Surveillance 5.1 [3][4] fue desarrollado por Edgar Barrero como trabajo fin de grado. La aplicación web ofrece un flujo de vídeo desde una cámara web, un flujo de imagen de profundidad procedente de un sensor Kinect y su representación en 3D, además de un sensor de humedad y un actuador.

La aplicación web se desarrolló en Ruby sobre Rails. El servidor de la aplicación se conectaba a los componentes de JdeRobot a través del interfaz ICE.

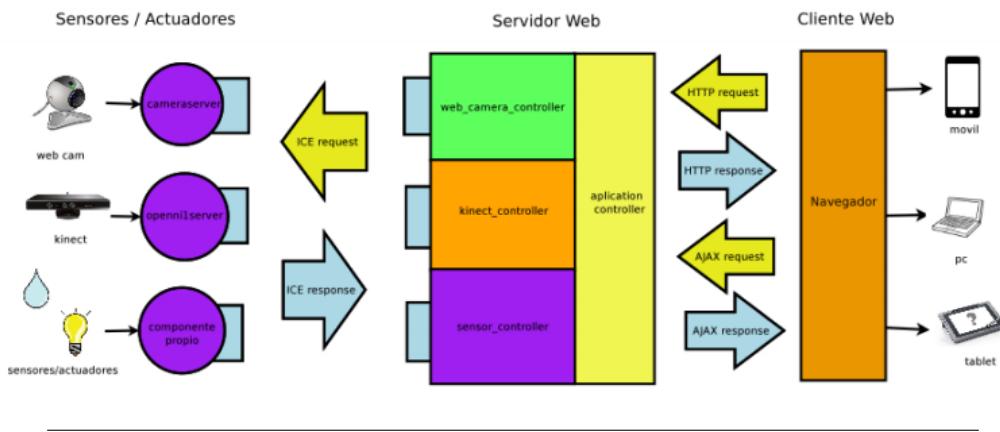


Figura 1.8: Esquema Surveillance 5.1.

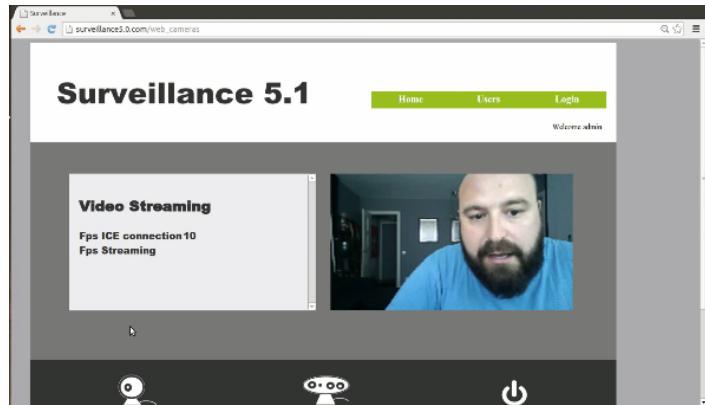


Figura 1.9: Interfaz Web Surveillance 5.1.

JdeRobotWebClients (URJC)

JdeRobotWebClients [5][6] fue desarrollado por Aitor Martínez Fernández como trabajo fin de grado. La aplicación Web crea seis versiones web de herramientas utilizadas por JdeRobot (CameraViewJS, RGBDViewerJS, KobukiViewerJS,) que estaban programadas en C++ o Python con su propio interfaz gráfico provocando que sean ejecutables solo en Linux.

Estas nuevas versiones son multiplataforma (Linux, Android, IOS, Windows) y accesibles desde un navegador web como interfaz gráfico permitiendo acceder a los sensores y actuadores sin un servidor intermedio.

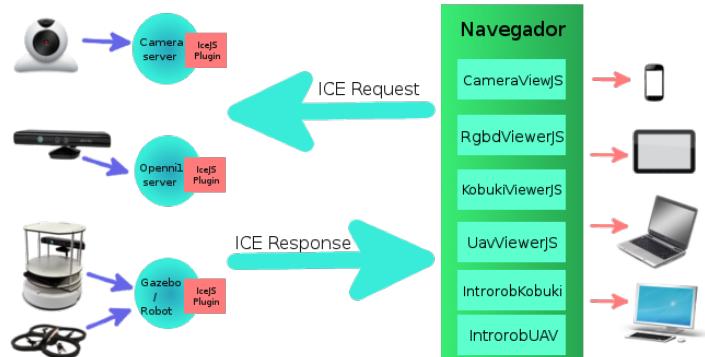


Figura 1.10: Esquema JdeRobotWebClients.

Drone con WebRTC.

Drone con WebRTC [7][8] fue desarrollado por Iván Rodríguez como trabajo de fin de grado. El objetivo era poder teleoperar un dron desde un dispositivo móvil

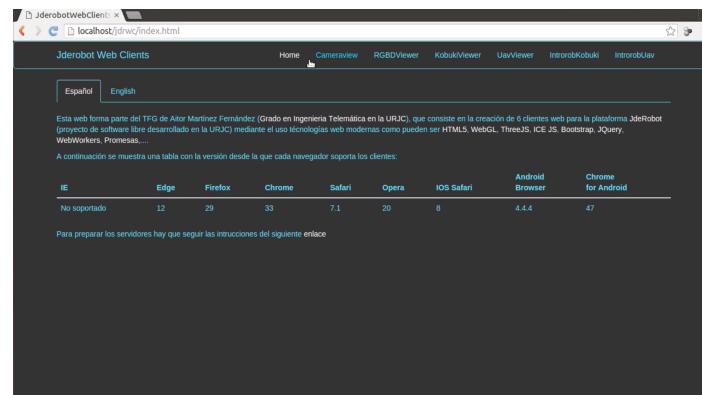


Figura 1.11: Interfaz Web JdeRobotWebClients.

(teléfono, tablets....) por medio de aplicaciones Web, es decir, desde un navegador web. Su desarrollo se basaba en WebRTC como tecnología Web para teleoperar el dron, además de emplear las tecnologías propias de JdeRobot para establecimiento de conexión entre el dron y los dispositivos móviles.

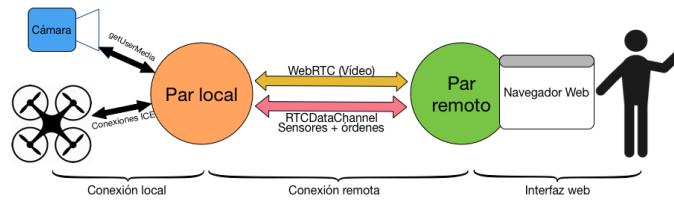


Figura 1.12: Arquitectura de la aplicación web Dron con WebRTC

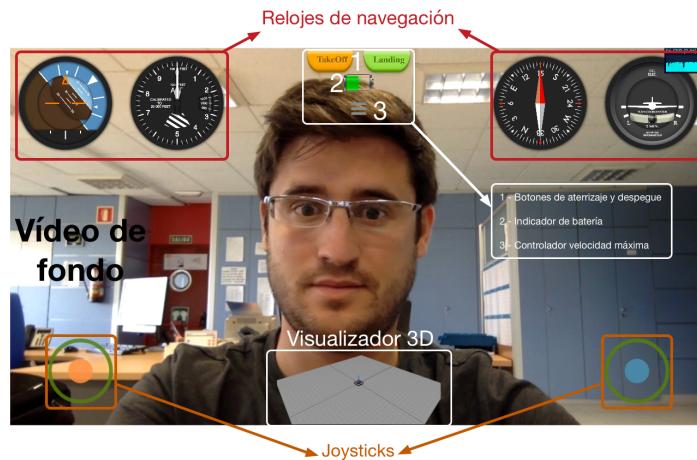


Figura 1.13: Interfaz Web Dron con WebRTC

La memoria está organizada en 8 capítulos. Tras presentar el contexto en la introducción, en el segundo capítulo se fijan los objetivos y el plan de trabajo empleado. El tercer capítulo reúne la descripción de las distintas tecnologías que se emplearan en el desarrollo del TFG. Del cuarto al séptimo capítulo trata cada una de las prácticas fijadas en el capítulo dos. El capítulo final recoge las conclusiones obtenidas tras realizar los distintos desarrollos.

Capítulo 2

Objetivos

EL TFG se centra en un entorno docente, donde se quiere exponer a los alumnos de la asignatura de Laboratorios de Tecnologías Audiovisuales en la Web de cuarto curso del grado de Sistemas Audiovisuales y Multimedia a un conjunto variado de tecnologías web proponiéndoles cuatro prácticas atractivas, que se pueden elaborar como aplicaciones tradicionales, de escritorio, pero también se pueden elaborar empleando tecnologías web, navegadores y clientes, siendo de este modo multiplataforma. Este objetivo global será estructurado en cuatro subobjetivos, correspondientes a cada una las siguientes cuatro prácticas.

1. Crear el juego del comecocos en la web basado en tecnologías del cliente. Para la visualización se utilizará Canvas mientras que para la funcionalidad se utilizará JavaScript permitiendo ejecutar eventos del teclado, además de incluir audio.
2. Crear una versión juego multijugador del juego del comecocos basado en tecnologías de comunicación bidireccional en tiempo real. Su diseño se basará en un servidor creado en NodeJS y WebSockets como mecanismo de comunicación bidireccional entre los navegadores y el servidor.
3. Crear un sitio Web de una tienda basada en tecnologías de servidor con manejo de base de datos. Su diseño se basará en el entorno Django para la gestión del sitio de Web y como BBDD MySQL.
4. Crear una aplicación de Videoconferencia Peer-to-Peer entre navegadores basada en tecnologías de comunicación audiovisual. Su diseño se realizará con WebRTC como tecnología core y NodeJS para la creación del servidor auxiliar.

2.1. Metodología

En la realización del proyecto se ha necesitado definir una metodología que permita planificar las tareas necesarias para llegar a nuestro objetivo. Por ello se ha seleccionado el modelo de desarrollo en cascada, figura 2.1. Este modelo define un

conjunto de etapas distribuidas en cascada donde cada etapa tiene que terminarse por completo para pasar a la siguiente. A continuación, se explica en detalle los pasos en cada etapa.

1. **Análisis de requisitos:** En esta fase se analizan las necesidades para determinar qué objetivos debe cubrir.
2. **Diseño del Sistema:** Descompone y organiza el sistema en elementos que puedan elaborarse por separado. Como resultado se obtiene la descripción de la estructura global del sistema y su especificación.
3. **Diseño del Programa:** Es la fase en donde se realizan los algoritmos necesarios para cubrir los requerimientos y analizar las herramientas necesarias para la etapa de Codificación.
4. **Codificación:** Es la fase en donde se implementa el código fuente, haciendo uso de prototipos así como de pruebas y ensayos para corregir errores.
5. **Pruebas:** Los elementos, ya programados, se ensamblan para componer el sistema y se comprueba que funciona correctamente y que cumple con los requisitos.

Este esquema se aplica como ciclo de vida para desarrollar cada una de las prácticas de las que trata el TFG.



Figura 2.1: Metodología en cascada.

Como parte de la metodología, durante el tiempo que ha durado el proyecto se acordaron reuniones semanales con el tutor de forma presenciales o por Video-Conferencia en las que se revisaba los objetivos semanales y se definían los nuevos objetivos.

Los avances del proyecto se añadían en la bitácora web Mediawiki¹. El contenido de esta bitácora se divide en una sección de aprendizaje de tecnologías web con pequeños ejemplos y otra centrada en el desarrollo de cada una de las prácticas del TFG detallando las características más relevantes de cada una de ellas. El código empleado en los ejemplos y prácticas de esta bitácora se encuentra en el repositorio de GitHub² siguiendo la filosofía de software libre.

2.2. Plan de trabajo

Para llevar a cabo con éxito el desarrollo de las distintas prácticas se definió el siguiente plan de trabajo.

1. **Fase de aprendizaje:** Esta primera etapa explora el aprendizaje de diferentes tecnologías Web novedosas. Se dividió en tecnologías de cliente como JavaScript y HTML5, tecnologías de comunicación cliente-servidor como Ajax, formularios y WebSockets, tecnologías del servidor como NodeJS y Django, BBDD como MySql y MongoDB y por último tecnologías de comunicación audiovisual como WebRTC.
2. **Diseño del enunciado de las prácticas:** Con los conocimientos adquiridos de las diferentes tecnologías se realiza una propuesta del enunciado de cada práctica. En cada enunciado recoge las tecnologías y requisitos que se tienen que cubrir durante el desarrollo.
3. **Implementación de las prácticas:** Tras validar el enunciado de cada práctica se pasa al desarrollo siguiendo las pautas marcadas. Primero se diseña la solución cubriendo los requisitos marcados para luego pasar a su implementación teniendo una visión general de los módulos, funciones y otros recursos que necesitan ser programados.
4. **Pruebas:** Cada una de las prácticas han sido ejecutadas en distintos navegadores para evaluar su funcionalidad.

¹<http://jderobot.org/Walter-tfg>

²<https://github.com/RoboticsURJC-students/2015-TFG-Walter-Cuenca>

Capítulo 3

Infraestructura

En esta sección se describe con cierto detalle las herramientas, estándares y bibliotecas que hemos usado directamente en el desarrollo de este TFG.

3.1. HTML5

Este nuevo estándar incorpora una serie de APIs accesibles desde JavaScript que surgen para dotar de funcionalidad nativa a la Web sin necesidad de utilizar plug-ins externos. Los distintos ingredientes asociados a HTML5 [11] los describimos en las siguientes secciones.

3.1.1. Canvas

El elemento <canvas> [12] permite trabajar con gráficos orientados a píxel (no vectorial) dentro de la web sin la necesidad de emplear programas externos. Dispone de varios métodos que permiten realizar múltiples tareas de carácter gráfico que se han utilizado en el desarrollo de las prácticas del *ComeCocos Web* y del *ComeCocos Multijugador*.

Context 2d

Es el punto de partida para utilizar las propiedades de canvas. Es necesario acceder a la etiqueta canvas existente en el documento para obtener el contexto. El contexto puede ser 2D y 3D dependiendo del tipo de elementos que se quieren dibujar.

```
var canvas = document.getElementById('tutorial');
var ctx = canvas.getContext('2d');
```

Fragmento de Código 3.1: Acceso al contexto de Canvas.

Formas Disponibles

Por medio del contexto tenemos acceso a un conjunto de métodos que permiten dibujar primitivas predefinidas, que se explican a continuación.

1. Trazos

- `beginPath()` : Marcan el inicio de un nuevo trazo.
- `closePath()` : Marcan el final del trazo definido.
- `stroke()` : Dibuja el contorno de la forma.
- `fill()` : Dibuja una forma sólida rellenando el área del trazo.

2. Líneas: La función `lineTo(x, y)` permite dibujar líneas. Toma como punto de partida el último punto conocido y como punto final la coordenada(x,y) que se le pasa.

3. Movimiento: La función `moveTo(x, y)` permite moverse a un punto del lienzo para empezar a dibujar a partir de él.

```

1  function draw() {
2      var canvas = document.getElementById('canvas');
3      if (canvas.getContext) {
4          var ctx = canvas.getContext('2d');
5
6          // Filled triangle
7          ctx.beginPath();
8          ctx.moveTo(25, 25);
9          ctx.lineTo(105, 25);
10         ctx.lineTo(25, 105);
11         ctx.fill();
12
13         // Stroked triangle
14         ctx.beginPath();
15         ctx.moveTo(125, 125);
16         ctx.lineTo(125, 45);
17         ctx.lineTo(45, 125);
18         ctx.closePath();
19         ctx.stroke();
20     }
21 }
```



Figura 3.1: Ejemplo del dibujo de un trazo.

4. Rectángulos

- `fillRect(x, y, width, height)` : Dibuja un rectángulo lleno.
- `strokeRect(x, y, width, height)` : Dibuja el contorno de un rectángulo.

- `clearRect(x, y, width, height)`: Borra el área rectangular especificada.

```

1  function draw() {
2    var canvas = document.getElementById('canvas');
3    if (canvas.getContext) {
4      var ctx = canvas.getContext('2d');
5
6      ctx.fillRect(25, 25, 100, 100);
7      ctx.clearRect(45, 45, 60, 60);
8      ctx.strokeRect(50, 50, 50, 50);
9    }
10 }
```

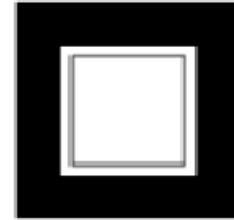


Figura 3.2: Ejemplo rectángulos canvas.

5. Arcos: Por medio de la función `arc(x, y, radio, angInit, angFin, true)` se dibuja una circunferencia o media circunferencia. Para realizar este proceso establece como centro las coordenadas (x,y), su tamaño depende del radio que se especifique y por ultimo establecemos el ángulo inicial y final que queremos.

Estilos y Colores

Para poder aplicar colores a los dibujos que se crean tenemos a nuestra disposición dos propiedades que podemos usar:

- `fillStyle`: Establece el color de relleno de la figura.
- `strokeStyle`: Establece el color del contorno de la figura.

Además de dibujar formas opacas en el lienzo, también podemos dibujar formas semitransparentes (o translúcidas). Esto se realiza estableciendo la propiedad `globalAlpha` o asignando un color semitransparente al estilo de trazo y/o de relleno.

- `globalAlpha`: Aplica el valor de transparencia especificado a todas las formas futuras del lienzo. El valor debe estar entre 0,0 (totalmente transparente) y 1,0 (completamente opaco).

Por otra parte se puede establecer el estilo de las líneas que componen las figuras, por medio las siguientes propiedades:

- `lineWidth`: Establece el ancho de las líneas.
- `lineCap`: Establece el aspecto de los extremos de las líneas. Estos atributos pueden ser: `butt`(extremos cuadrado), `round`(extremos redondeados) y `square()`

- `lineJoin`: Establece el aspecto de las esquinas donde se encuentran las líneas. Estos atributos pueden ser: `round()`, `bevel()` y `miter()`.

Texto

Permite generar cadenas de texto dentro de canvas por medio del método `fillText(text, x, y)` que recibe como parámetro el texto y la coordenada donde se dibujarán, figura 3.3.

- `fillText(text, x, y)` : Dibuja el texto dado en las coordenadas(x,y) del lienzo.
- `strokeText(text, x, y)` : Dibuja el texto dado en las coordenadas(x,y) del lienzo sin relleno.

Además permite establecer el estilo del texto por medio de las siguientes propiedades:

- `font` : Estilo del texto que se utiliza al dibujar el texto. Los valores permitidos son similares a las propiedades CSS font.
- `textAlign` : Alinea el texto. Los valores disponibles son: start, end, left, right o center.
- `direction` : Direccionalidad del texto.

```

1 | function draw() {
2 |   var ctx = document.getElementById('canvas').getContext('2d');
3 |   ctx.font = '48px serif';
4 |   ctx.fillText('Hello world', 10, 50);
5 |

```



Figura 3.3: Ejemplo texto canvas.

Imágenes

Para importar imágenes en el lienzo es necesario obtener la referencia de un objeto `HTMLImageElement` como fuente por medio de JavaScript y dibujarla en el lienzo por medio de la función `drawImage()`. Para ello, utilizamos el constructor `Image()` en el que definimos la dirección de la imagen en el atributo `src` y por medio del método `load` nos aseguraremos que la carga se ha completado.

Tras obtener la referencia a nuestra imagen podemos utilizar los métodos disponibles para cargar las imágenes en el lienzo. A continuación se muestran las variantes del método ya que se encuentra sobrecargado.

- `drawImage(img, x, y)` : Dibuja la imagen específica en las coordenadas(x,y) del lienzo.

- `drawImage(image, x, y, width, height)`: Dibuja la imagen específica en las coordenadas(x,y) estableciendo la escala en la que se dibuja por medio de los parámetros width y height, figura 3.4.

```

1 | function draw() {
2 |   var ctx = document.getElementById('canvas').getContext('2d');
3 |   var img = new Image();
4 |   img.onload = function() {
5 |     for (var i = 0; i < 4; i++) {
6 |       for (var j = 0; j < 3; j++) {
7 |         ctx.drawImage(img, j * 50, i * 38, 50, 38);
8 |       }
9 |     }
10 |   };
11 |   img.src = 'https://mdn.mozilla.org/files/5397/rhino.jpg';
12 |

```

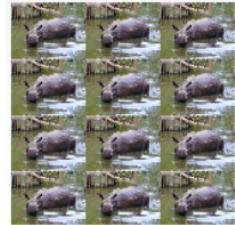


Figura 3.4: Ejemplo escalado de imagen canvas.

- `drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)`: Dada una imagen, esta función toma el área de la imagen de origen por el rectángulo cuya esquina superior izquierda es (sx,sy) y cuya anchura y la altura son sWidth y sHeight, colocándolo en el lienzo en las coordenadas(dx,dy) y escalado por medio de dWidth y dHeight, figura 3.5.

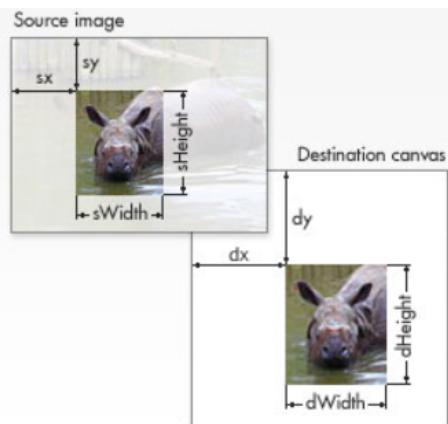


Figura 3.5: Ejemplo recorte imagen canvas.

Transformaciones

Antes de indicar las transformaciones disponibles es necesario conocer los métodos `save()` y `restore()`. El primero se utiliza para guardar el estado del lienzo

antes de aplicar un cambio en el lienzo mientras que el otro se encarga de recuperar el estado del lienzo guardado.

Ahora es momento de presentar las distintas transformaciones que se pueden utilizar

- `translate(x, y)` : Mueve el lienzo y su origen a la coordenada (x,y).
- `rotate(angle)` : Gira el lienzo en sentido a las agujas del reloj hasta encontrar el ángulo (radianes) indicado.
- `scale(x, y)` : Escala x e y unidades del lienzo. Se tratan de números reales, si son menores a la unidad disminuyen el tamaño en caso contrario aumentan el tamaño.

3.1.2. Media

El nuevo estándar HTML5 incorpora etiquetas que permiten incorporar contenido multimedia [13] en la web de forma nativa, antes era necesario utilizar plug-ins externos. La etiqueta `<video>` permite incrustar un vídeo en la web e incorpora la posibilidad de establecer múltiples formatos ya que no todos los formatos de vídeo son compatibles en cada uno de los navegadores.

- Normalmente, un contenedor WebM empaqueta audio Ogg Vorbis con vídeo VP8 / VP9. Esto es soportado principalmente por Firefox y Chrome.
- Un contenedor MP4 suele empaquetar audio AAC o MP3 con vídeo H.264. Esto se utiliza principalmente en Internet Explorer y Safari.
- El contenedor Ogg, más antiguo, tiende a ir con Ogg Vorbis audio y vídeo Ogg Theora. Fue apoyado principalmente en Firefox y Chrome, pero básicamente ha sido reemplazado por el mejor formato WebM.

Las principales características de las que dispone la etiqueta `<video>` son las siguientes:

- `width/height` : Tamaño del vídeo
- `autoplay` : Indica que una vez se ha cargado el elemento empiece la reproducción automáticamente.
- `loop` : Se crea un bucle en el que se repite indefinidamente el vídeo.
- `muted` : Desactiva el sonido.
- `poster` : Toma la dirección de una imagen que se muestra antes de empezar la reproducción del vídeo.
- `src` : Contiene la ruta de acceso al vídeo

- **controls**: Permite a los usuarios pausar, reproducir o ajustar el volumen del vídeo.

Además tenemos la etiqueta `<audio>` que permite incrustar audio en la web. Su funcionalidad es similar al de la etiqueta `<video>` aunque no presenta un interfaz gráfico.

El primer elemento se emplea en el desarrollo del *Sitio Web y Videoconferencia con WebRTC* mientras el segundo elemento además se emplea en el *ComeCocos, ComeCocos Multijugador*.

3.1.3. WebSockets

Internet se ha creado a partir del paradigma solicitud desde el cliente / respuesta por parte del servidor de HTTP. Un cliente carga una página web, se cierra la conexión y no ocurre nada hasta que el usuario hace clic en un enlace o envía un formulario.

Hace algún tiempo que existen técnicas que ofrecen al servidor llevar la iniciativa, como por ejemplo Comet. Uno de las técnicas más comunes para crear la ilusión de una conexión iniciada por el servidor se denomina Long Polling. Con el Long Polling, el cliente abre una conexión HTTP con el servidor, el cual la mantiene abierta hasta que se envíe una respuesta. Cada vez que el servidor tenga datos nuevos los enviará como respuesta a cada petición pendiente.

WebSockets [15] ofrece estas características de modo nativo sin necesidad de utilizar trucos como en caso de Comet permitiendo al servidor llevar la iniciativa sin problemas, figura 3.6, ofreciendo las siguientes características:

1. **Conexión bidireccional**: Esta conexión se produce en tiempo real y se mantiene permanentemente abierta entre servidor y cliente hasta que se cierre de manera explícita.
2. **Gran rendimiento y escalabilidad**: Si un socket está abierto, el servidor puede enviar datos a todos los clientes conectados al socket, sin tener que estar constantemente procesando peticiones.
3. **Baja latencia**: Como el socket está siempre abierto y escuchando, los datos son enviados inmediatamente desde el servidor al navegador.
4. **Eficiencia en la transmisión de datos**: 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.

Funcionamiento del API

El cliente establece una conexión WebSockets [16] a través de un proceso conocido como *Handshake WebSocket*. Este proceso se inicia con una solicitud HTTP normal al servidor que incluye el campo `Upgrade` para informar al servidor que se desea establecer una conexión WebSocket.

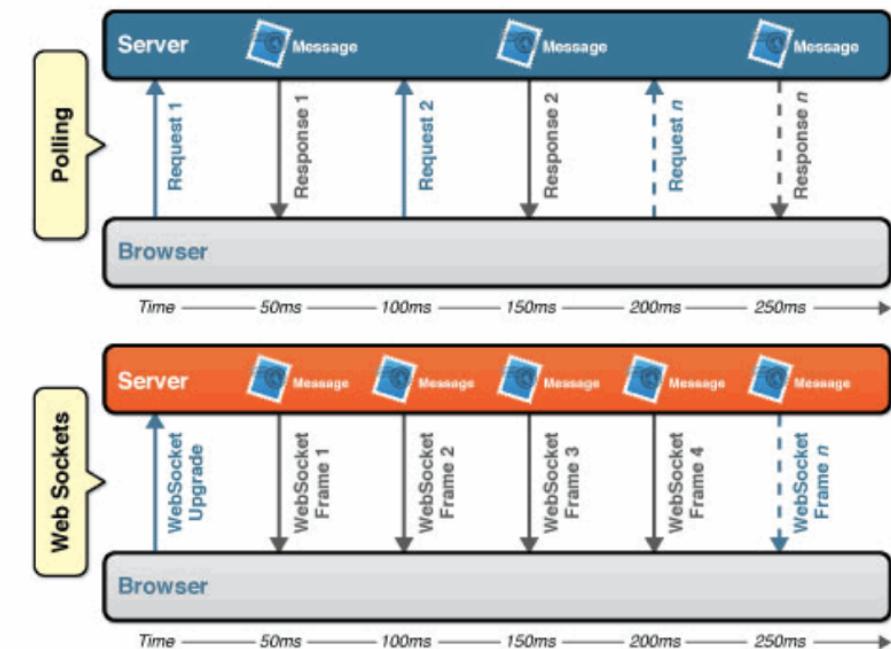


Figura 3.6: Comparación Polling-WebSockets.

```
GET ws://websocket.example.com/ HTTP/1.1
Origin: http://example.com
Connection: Upgrade
Host: websocket.example.com
Upgrade: websocket
```

Fragmento de Código 3.2: Petición conexión WebSockets.

Si el servidor soporta el protocolo WebSockets, lo comunica a través del campo **Upgrade** en la respuesta.

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Wed, 16 Oct 2013 10:07:34 GMT
Connection: Upgrade
Upgrade: WebSocket
```

Fragmento de Código 3.3: Respuesta conexión WebSockets.

Ahora que el proceso *Handshake* se ha completado, la conexión inicial de HTTP se sustituye por una conexión WebSockets que utiliza la misma conexión TCP / IP subyacente. En este punto, cualquiera de las partes puede iniciar el envío de datos.

Librería Socket.io

Socket.IO [?] es una librería de JavaScript que facilita el desarrollo de aplicaciones basadas en Websockets tanto en el cliente como en el servidor.

En lado servidor (NodeJS) se importa la librería por medio de la sentencia require ()

```
var io = require('socket.io')(server);
```

Fragmento de Código 3.4: Llamada librería Socket.IO Server.

Para iniciar la conexión WebSockets se emplea el evento io.on ('connection', function ())

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

Fragmento de Código 3.5: Comprobación conexión WebSockets Server.

En el lado cliente es necesario incluir el script socket.io.js en el fichero HTML de proyecto.

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

Fragmento de Código 3.6: Instancia librería Socket.IO cliente.

Para establecer la conexión Websocket utilizamos el método io.connect al que se le pasa la url donde se encuentra el servidor con el que se quiere establecer la conexión.

```
var socket = io.connect('http://localhost:8080');
```

Fragmento de Código 3.7: Creación conexión WebSocket cliente.

Establecida la conexión entre el cliente y servidor pueden enviar mensajes¹. El método emit permite el envío de mensajes recibiendo como parámetros el nombre del mensaje y los datos que se envían. El receptor del mensaje utiliza el método on para definir un evento con el nombre del mensaje que espera recibir y un callback que se encarga de obtener los datos del mensaje.

```
//cliente
socket.emit('new-message', 'Hola');

//server
socket.on('new-message', function(data) {
    console.log(data)
});
```

Fragmento de Código 3.8: Ejemplo Envió-Recepción mensaje con WebSockets.

La versión 1.0 de la librería se utiliza en el desarrollo de las prácticas del *Come-Cocos Multijugador* y de *Videoconferencia con WebRTC*.

¹<https://socket.io/>

3.1.4. API File

El tratamiento de ficheros locales en la maquina cliente desde los navegadores de forma nativa no fue posible hasta la aparición de API File [14]. La API requiere de una etiqueta `<input>` para seleccionar el archivo con el que se quiere trabajar. A continuación JavaScript se encarga de obtener la información del documento a través del objeto `FileReader()`.

```

<!DOCTYPE html>
<head>
<meta charset="utf-8">
<title>File API</title>
<script type="text/javascript">
function processFiles(file){
    var files = file[0];
    var reader = new FileReader();
    reader.onload = function (e) {
        /*
        e.result tiene el resultado de la
        lectura del fichero
        */
    };
    reader.readAsArrayBuffer(files);
}
</script>
</head>
<body>
Select a text file:
<input type="file" id="fileInput" onchange="processFiles(this.
files)">
</body>
</html>

```

Fragmento de Código 3.9: Ejemplo API File.

Es necesario especificar el modo en el que la API va a leer la información. A continuación se resumen los métodos disponibles:

- `reader.readAsText()` : Permite leer archivos de texto y recibe dos parámetros. El primer parámetro es el objeto file o Blob que se va a leer y el segundo parámetro se utiliza para especificar la codificación del archivo.
- `reader.readAsDataURL()` : Permite leer un File o Blob y generar una URL de datos . Esto es básicamente una cadena base64 de los datos del archivo. Se puede utilizar esta URL datos para cosas como el establecimiento de la propiedad `src` de una imagen.
- `reader.readAsBinaryString()` : Permite leer cualquier tipo de archivo. El método devuelve los datos binarios sin formato.

- `reader.readAsArrayBuffer()` : Permite leer un File o Blob y obtener como resultado un ArrayBuffer, es decir, un buffer de datos binarios de longitud fija.

Esta API se utiliza en el desarrollo de la práctica de *Videoconferencia con WebRTC*.

3.2. Lenguaje JavaScript

JavaScript [18] es un lenguaje de programación interpretado que permite crear script con eventos, clases y acciones para la ejecución de aplicaciones Internet del lado del cliente. Los usuarios no leerán únicamente las páginas de forma pasiva sino que adquieran un carácter interactivo permitiendo cambiar las páginas dentro de una aplicación: poner botones, cuadros de texto, código para hacer una calculadora, un editor de texto, un juego, o cualquier otra cosa de modo que se ejecuta cierto código desde el propio navegador.

Propiedades

- Se interpreta en el ordenador que recibe el programa, no se compila.
- Tiene una programación orientada a objetos. El código de los objetos está pre-definido y es expandible. No usa clases ni herencia.
- Tipicamente el código está incluido en los documentos HTML.
- No se declaran los tipos de variables.
- Ejecución dinámica: los programas y funciones no se chequean hasta que se ejecutan.
- Parte de los programas de JavaScript se ejecuta cuando sucede un evento.

3.3. Biblioteca JQuery

JQuery [19][20] es un entorno Javascript que sirve como base para la programación avanzada de aplicaciones del lado del cliente aportando una serie de funciones o códigos para realizar tareas habituales. Sus principales características son:

- efectos dinámicos.
- aplicaciones que hacen uso de Ajax.
- manipular el árbol DOM.
- manejo de eventos.
- desarrollar animaciones

- simplifica la manera de interactuar con los documentos HTML

La versión 3.2.0 de la librería se emplea en el desarrollo de todas las prácticas ya que nos permite simplificar ciertas tareas.

3.4. Biblioteca Bootstrap

Bootstrap [21][22] es un entorno (front-end) de twitter para desarrollo de aplicaciones web. Se basa en un sistema de grid de 12 columnas que escalan adecuadamente a medida que aumenta el tamaño del dispositivo o la ventana de visualización. Además contiene elementos de diseño básicos de HTML y CSS, como pueden ser barras de navegación, plantillas predefinidas, botones, desplegables entre otras extensiones de JavaScript. Sus principales características son:

- Sencillo y ligero
- Basado en los últimos estándares de desarrollo de Web
- Curva de aprendizaje baja
- Compatible con todos los navegadores habituales
- Responsive web design.

La versión 3.3.7 de este entorno se utiliza para diseñar la apariencia de todas las prácticas.

3.5. Entorno de servidor NodeJS

Es un proyecto de software libre creado por Ryan Dahl a principios de 2009 orientado a la creación de aplicaciones para Internet, principalmente Web. La idea empezó a gestarse a partir de otro proyecto para el entorno Ruby sobre Rails, un pequeño y rápido servidor web llamado Ebb, que evolucionó a una librería en C.

Una de las razones de la evolución del proyecto desde Ruby a C, y luego de C a JavaScript fue el objetivo de realizar un sistema en que la Entrada/Salida fuera enteramente no bloqueante que es esencial para obtener un alto rendimiento. Según [23] “*Node.js es una plataforma construida encima del entorno de ejecución javascript de Chrome para fácilmente construir rápidas, escalables aplicaciones de red. Node.js usa un modelo de E/S no bloqueante dirigido por eventos que lo hace ligero y eficiente, perfecto para aplicaciones data-intensive en tiempo real*”

Fácil desarrollo y escalabilidad

Una de las ventajas de emplear JavaScript como lenguaje para las aplicaciones Node es que, al tratarse de un lenguaje con una curva de aprendizaje suave permite

desarrollar aplicaciones rápidamente con sólo tener unas nociones básicas de las características de este lenguaje.

Uno de los aspectos con mayor impacto en cuanto a la escalabilidad es el diseño del sistema. Este es uno de los punto fuertes de Node ya que su arquitectura y la forma de desarrollar sus aplicaciones hacen que se cumplan los principios básicos de escalabilidad.

E/S no bloqueante por eventos

Uno de los puntos críticos es el cuello de botella que afecta en alto grado al rendimiento de cualquier sistema, en especial aquellos que hacen un uso intensivo de operaciones de Entrada/Salida con ficheros y dispositivos. Para solventar este problema Node utiliza un modelo de concurrencia basado en eventos que implicar utilizar los siguientes requisitos:

- Necesidad de un bucle de procesado de eventos que se tratará como un único proceso y que sólo ejecuta un manejador, o *callback*, a la vez.
- Emplear un lenguaje que se adapte a este modelo, como es el caso de JavaScript ya que su intérprete se basa en un modelo idéntico.

Ligero y Eficiente

Node es una fina capa de software entre el sistema operativo y la aplicación escrita ya que con su arquitectura se persigue velocidad y eficiencia. Centrado en este propósito descarta emplear un modelo *multihead* para manejar las distintas conexiones ya que el coste es muy elevado. Se busca entonces una solución de alto rendimiento que permita realizar operaciones Entrada/Salida no bloqueantes delegando en el sistema operativo y coordinarlo a través de uno o varios bucles de eventos.

Perfecto para aplicaciones en tiempo real

Node encaja con los requisitos que exigen las aplicaciones en tiempo real flexibles. De acuerdo a su capacidad de manejar un alto número de conexiones y procesar un enorme número de operaciones de Entrada/Salida muy rápido se puede afirmar que Node encaja perfectamente si se requiere:

- **Interfaces ligeros REST/JSON:** Su modelo de Entrada/Salida para atender peticiones REST junto al soporte nativo JSON lo hacen óptimo como capa superior de fuente de datos como base de datos.
- **Aplicaciones mono página:** En las que la interacción del cliente con el servidor se realiza por medio de peticiones Ajax. El uso de Ajax puede producir una avalancha de peticiones que el servidor tiene que ser capaz de procesar es aquí donde Node cumple con éxito.

- **Datos por streaming:** Al tratarse de conexiones HTTP como *streams*, permite procesar ficheros al vuelo.
- **Comunicación:** Aplicaciones de mensajería instantánea o web en tiempo real, e incluso, juegos multijugador.

La versión 6.10 se utiliza en el desarrollo del las prácticas del *ComeCocos Multijugador y Videoconferencia con WebRTC*.

3.6. Base de datos en aplicaciones web

El objetivo principal de las BBDD [24] es unificar los datos que se manejan y los programas o aplicaciones que los manejan. Antiguamente los programas se codificaban junto con los datos lo que desembocaba en una dependencia de los programas respecto a los datos. Además, cada aplicación utiliza ficheros que pueden ser comunes a otros sectores de la misma aplicación lo que producía redundancia en la información.

Con las BBDD se busca independizar los datos que residen en memoria y las aplicaciones que manipulan la información mediante un sistema gestor de BBDD. Por lo tanto una base de datos pretende conseguir a través del Sistema Gestor de Bases de Datos (SGBD):

- **Independencia de datos:** Cambios en la estructura de la Base de Datos no modifican las aplicaciones.
- **Integridad de los datos:** Los datos han de ser siempre correctos. Se establecen una serie de restricciones (reglas de validación) sobre los datos.
- **Seguridad de los datos:** Control de acceso a los datos para evitar manipulaciones no deseadas.

3.6.1. MySQL

MySQL es uno de los SGBD más conocido y usado de código libre, desarrollado por la empresa MySQL AB que lo desarrolla bajo licencia de código libre. Se trata de un SGBD extremadamente rápido aunque no ofrece las mismas prestaciones que otras bases de datos, lo compensa con un rendimiento excelente. Alguno de los comandos de los que dispone son:

1. **Uso BBDD:** La sentencia `USE nombreBd` permite seleccionar la BBDD con la que se quiere trabajar.
2. **Visualización:** Para ver todas las BBDD se utiliza la sentencia `SHOW DATABASES` mientras que para las tablas de una BBDD se utiliza `SHOW TABLES`.
3. **Búsqueda:** Una de las sentencias de este tipo es `SELECT * FROM nombreTabla` que permite visualizar el contenido de una determinada tabla.



Figura 3.7: Ejemplo Base de Datos.

4. **Crear y Eliminar:** La sentencia `CREATE DATABASE nombreBd` crea y la sentencia `DROP DATABASE nombreBd` elimina una BBDD.

La versión 5.0 se ha utilizado para desarrollar la práctica Sitio Web de una tienda.

3.7. Entorno de servidor Django

Consideremos el diseño de una aplicación Web escrita usando el estándar Common Gateway Interface (CGI), una forma popular de escribir aplicaciones Web en el año 1998. En esa época, cuando escribías una aplicación CGI se desarrollaba todas las tareas por uno mismo. Este enfoque es válido si la aplicación solo utiliza un fichero pero a medida que una aplicación Web crece aparecen una serie de problemas:

- ¿Qué sucede cuando múltiples páginas necesitan conectarse a la BBDD? El código de conexión a la BBDD no debería estar en cada uno de los script ya que la mejor forma de hacerlo es refactorizarlo en una función compartida.
- ¿Qué sucede cuando este código es reutilizado en múltiples entornos, con BBDD y contraseñas diferentes? En ese punto, se vuelve esencial alguna configuración específica del entorno.
- ¿Qué sucede cuando un diseñador Web no tiene experiencia programando y desea rediseñar la página? Lo ideal sería que la lógica de la página esté separada del código HTML de la página.

Precisamente estos son los problemas que un entorno Web intenta resolver por medio de una infraestructura de programación para las aplicaciones permitiendo escribir código limpio y de fácil mantenimiento. En nuestro caso utilizamos Django [27].

Patrón de diseño MVC

Django sigue el patrón de diseño Modelo-Vista-Controlador. Este patrón permite que el código que interactúa con los datos (modelo) esté separado de la asignación de rutas (controlador) y a su vez separado del interfaz del usuario (vista). La principal ventaja de este enfoque es el acoplamiento débil entre sí, es decir, cada pieza de la aplicación web tiene un único propósito por lo que puede ser modificada independientemente sin afectar a las otras piezas.

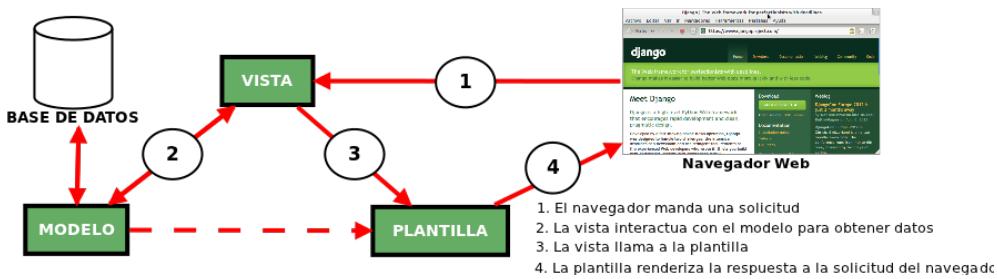


Figura 3.8: Esquema MVC Django.

Models

Los modelos se representan como una clase Python dentro del fichero *models.py* en el que se define cada uno de los elementos por medio del objeto *models*².

```
from django.db import models
class Artista(models.Model):
    name = models.CharField(max_length=100, null=True)
    videos = models.ManyToManyField(Videos)
    galeria = models.ManyToManyField(Imagenes)
```

Fragmento de Código 3.10: Ejemplo de un Modelo.

En este punto destacamos la posibilidad de establecer relación entre los distintos modelos a través de los siguientes atributos:

- *models.ForeignKey*: relación 1 a 1.
- *models.ManyToManyField*: relación 1 a n.

Tras migrar la información del fichero se genera tantas tablas como clases existan.

²fuente: <https://docs.djangoproject.com/en/1.10/ref/models/fields/>

Vistas

Una vista es una función en Python que contiene la lógica para interactuar con la BBDD definida dentro del fichero *views.py*. Para obtener la información Django permite emplear filtro definidos en Python sin necesidad de programar en SQL.

```
from django.http import HttpResponse
def EventSelect(request, evento):
    event = Evento.objects.filter(name__startswith=evento)
    return render(request, 'IndexEvent.html', {'event':event})
def IndexView(request):
    list_video=Videos.objects.all()
    return render(request, 'fullVideo.html', {'list_video':list_video})
```

Fragmento de Código 3.11: Ejemplo de vistas.

Las funciones se pueden dividir en dos grupos:

- **Dinámicas:** Son aquellas que reciben el parámetro `request` y un parámetro adicional a través de la url.
- **Estáticas:** Son aquellas que reciben solo el parámetro `request`.

Controlador

Un controlador se representa como una dupla en Python dentro del fichero *urls.py* que define las distintas rutas de acceso a la aplicación y la vista asociada.

```
# urls.py
from django.conf.urls.defaults import patterns, url
from .views import ....
urlpatterns = patterns('',
    url(r'^index/$', views.MainPage),
    url(r'^eventos/(?P<evento>\w{1,50})/$', views.EventSelect),
)
```

Fragmento de Código 3.12: Ejemplo de url's.

Al igual que en las vistas las URLs pueden ser de dos tipos:

- **Dinámicas:** la ruta esta formado por texto y un parámetro adicional.
- **Estáticas:** la ruta se define como una cadena de texto.

Plantilla

Una plantilla es un fichero HTML enriquecido con un lenguaje de etiquetas adicional que permite renderizar la información enviadas por la vista correspondiente. Para plasmar esta información utilizamos el lenguaje de plantillas³ que es similar a

³<https://docs.djangoproject.com/en/1.10/ref/templates/>

un lenguaje de programación ya que posee bucles, variables de plantillas asociadas a variables Python, sentencias condicionales entre otras características.

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
<p>Dia y hora actual: {{fechahora\_actual}}</p>
{ %< for cantante in evento.artistas.all %}
<p>{{cantante.name}}</p>
</div>
{ % endfor %
{ % block info %}{ % endblock %
</body>
</html>
```

Fragmento de Código 3.13: Ejemplo de plantilla.

El lenguaje de plantillas define los siguientes modos de acceso a la información:

- El contenido entre {{ }} muestra el valor de una variable Python en la plantilla.
- El contenido entre { %% } hace referencia a las etiquetas del lenguaje de plantilla.
- Se puede definir bloques por medio de las etiquetas { % block namesection %} y { % endblock %}, que se utilizan al establecer herencia entre las plantillas.

La versión 1.9 de este entorno se utiliza para desarrollar la práctica *Sitio Web de una tienda*.

3.8. WebRTC

WebRTC es una tecnología que permite establecer comunicación entre (navegadores) e intercambiar audio, vídeos y archivos. Para construir una aplicación de este tipo desde cero, sería necesario una gran cantidad bibliotecas que se ocupan de problemas típicos como la pérdida de datos, caída de conexiones y atravesar NATs pero WebRTC [17] incorpora de forma nativa las soluciones a estos problemas que se detallan a continuación.

Servidor Señalización

Para conectarse con otro usuario es necesario saber la dirección IP de su dispositivo. Tan pronto como los dispositivos saben encontrarse a través de Internet, comienza el intercambio de datos sobre qué protocolos y códecs soporta cada dispositivo. Este proceso recibe el nombre señalización que sigue el esquema de la figura 3.9 explicado a continuación.

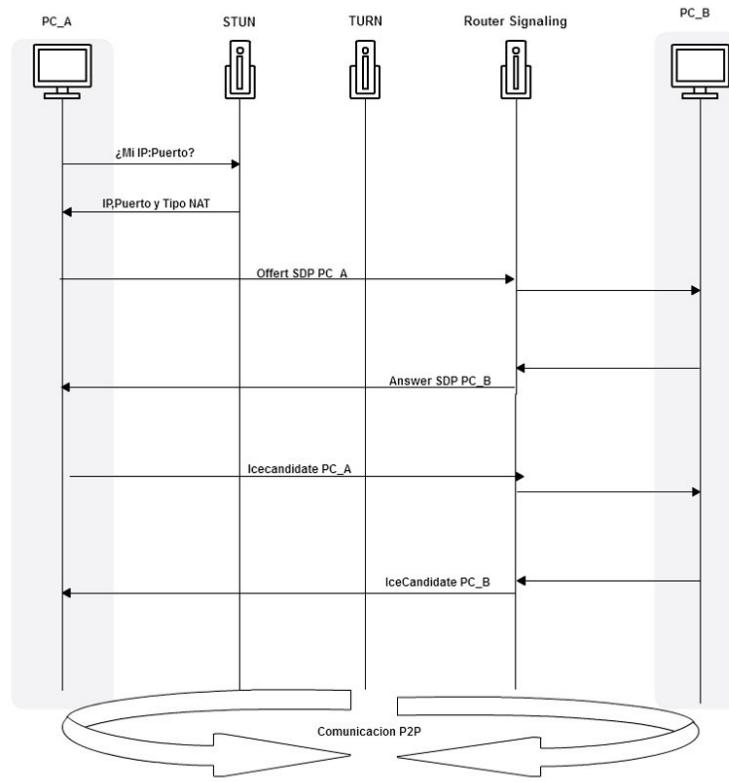


Figura 3.9: Proceso del Servidor Señalización.

1. Crea una lista de candidatos potenciales para una conexión entre iguales.
2. El usuario o una aplicación selecciona un usuario con el que establecer una conexión enviando un mensaje que contiene la descripción de la sesión de comunicación (SDP). Este proceso recibe el nombre de Oferta.
3. La capa de señalización notifica el mensaje al otro usuario.
4. Al recibir la oferta el usuario contrario genera un mensaje con su descripción de sesión. Este proceso recibe el nombre de Respuesta.
5. Ambos usuarios guardan el mensaje SDP dentro de su objeto RTCPeerConnection como una descripción de sesión remota.
6. Además ambos usuarios intercambian información de ubicación por medio de los ICECandidate.
7. Finalmente la conexión se establece con éxito o falla.

La especificación de WebRTC no define ningún estándar sobre el intercambio de información, así que se puede utilizar cualquier protocolo o tecnología en el diseño

de la etapa de señalización. En las siguiente sección explicamos los protocolos que a día de hoy se utilizan con mayor afluencia.

Conexión entre pares

SDP

SDP (Session Description Protocol) es una parte importante del WebRTC. Se trata de un protocolo utilizado para describir las sesiones de comunicación que no entrega los datos de los medios sino que se utiliza para la negociación de codecs de audio y vídeo, topologías de red y otra información de los dispositivos. Se puede definir SDP como una cadena de datos que contiene conjuntos de pares clave-valor, separados por saltos de línea: Key = value como se puede ver en la figura 3.10.

```

descripcion de la session
v=0
o=mozilla...THIS_IS_SDPARTA-46.0.1 4867651871611090737 0 IN IP4 0.0.0.0
S=-
t=0 0
a=fingerprint:sha-256
22:F9:81:E5:8D:1F:8A:FF:D7:11:17:79:F9:8A:A9:7A:C3:E6:33:9B:BE:3C:30:41:40:EF:5A:7D:3E:7D:CE:A6
a=group:BUNDLE sdparta_0 sdparta_1
a=ice-options:trickle
a=msid-semantic:WMS *
m=video 9 UDP/TLS/RTP/SAVPF 120 126 97
c=IN IP4 0.0.0.0
a=sendrecv
a=fmtp:126 profile-level-id=42e01f;level-asymmetry-allowed=1;packetization-mode=1
a=fmtp:97 profile-level-id=42e01f;level-asymmetry-allowed=1
a=fmtp:120 max-fs=12288;max-fr=60
a=ice-pwd:2a534092af4f34a5ba94da255ee4984f
a=ice-ufrag:d2d5c9ac
a=mid:sdparta_0
a=msid:{c0db156c-5c3d-44b1-a42e-2d757d297b92} {7a01e077-b75f-4b77-93ff-6d5ea8d20e2b}
a=rtcp-fb:120 nack
a=rtcp-fb:120 nack pli
a=rtcp-fb:120 ccm fir
a=rtcp-fb:126 nack
a=rtcp-fb:126 nack pli
a=rtcp-fb:126 ccm fir
a=rtcp-fb:97 nack
a=rtcp-fb:97 nack pli
a=rtcp-fb:97 ccm fir
a=rtcp-mux
a=rtpmap:120 VP8/90000
a=rtpmap:126 H264/90000
a=rtpmap:97 H264/90000
a=setup:actpass
a=ssrc:3704999849 cname:{a79cf374-f9bd-4378-9226-3d0f54982415}
m=application 9 DTLS/SCTP 5000
c=IN IP4 0.0.0.0
a=sendrecv
a=ice-pwd:2a534092af4f34a5ba94da255ee4984f
a=ice-ufrag:d2d5c9ac
a=mid:sdparta_1
a=sctpmap:5000 webrtc-datachannel 256
a=setup:actpass
a=ssrc:2709130962 cname:{a79cf374-f9bd-4378-9226-3d0f54982415}

```

Figura 3.10: Ejemplo Protocolo SDP.

Este protocolo es la primera parte de la conexión entre pares ya que dicha in-

formación se tiene que intercambiar por medio del canal de señalización para finalmente establecer la conexión.

STUN

STUN (Session Traversal Utilities for NAT) es un protocolo de red de tipo cliente/servidor que permite a clientes conocer su dirección IP pública y el tipo de NAT sobre el que se encuentran. Esta información es usada para configurar una comunicación entre pares que se encuentren detrás *router* NAT.

STUN es usado principalmente por teléfonos o software VoIP. Un cliente envía una petición a un servidor STUN. El servidor STUN informa entonces al cliente de la IP pública y qué puerto ha sido abierto por NAT para el tráfico entrante a la red del cliente, como se puede ver en la figura 3.11.

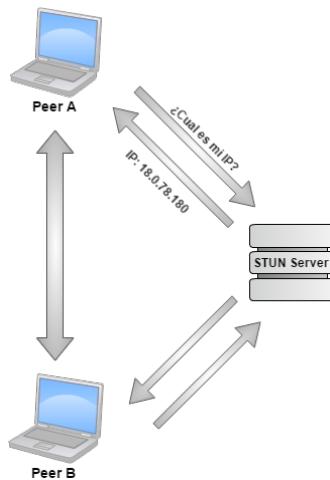


Figura 3.11: Ejemplo petición-respuesta STUN.

Una vez el cliente ha descubierto su dirección pública, puede comunicar ésta a sus pares. STUN es útil como complemento a protocolos basado en SDP como SIP que utiliza paquetes UDP para la señalización de tráfico de sonido, vídeo y texto sobre Internet, pero no permite establecer la comunicación cuando los extremos se encuentran detrás de NATs.

TURN

En ocasiones hay cortafuegos que no permite ningún tráfico basado en STUN entre pares. En estas situaciones, es necesario utilizar un nodo intermedio que actúa como distribuidor por lo que se utiliza TURN (Traversal Using Relays around NAT)

TURN permite encontrar el camino más eficaz para comunicarse con otros clientes y en caso de no encontrarse ningún camino directo retransmitir los flujos de voz y datos, como se puede ver en la figura 3.12.

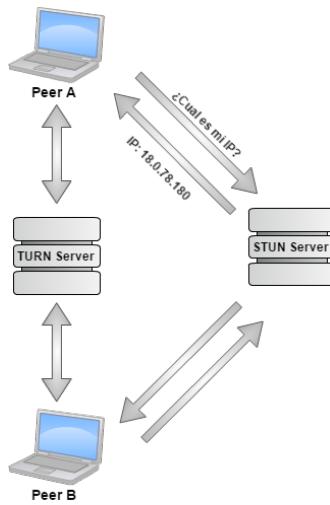


Figura 3.12: Ejemplo petición-respuesta TURN.

ICE

ICE (Interactive Connectivity Establishment) es el mecanismo que utiliza WebRTC para obtener información sobre la red de cada uno de los pares que intervienen en una comunicación WebRTC apoyándose en STUN y TURN. ICE encuentra y prueba en orden un rango de direcciones que funcionarán para ambos usuarios. Al empezar no sabe nada sobre la red de cada usuario por lo que a través de un conjunto de etapas de forma incremental descubre cómo se configura la red de cada cliente. La tarea principal es encontrar suficiente información de cada red para poder tener éxito en la conexión.

Envío de datos multimedia entre los pares

SCTP

SCTP (Stream Control Transmission Protocol) es un protocolo para el control de transmisión que funciona a la altura de la capa de aplicación. Fue definido por el grupo SIGTRAN de IETF en el año 2000. Este protocolo posee características similares a UDP y TCP, pero además reúne otras características y otras funcionalidades prácticas como las que se mencionan a continuación:

- Existen dos modos de transporte: fiable y no fiable.
- La capa de transporte está protegida
- Cuando se transportan mensajes de datos, se permite que se descompongan y vuelvan a montarse en el otro lado.
- El control del flujo y la congestión se proporciona a través de la capa de transporte.

Por lo que SCTP es una alternativa a los protocolos de transporte TCP y UDP pues provee confiabilidad, control de flujo y secuenciación como TCP. Sin embargo, SCTP opcionalmente permite el envío de mensajes fuera de orden y a diferencia de TCP, SCTP es un protocolo orientado al mensaje (similar al envío de datagramas UDP). Este protocolo se emplea al crear un objeto `RTCDataChannel` en la conexión WebRTC, a través del cual se crea un canal de comunicación bidireccional para enviar datos (blob) entre pares.

Una vez descrito el establecimiento de conexión entre pares WebRTC y el envío de datos multimedia entre ambos se pasa a describir los tres interfaces de programación que ofrece WebRTC para integrar su tecnología en aplicaciones web concretas.

API GetUserMedia

Es una API diseñada para acceder fácilmente a los flujos de datos de las cámaras y micrófonos desde el navegador web. El método `getUserMedia()` es la forma principal para acceder al flujo de datos de los dispositivos.

Método

1. `navigator.getUserMedia(constraints, successCallback, errorCallback)`: Pide al usuario permiso para usar los dispositivo multimedia como una cámara o micrófono definidos en el objeto `constraints`. Si el usuario concede este permiso, el `successCallback` es invocado en caso contrario se invoca `errorCallback`.

Características

1. Un flujo de *stream* en tiempo real está representado por un objeto *stream* en forma de vídeo o audio.
2. Proporciona un nivel de seguridad ya que pide permiso a los usuarios humanos acceso a los elementos multimedia locales.

API RTCPeerConection

`RTCPeerConnection` es el núcleo de la conexión peer-to-peer entre cada uno de los navegadores. Para crear el objeto `RTCPeerConnection` se realiza la siguiente llamada.

```
var pc = new RTCPeerConnection(config);
```

Fragmento de Código 3.14: Instancia RTCPeerConnection.

Donde el argumento `config` contiene al menos la clave, `iceServers`, que es una matriz de objetos URL que contiene información sobre los servidores STUN y TURN utilizados durante la búsqueda de los candidatos ICE.

Propiedades

1. `RTCPeerConnection.localDescription` (read only) : Devuelve un objeto `RTCSessionDescription` que describe la sesión local.
2. `RTCPeerConnection.remoteDescription` (read only) : Devuelve un objeto `RTCSessionDescription` que describe la sesión remota.

Controladores de eventos

1. `RTCPeerConnection.onaddstream`: Se activa cada vez que se añade un objeto `MediaStream` por el par remoto.
2. `RTCPeerConnection.ondatachannel`: Se activa cuando se incluye el canal de datos en el par remoto.
3. `RTCPeerConnection.onicecandidate`: Se activa cuando se agrega un objeto `RTCIceCandidate`.
4. `RTCPeerConnection.onremovestream`: Se activa cuando se quita un objeto `MediaStream` de la conexión.

Métodos

1. `RTCPeerConnection()` : Devuelve un nuevo objeto `RTCPeerConnection`.
2. `RTCPeerConnection.createOffer(HandlerOffer, HandlerError, Options)` : Crea una solicitud de oferta para encontrar un peer remoto. Los dos primeros parámetros de este método son los retornos de la función de éxito y error. El tercer parámetro es opcional.
3. `RTCPeerConnection.createAnswer(HandlerAnswer, HandlerError, Options)` : Crea una respuesta a la oferta recibida por el peer remoto durante el proceso de negociación (oferta/respuesta). Los dos primeros parámetros de este método son los retornos de la función de éxito y error. El tercer parámetro es opcional.
4. `RTCPeerConnection.setLocalDescription()` : Cambia la descripción de la conexión local. El método toma tres parámetros, el objeto `RTCSessionDescription` y los retornos de la función de éxito y fallo.
5. `RTCPeerConnection.setRemoteDescription()` : Cambia la descripción de la conexión remota. El método toma tres parámetros, el objeto `RTCSessionDescription` y los retornos de la función de éxito y fallo.
6. `RTCPeerConnection.addStream()` : Añade un objeto `MediaStream` como una fuente local de vídeo o audio.
7. `RTCPeerConnection.addIceCandidate()` : proporciona un candidato remoto al agente ICE.

8. `RTCPeerConnection.createDataChannel()` : crea un nuevo objeto `RTCDataChannel`.
9. `RTCPeerConnection.close()` : finaliza la conexión.

API RTCDataChannel

WebRTC no sólo permite transferir secuencias de audio y vídeo, sino también datos genéricos de modo eficiente y bidireccional por medio del objeto `RTCDataChannel`.

Propiedades

1. `RTCDataChannel.label (read only)` : Devuelve el nombre del canal de datos.
2. `RTCDataChannel.protocol (read only)` : Devuelve una cadena con el nombre de subprotocalo utilizado para este canal.
3. `RTCDataChannel.readyState (read only)` : Devuelve el estado de la conexión.
Los posibles valores:
 - `connecting`: Indica que la conexión aún no está activa.
 - `open`: Indica que la conexión se está ejecutando
 - `closing`: Indica que la conexión no se pudo establecer o se ha cerrado

Controladores de eventos

1. `RTCDataChannel.onopen` : Se activa cuando se ha establecido la conexión de datos.
2. `RTCDataChannel.onmessage` : Se activa cuando está disponible un mensaje en el canal de datos.
3. `RTCDataChannel.onclose` : Se activa cuando la conexión se ha cerrado.
4. `RTCDataChannel.onerror` : Se activa cuando se produce un fallo.

Métodos

1. `RTCDataChannel.close()` : Cierra el canal de datos.
2. `RTCDataChannel.send()` : Envía los datos pasados como parámetro a través del canal. Los datos pueden ser blob, cadenas de texto, un `ArrayBuffer` o un `ArrayBufferView`.

Se utiliza la versión 1.0 en el desarrollo de la práctica de Videoconferencia con WebRTC.

3.9. Web Services

Es un estándar de comunicación entre procesos diseñado para ser multiplataforma y multilenguaje, es decir, no importa el lenguaje ni plataforma donde se ejecute el *Web Services* [25]. Antiguamente se utilizaban estándares como DCOM (*Distributed Component Object Model*) y CORBA (Common Object Request Broker Architecture). Estos estándares presentaban graves problemas de configuración en entornos con cortafuegos ya que era imposible habilitar cierto puertos seguridad. Por lo que se prefería utilizar el puerto 80 de *HTTP*, que normalmente se encontraba habilitado debido al uso de navegadores y servidores Web. De esta forma *HTTP* se convirtió en el protocolo preferido para el transporte de mensajes de los *Web Services*.

Características

1. **Combinación:** Las operaciones de un servicio web pueden utilizar otros servicios web para sus operaciones.
2. **Patrones de comunicación**
 - **Petición-respuesta síncrona:** Invocamos al servicio y esperamos la respuesta a la petición.
 - **Comunicación asíncrona:** Se envía la petición y se continúa la ejecución.
 - **Mediante eventos:** El cliente se suscribe a eventos ofrecidos por el servicio.
3. **Desacoplamiento:** Se refiere a minimizar las dependencias entre los servicios para ofrecer una mayor flexibilidad en la arquitectura.
4. **Representación de mensajes**
 - **Textual:** SOAP representa los servicios y los mensajes en XML.
 - **Binario:** Los datos ocupan menos espacio aunque son ilegibles.
5. **Referencia y activación del servicio:** Los servicios se referencian generalmente mediante una URL, que se conoce como punto final. El servicio Web puede ejecutarse en la máquina de punto final, o en servidores secundarios.
6. **Transparencia:** Protege al programador de los detalles de la representación de los datos y asemeja una petición local a una remota.

Tipos de WebServices

SOAP [26] (Simple Object Access Protocol) es el protocolo base de los Web Services. Este protocolo está basado en XML y no se encuentra atado a ninguna plataforma o lenguaje de programación. Si bien es un protocolo, este no es un protocolo de comunicación entre mensajes como lo es HTTP. Básicamente SOAP son documentos XML que necesitan utilizar algún otro protocolo para ser transmitidos como

puede ser HTTP o cualquier otro tipo de protocolo. Consta de tres componentes principales:

1. **WSDL**: lenguaje de descripción del servicio.
2. **HTTP/SMTP**: protocolo de comunicación.
3. **XML**: lenguaje de peticiones y respuestas.

REST(Representational State Transfer) intentan emular al protocolo HTTP o protocolos similares mediante la restricción de establecer el interfaz en un conjunto conocido de operaciones estándar (por ejemplo GET, PUT,...). Este estilo se centra más en interactuar con recursos con estado que con mensajes y operaciones.

Cabe destacar que REST no es un estándar, ya que es tan solo un estilo de arquitectura. Aunque REST no es un estándar, está basado en estándares:

- HTTP
- URL
- Representación de los recursos: XML/HTML/GIF/JPEG/...
- Tipos MIME: text/xml, text/html, ...

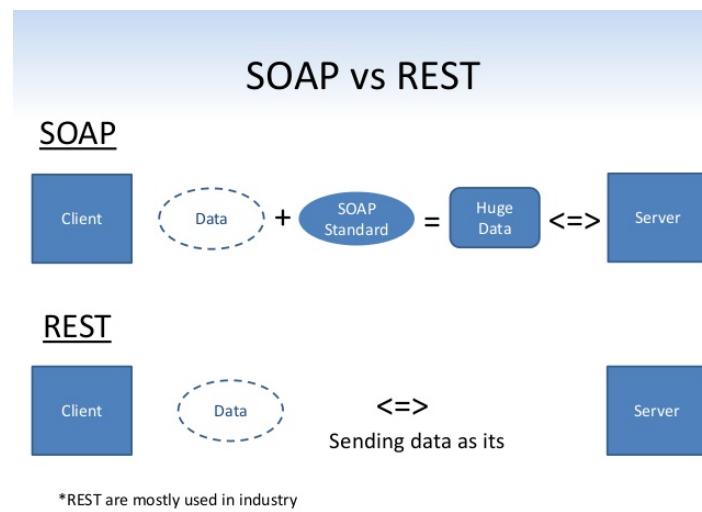


Figura 3.13: Comparativa Soap-Rest.

3.9.1. WebServices Google Maps

En el desarrollo de la práctica *Sitio Web de una tienda* se ha usado el WebServices de Google Maps⁴ que sigue la estructura REST. Consta de un conjunto de interfaces HTTP que proporcionan datos geográficos para aplicaciones que utilizan mapas. Antes de establecer la comunicación con el WebService es necesario obtener una clave⁵ asociado a la aplicación que implementa las peticiones. De los numerosos servicios de los que se dispone nos centramos en utilizar dos.

1. **Geocode**⁶: permite conocer las coordenadas geográficas de un lugar a partir del nombre. La petición se realiza a la URL definida en la que se incluye el formato de los datos(json o xml), el dato que se envía y la clave asociada a la aplicación.

```
https://maps.googleapis.com/maps/api/geocode/json?address=Madrid&key=YOUR_API_KEY
```

Fragmento de Código 3.15: Petición Geocode WS GoogleMaps.

2. **Place**⁷: permite obtener información asociada a un servicio (hoteles, restaurantes,...) tomando como punto de referencia una coordenada geográfica. La petición se realiza a la URL definida en la que se incluye el formato de los datos(json o xml),el dato que se envía y la clave asociada a la aplicación.

```
https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=-33.8670,151.1957&radius=500&types=food&key=YOUR_API_KEY
```

Fragmento de Código 3.16: Petición Place WS GoogleMaps.

⁴Descripción WebServices: <https://developers.google.com/maps/web-services/overview>

⁵<https://developers.google.com/maps/documentation/geocode/get-api-key>

⁶Descripción API Geocode :<https://developers.google.com/maps/documentation/geocoding/>

⁷Descripción API Place :<https://developers.google.com/places/web-service/search>

Capítulo 4

Comecocos Web

4.1. Enunciado

En la actualidad gracias a las novedades introducidas en la Web han aumentando los juegos desarrollados sin la necesidad de plugins o software de terceros para su ejecución.

Por ello en esta primera práctica se pide desarrollar un juego con herramientas nativas de la web. Dentro de las varias opciones existentes el juego seleccionado es Pac-Man (comecocos).



Figura 4.1: Aspecto clásico Pacman.

Requisitos Se presenta los distintos elementos que forman parte del juego y la función que deben desarrollar.

1. **Escenario:** Contiene todos los elementos estáticos del juego con los que interactúa Pacman como pueden ser los obstáculos y el contorno del escenario que no pueden ser sobrepasados y los cocos que es la comida de Pacman. Además informará al usuario del tiempo y puntos a medida que progrese la partida.

2. **Pacman:** se trata del personaje principal del juego que controle el usuario.
3. **Fantasmas:** son los enemigos que persiguen a Pacman durante la partida. En total serán 4 fantasmas¹ que ha diferencia de Pacman no interactuan con los elementos del juego ya que solo basan su comportamiento en la posición de Pacman, esto es conocido como modo de persecución por lo es necesario aplicar un algoritmo básico de Inteligencia Artificial.

Tecnologías Sera necesario emplear las etiquetas canvas y audio ademas de la API LocalStorage de HTML5, JavaScript e inteligencia artificial en juegos.

4.2. Diseño

Una vez se ha planteado la practica pasamos a definir el esquema que se va a seguir en la etapa del desarrollo.

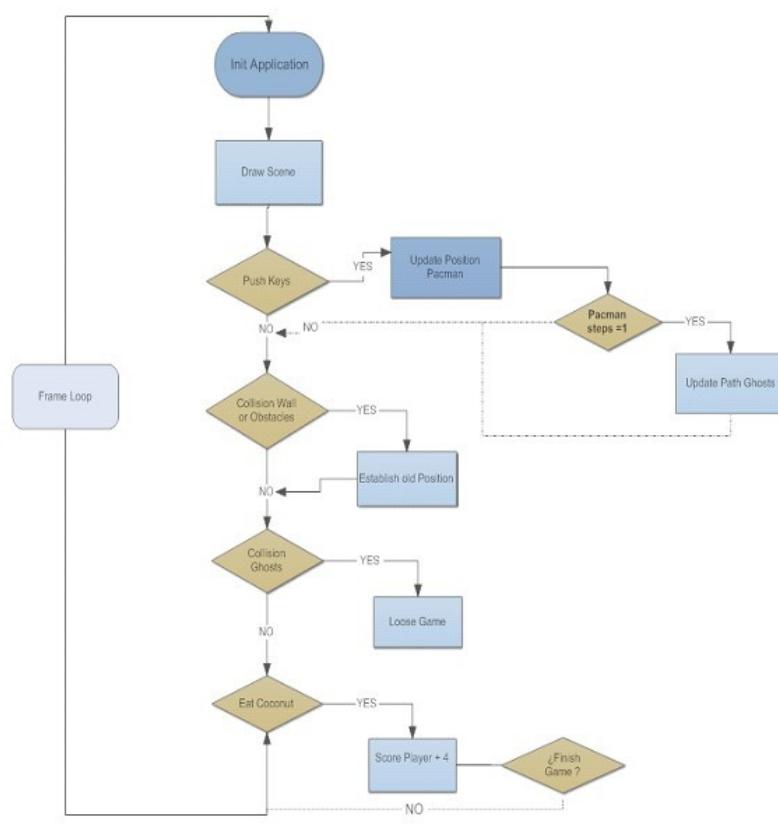


Figura 4.2: Diseño comeccocos web.

¹fuente:<http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior>

4.3. Desarrollo

Como se ha dicho a lo largo de este desarrollo la lógica del juego recae sobre JavaScript. Por esto se han creado tres tipos de objetos (GameArea, Pacman, Ghost) con el objetivo de hacer más compacto el código y menos repetitivo ya que hay muchas funciones que tienden a repetirse.

4.3.1. Game Area

Este objeto es el encargado de generar y dibujar los elementos del juego. El objeto inicializa una serie de variables que hacen referencia a dichos elementos.

```
value_cuad : 40,
filas:19,
columnas:18,
canvas : document.createElement("canvas"),
image :new Image(),
img_loose : new Image(),
img_win : new Image(),
/* time */
seconds : 0,
min : 0,
horas :0,
time : '00'+':'+ '00'+':'+ '00',
state : 0,
score :0,
start_crono : false,
lifePlayer : 1,
shape_1 : [...
],
shape_2 : [...
],
list_cocos : [...
],
list_obstaculos : [...
],
var map: [...
]
```

Fragmento de Código 4.1: Iniciacion de variables del Obj.Game

Tras la creación de las variables es necesario dotar al objeto de una serie de funciones que permitan dibujar los elementos.

Start

Inicializa el tamaño del lienzo y obtiene el contexto de *canvas* para poder dibujar los elementos dentro de el. Ademas obtenemos la referencia a las imágenes y

fuentes de audio que vamos a utilizar.

```
start : function() {
    this.img_loose.src = 'game_over.jpeg';
    this.img_win.src = 'game_win.jpg';
    this.image.src = 'pacman_fruit.png';
    this.canvas.width = this.value_cuad*this.filas;
    this.canvas.height = this.value_cuad*(this.columnas+4);
    this.context = this.canvas.getContext("2d");
    document.body.insertBefore(this.canvas,
    document.body.childNodes[2]);
    this.AudioGame = document.getElementById('musica');
    this.AudioDied = document.getElementById('hitPacman');
    this.AudioEat = document.getElementById('eating');
},

```

Fragmento de Código 4.2: Iniciacion de variables del Obj.Game

Escenario y elementos

La función **shape_scene** dibuja el escenario del juego por medio de un array con los comandos que se necesitan ejecutar.

```
shape_scene : function(shape) {
    this.context.save();
    this.context.beginPath();
    for (var i = 0; i < shape.length; i++) {
        var elemento = shape[i];
        for (var propiedad in elemento) {
            var x = elemento[propiedad];
            if(x.moveTo) {
                this.context.moveTo(this.value_cuad*x.moveTo[0],
                    this.value_cuad*x.moveTo[1]);
            }else if(x.lineTo) {
                this.context.lineTo(x.lineTo[0]*this.value_cuad,
                    this.value_cuad*x.lineTo[1]);
            }
        }
    }
    this.context.strokeStyle = 'blue';
    this.context.lineWidth = 2;
    this.context.stroke();
    this.context.restore();
},

```

Fragmento de Código 4.3: Visualizacion escenario.

Para dibujar los obstáculos definimos la función **draw_obstaclos** que lee el contenido de la variable **list_obstaculos**. Cada elemento esta formado por las coordenadas y dimensiones que se multiplican por el tamaño que tiene cada cuadro del escenario.

```
draw_obstaclos : function() {
    for(var i=0;i<this.list_obstaculos.length;i++) {
        var elemento = this.list_obstaculos[i];
        this.context.fillRect(elemento.x*value_cuad,elemento.y*
            value_cuad,
            elemento.width*value_cuad,elemento.height*value_cuad);
    }
},
```

Fragmento de Código 4.4: Visualizacion obstaculos.

Los últimos elementos son los cocos que por medio de la función **draw_dots** se dibujan. La función lee el contenido de la variable **list_cocos**, donde cada elemento tiene la misma estructura que los elementos de la variable **list_obstaculos**.

```
draw_dots : function() {
    if(this.list_cocos.length > 0) {
        for(var i=0;i<this.list_cocos.length;i++) {
            var elemento = this.list_cocos[i];
            this.context.fillStyle = 'white';
            this.context.beginPath();
            this.context.arc((elemento.x*40)+20,(elemento.y*40)+20,
                elemento.radio,0,(Math.PI/180),true);
            this.context.fill();
        }
    }
},
```

Fragmento de Código 4.5: Visualizacion cocos.

En la imagen 4.3 se muestra el aspecto que tiene el escenario al utilizar las funciones anteriores.

Información de la partida

Otra característica que presenta el objeto es mostrar información sobre el estado de la partida.

La función **time_game** se encarga de dibujar el cronometro del juego. Para ello dibuja el valor que contiene la variable **time**. El valor de esta variable se actualiza por medio de la función **CronoTime** que se explica mas adelante.

```
time_game:function() {
    this.context.save();
    this.context.font = "30px BDCartoonShoutRegular";
    roundedRect(this.context,8*40,20*40,5*40,1*40,10);
```

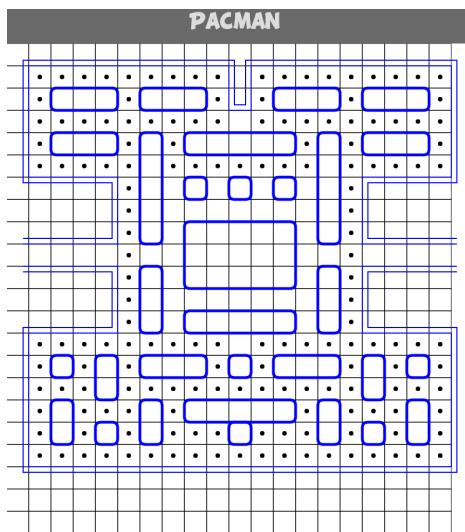


Figura 4.3: Apariencia elementos del juego.

```
this.context.fillStyle = "#00FA9A";
this.context.fillText(this.time, 8.75*40, 20.80*40);
this.context.restore();
},
```

Fragmento de Código 4.6: Visualizacion cronometro.

También dibuja la información del marcador del usuario por medio de la función **score_user** que utiliza la información de variable **score**.

```
score_user : function() {
this.context.save();
this.context.fillStyle = "white";
this.context.font = "30px BDCartoonShoutRegular";
this.context.fillText('score: '+this.score, 1*40, 20*40);
this.context.restore();
},
```

Fragmento de Código 4.7: Visualizacion Marcador.

Por ultimo, la función **life_user** se encarga de dibujar el contenido de la variable **life**.

```
life_user : function() {
this.context.save();
this.context.fillStyle = "white";
this.context.font = "30px BDCartoonShoutRegular";
this.context.fillText('life: '+this.life, 1*40, 21*40);
this.context.restore();
},
```

Fragmento de Código 4.8: Visualizacion vidas.

En la imagen 4.4 se muestra el aspecto que tiene el escenario al utilizar las funciones anteriores.

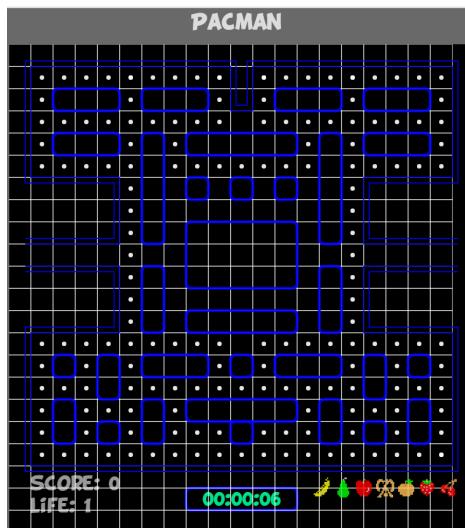


Figura 4.4: Apariencia información del juego.

Fin de Partida

La partida puede terminar cuando Pacman ha logrado comerse todos los cocos de la partida lo que produce la ejecución de la función **win_game** que carga una imagen indicando al usuario que ha ganado y permite guardar la información de la partida a través de una función **PosMouseClick** que se explica mas adelante.

```
win_game : function() {
    var finish = false;
    if(this.list_cocos == 0) {
        this.context.globalAlpha = 0.9;
        this.context.save();
        this.context.drawImage(this.img_win,0,0,
            this.canvas.width,this.canvas.height);
        roundedRect(this.context,this.canvas.width/2-(4*40),13*40,
            8*40,1.5*40,10);
        this.context.fillStyle = "#00FA9A";
        this.context.font = "30px BDCartoonShoutRegular";
        this.context.fillText('Save Score',
            this.canvas.width/2-(2.5*40),14*40);
        this.context.restore();
        finish = true;
    }
}
```

```

    }
    return finish
},

```

Fragmento de Código 4.9: Visualizacion Partida Ganada.

Otro forma de terminar la partida se produce cuando los fantasmas capturan a Pacman lo que produce la ejecución de la función **lose_game** que carga una imagen indicando al usuario que ha perdido.

```

lose_game : function() {
    this.context.save();
    this.context.globalAlpha = 0.6;
    this.context.drawImage(this.img_loose,0,0,
        this.canvas.width,this.canvas.height);
    this.context.restore();
},

```

Fragmento de Código 4.10: Visualizacion Partida Perdida.

4.3.2. Pacman

Es el protagonista del juego y por ende el usuario interactúa con el moviéndolo por todo el escenario. En su lógica tenemos que tener en cuenta diversos factores que afectan a su progreso por el juego.

Detectar colisiones

Tenemos que tener en cuenta el entorno del juego con respecto a la posición en la que se encuentra Pacman. Para ello la función **hitt_Counter** evalúa que las coordenadas(x,y) no sobrepasen el contorno del escenario.

```

this.hitt_counter = function() {
    var hitt_counter = false;
    if(this.x < 0 || this.x > GameArea.filas) {
        hitt_counter = true;
    }else if(this.y < 0 || this.y > GameArea.columnas) {
        hitt_counter = true;
    }
    return hitt_counter;
}

```

Fragmento de Código 4.11: Detección de colisiones con el escenario.

Otro tipo de colisión a tener en cuenta es con los obstáculos, de esto se encarga la función **hitObject**. La función obtiene los vértices del objeto y de Pacman para comprobar si alguno de los vértices de Pacman se encuentran dentro del área que forman los vértices del objeto.

```

this.hitObject = function(list){
    var hitt = false;
    for(var i = 0;i<list.length;i++) {
        var obstaculo = list[i];
        var Aobstaculo = {x:obstaculo.x,y:obstaculo.y};
        var Bobstaculo = {x:obstaculo.x+obstaculo.width,
                          y:obstaculo.y};
        var Cobstaculo = {x:obstaculo.x,
                          y:obstaculo.y+obstaculo.height};
        var Dobstaculo = {x:obstaculo.x+obstaculo.width,
                          y:obstaculo.y+obstaculo.height};
        var Apac = {x:this.x,y:this.y};
        var Bpac = {x:this.x+this.width,y:this.y};
        var Cpac = {x:this.x,y:this.y+this.height};
        var Dpac = {x:this.x+this.width,y:this.y+this.height};
        if(Apac.x > Aobstaculo.x && Apac.x < Bobstaculo.x &&
            Apac.y > Aobstaculo.y && Apac.y < Cobstaculo.y){
            hitt = true;
            break
        }
        if(Bpac.x > Aobstaculo.x && Bpac.x < Bobstaculo.x &&
            Bpac.y > Aobstaculo.y && Bpac.y < Cobstaculo.y){
            hitt = true;
            break
        }
        if(Cpac.x > Aobstaculo.x && Cpac.x < Bobstaculo.x &&
            Cpac.y > Aobstaculo.y && Cpac.y < Cobstaculo.y){
            hitt = true;
            break
        }
        if(Dpac.x > Aobstaculo.x && Dpac.x < Bobstaculo.x &&
            Dpac.y > Aobstaculo.y && Dpac.y < Cobstaculo.y){
            hitt= true;
            break
        }
    }
    if(hitt == true){
        this.move = false;
    }
}

```

Fragmento de Código 4.12: Detección de colisiones con objetos del juego.

En la imagen 4.5 se muestra la colisión con un objeto impidiendo a Pacman avanzar.

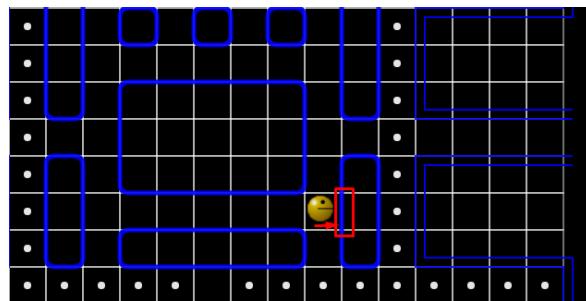


Figura 4.5: Colisión Pacman-Obstáculo.

Comer Cocos

Los cocos se encuentran por todo el escenario por lo que es necesario comprobar si hay colisión con ellos.

La función **eat_doit** calcula los vértices de pacman para evaluar si la posición del coco esta dentro del área formada por los vértices anteriores. En caso afirmativo se elimina este elemento de la variable **GameArea.score** para no volverlo a ser dibujado.

```
this.eat_doit = function(list_cocos) {
    var eat = false;
    for(var i=0;i<list_cocos.length;i++) {
        var coco = list_cocos[i];
        var Apac = {x:this.x,y:this.y};
        var Bpac = {x:this.x+this.width,y:this.y};
        var Cpac = {x:this.x,y:this.y+this.height};
        var Dpac = {x:this.x+this.width,y:this.y+this.height};
        if (coco.x+0.5 > Apac.x && coco.x+0.5 < Bpac.x &&
            coco.y+0.5 > Apac.y && coco.y+0.5 < Cpac.y ) {
            list_cocos.splice(i,1);
            eat = true
            GameArea.score = GameArea.score +4 ;
            break;
        }
    }
    return eat;
}
```

Fragmento de Código 4.13: Detección de colisión con los cocos.

En la imagen 4.6 se muestra la colisión con un coco lo que provoca que dicho coco desaparezca.

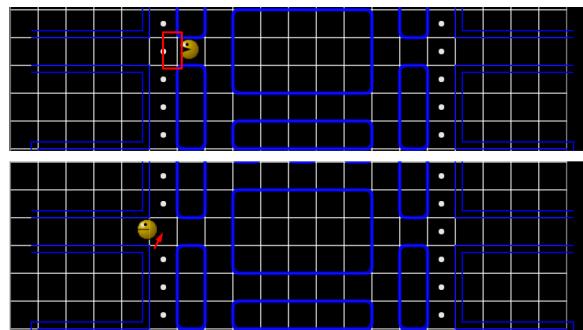


Figura 4.6: Colisión Pacman-Cocos.

Actualizar posición Canvas

Para actualizar la posición de Pacman la función **new_position** suma los nuevos valores que han sido asignados al haber pulsado una tecla en las variables **x_speed** e **y_speed**.

```
this.new_position = function () {
    this.x += this.speed_x;
    this.y += this.speed_y;
}
```

Fragmento de Código 4.14: Actualizar posición en canvas.

Tras esto, la función **add_steps** actualiza el valor de la variable **pasos** que es un contador definido para evaluar si su valor es +/-1 lo que indica que se ha desplazado a una nueva casilla del mapa.

```
this.add_steps = function() {
    if(this.speed_x != 0 || this.speed_y != 0) {
        if(this.type_move == 'y_pos' || this.type_move == "y_neg") {
            this.pasos += this.speed_y;
        }else{
            this.pasos += this.speed_x;
        }
    }
}
```

Fragmento de Código 4.15: Actualización de pasos dados.

Actualizar posición Mapa

Actualizamos el valor de las variables **x_map** e **y_map** según corresponda y restablecemos el valor de la variable **pasos**.

```
this.new_path = function () {
```

```

if(this.x_map < 22 && this.x_map >= 0) {
    if(this.y_map < 19 && this.y_map >= 0) {
        if(this.type_move == 'y_pos') {
            this.y_map += 1;
        }else if(this.type_move == 'x_pos') {
            this.x_map += 1;
        }else if(this.type_move == 'y_neg') {
            this.y_map -= 1;
        }else{
            this.x_map -= 1;
        }
        this.pasos = 0;
        this.new_pos = true;
    }
}
}

```

Fragmento de Código 4.16: Actualización coordenada del mapa.

Dibujar

Finalmente, para dibujar a Pacman utilizamos la función **drawImagen()** de canvas que carga el trozo de imagen correspondiente a Pacman del spreadsheet. Posee dos estados de dibujo para crear la sensación de animación.

```

this.draw = function() {
    GameArea.context.save();
    if(this.state_draw == 0) {
        GameArea.context.drawImage(this.image,320,this.yDraw,32,32
            ,this.x*40,this.y*40,35,35);
        this.state_draw = 1;
    }else{
        GameArea.context.drawImage(this.image,320+32,this.yDraw,
            32,32,this.x*40,this.y*40,35,35);
        this.state_draw = 0;
    }
    GameArea.context.restore();
}

```

Fragmento de Código 4.17: Dibujar Pacman.

4.3.3. Fantasmas

Definimos el objeto **Ghost** que tiene la lógica del comportamiento de los cuatro fantasmas que forman parte del juego.

En el momento de instanciar el objeto **Ghost(x_map,y_map,name,speed,initMoment)**

le pasamos las coordenadas (x_map,y_map) donde se dibujara,el nombre,su velocidad y el numero de cocos que Pacman tiene que haber comido para salir a perseguirlo.

Persecución Pacman

La función **init** se encarga de evaluar si se tiene que activar el fantasma, en caso afirmativo lo coloca fuera de la casa y empieza la persecución.

```
this.init = function(score) {
    if(score == this.initMoment && !this.move) {
        this.x_map =11;
        this.y_map =7;
        this.move = true;
    }
}
```

Fragmento de Código 4.18: Actualizar posicion del los Fantasmas.

Actualizar objetivo

La función **new_path** aplica IA a cada fantasmas, por medio del mapa del juego obtiene el camino que cada fantasma tiene que seguir. Para ello tomamos como punto de inicio la posición actual del fantasma y la posición de Pacman formado por las variables **x_map** e **y_map** como punto de destino .A continuación explicamos el modo de persecución de cada fantasma.

- *Blinky*: su comportamiento es seguir a Pacman en la misma dirección, entonces el punto de inicio es la posición actual de fantasma y el punto final es la posición actual de Pacman.
- *Speedy*: al igual que Blinky persigue a Pacman, pero tiene en cuenta su dirección. En este caso el cálculo se realiza tomando como punto de partida la posición de Speedy y se consulta la dirección de Pacman , al resultado le sumamos/restamos 4 posiciones según corresponda para obtener el punto final.
- *Clyde*: tiene dos tipos de comportamientos que dependen de la distancia entre él y Pacman, si dicha distancia es mayor a ocho sigue en modo persecución en caso contrario deja de seguirlo y se aleja de él tomando como nuevo objetivo una de las esquinas.

El resultado de cada operación se guarda en la variable **result** correspondiente.

```
this.new_path = function(graph,pacman_x,pacman_y,direccion) {
    var start = graph.grid[this.x_map][this.y_map];
    if(this.name == 'blinky') {
        var end = graph.grid[pacman_x][pacman_y];
    }else if(this.name == 'speedy') {
```

```

if(pacman_x+4 < 21 && direccion == 'x_pos') {
    pacman_x += 4;
} else if(pacman_x-4 > 0 && direccion == 'x_neg') {
    pacman_x -= 4;
} else if(pacman_y+4 < 19 && direccion == 'y_pos') {
    pacman_y += 4;
} else if(pacman_y-4 > 0 && direccion == 'y_neg') {
    pacman_y -= 4;
}
var end = graph.grid[pacman_x][pacman_y];
} else if (this.name == 'clyde') {
    var end = graph.grid[pacman_x][pacman_y];
    this.result = astar.search(graph,start,end,false);
    if(this.result.length < 8){
        var end = graph.grid[this.cuad_static.x][this.cuad_static.y];
    }
}
this.result = astar.search(graph,start,end,false);
}

```

Fragmento de Código 4.19: Actualizar direccion hacia el objetivo

Actualizar posición

Para actualizar la posición de los fantasma vinculamos la función **nexStepGhost** por medio de un **timer** de forma que dicha función se ejecute según el valor definido en la variable **speed** de cada fantasmas.

```

function nexStepGhost(ghost) {
    ghost.flag = 1;
    if(ghost.result.length > 0){
        ghost.x = ghost.result[0].x;
        ghost.y = ghost.result[0].y;
        ghost.x_map = ghost.result[0].x;
        ghost.y_map = ghost.result[0].y;
        ghost.result.splice(0,1);
    }
    ghost.interval = setTimeout(function() {
        nexStepGhost(ghost)
    },ghost.speed);
}

```

Fragmento de Código 4.20: Actualizar posicion del los Fantasmas.

Dibujar

Para dibujar los fantasmas lo hacemos igual que Pacman, pero al tener diferentes fantasmas por medio del nombre asignado en su creación obtenemos la imagen

correspondiente a cada uno.

```

this.draw = function() {
    GameArea.context.save();
    if(this.name == 'blinky') {
        if(this.state_draw == 0) {
            GameArea.context.drawImage(this.image, 0, 0, 32, 32, this.x_map
                *40, this.y_map*40, 35, 35);
        } else {
            GameArea.context.drawImage(this.image, 32, 0, 32, 32, this.x_map
                *40, this.y_map*40, 35, 35);
        }
    } else if(this.name == 'clyde') {
        if(this.state_draw == 0) {
            GameArea.context.drawImage(this.image, 64, 0, 32, 32, this.x_map
                *40, this.y_map*40, 35, 35);
        } else {
            GameArea.context.drawImage(this.image, 94, 0, 32, 32, this.x_map
                *40, this.y_map*40, 35, 35);
        }
    } else if(this.name == 'inky') {
        if(this.state_draw == 0) {
            GameArea.context.drawImage(this.image, 192, 0, 32, 32, this.x_map
                *40, this.y_map*40, 35, 35);
        } else {
            GameArea.context.drawImage(this.image, 222, 0, 32, 32, this.x_map
                *40, this.y_map*40, 35, 35);
        }
    } else if(this.name == 'speedy') {
        if(this.state_draw == 0) {
            GameArea.context.drawImage(this.image, 128, 0, 32, 32, this.x_map
                *40, this.y_map*40, 35, 35);
        } else {
            GameArea.context.drawImage(this.image, 160, 0, 32, 32, this.x_map
                *40, this.y_map*40, 35, 35);
        }
    }
    this.state_draw = ( this.state_draw === 1 ) ? 0 : 1;
    GameArea.context.restore();
}

```

Fragmento de Código 4.21: Dibujar Fantasma.

En la imagen 4.7 se muestra un ejemplo del comportamiento de los fantasmas..

4.3.4. Movimiento

Para dotar de movimiento al juego a la función **updateGameArea** se vincula a un timer que se ejecuta cada 100ms y de esta forma mostrar los cambios de cada

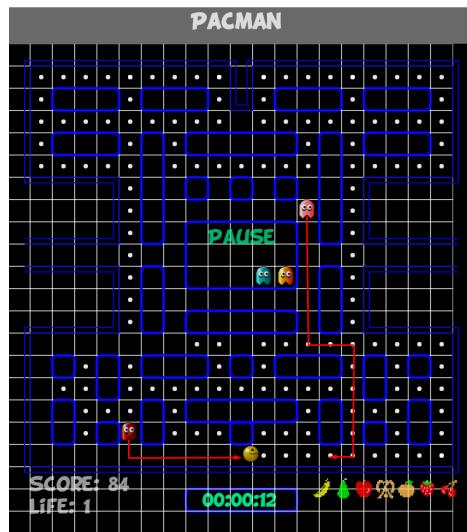


Figura 4.7: Apariencia comportamiento Fantasmas.

objeto.

```

function updateGameArea() {
    /** GameArea **/
    GameArea.clear();
    GameArea.AudioGame.play();
    GameArea.time_game();
    GameArea.score_user();
    GameArea.life_user();
    GameArea.shape_scene();
    GameArea.draw_dots(list_cocos);
    GameArea.draw_obstacles(list_obstaculos);
    GameArea.draw_fruit();
    /** Fantasma **/
    for(var i = 0; i < list_Ghost.length;i++){
        var _ghost = list_Ghost[i];
        _ghost.init(GameArea.score);
        if (_ghost.move ){
            _ghost.new_path(graph,Pac.x_map,Pac.y_map,Pac.type_move);
        }
    }
    _ghost.draw();
    /** Pacman **/
    Pac.new_position();
    Pac.hitObject(list_Ghost);
    if (!Pac.move){
        /*Refibujamos todo el conteido del canvas diciendo que a
         *perdido */
        GameArea.AudioGame.pause();
    }
}

```

```

GameArea.context.globalAlpha = 0.9;
GameArea.context.save();
GameArea.lose_game();
GameArea.context.restore();
GameArea.AudioDied.play();
GameArea.stop();
}else{
    var x = Pac.hitt_counter();
    Pac.hitObject(list_obstaculos);
    if(Pac.move != true || x == true) {
        /* si hay choque hay que resetear el contenido para dibujar */
        Pac.reset();
    }else{
        Pac.add_steps();
        if(Pac.pasos == 1 || Pac.pasos == -1 ){
            Pac.new_path();
        }
    }
}
/* se comprueba si hemos comido algo para poner el sonido*/
var eatPil = Pac.eat_doit(list_cocos);
if(eatPil){
    GameArea.AudioGame.volume=0;
    GameArea.AudioEat.play();
    GameArea.AudioGame.volume=0;
}
Pac.reset_speed();
Pac.draw();
GameArea.win_game(list_cocos);
if(list_cocos.length == 0){
    GameArea.stop();
}
}
}

```

Fragmento de Código 4.22: Renderizado del juego.

4.4. Pruebas

Para realizar las pruebas necesitamos disponer de un portátil en el que se encuentre el fichero de la practica, ademas de tener instalado varios navegadores como pueden ser Chrome, FireFox e Internet Explorer.

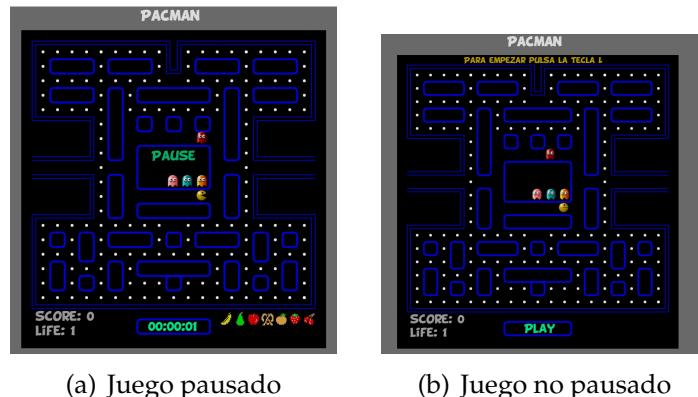
Se ejecuta el fichero **game_pacman.html** en cada navegador provocando que se visualice el juego, figura 4.8.

Para iniciar la aplicación el usuario pulsa la **tecla L** provocando el movimiento de los fantasmas y el inicio del cronometro. A partir de este momento el usuario puede manipular a Pacman por medio de las flechas del teclado. Ademas se permite



Figura 4.8: Inicio juego Pacman.

al usuario pausar el juego pulsado la **tecla P**, figura 4.9.



(a) Juego pausado

(b) Juego no pausado

Figura 4.9: Pause/Start juego.

La partida termina cuando uno de los fantasmas capture a Pacman provocando que se visualice **Game Over** o cuando Pacman se ha comido todos los cocos provocando que se visualice **Win Game**, figura 4.10

A continuación, se muestra una tabla resumen del resultado de las pruebas en los distintos navegadores.

Cuadro 4.1: My caption

	Chrome	Firefox	Internet Explore
Funcionamiento APP	OK	OK	OK



(a) Partida perdida

(b) Partida Ganada

Figura 4.10: Game/Loose juego.

Capítulo 5

Comecocos Multijugador

5.1. Enunciado

En la comunidad de internet han aumentado el desarrollo de aplicaciones en tiempo real entre distintos usuarios que se encuentran en diferentes partes del mundo. Esta segunda practica es una extensión de la primera ya que ahora el juego se va a ser multijugador online por medio de interfaz Web.



Figura 5.1: Ejemplo Aspecto multijugador Pacman.

Requisitos El comportamiento del juego sera el desarrollado en la primera practica pero sera necesario organizarlo de tal forma que el cliente se encargue de ciertas tareas y el servidor de otras con el objetivo de intercambiar informacion por medio de WebSockets, a continuación se explica en detalle cada una.

1. Cliente

- Conexión WebSockets con el servidor.

- Dibujar el escenario del juego,fantasmas,jugadores e información de la partida.
 - Gestión del movimiento del personaje por medio de los eventos del teclado.

2. Servidor

- Entregar el fichero que contiene la aplicación.
 - Conexión WebSockets con los clientes.
 - Creación de un modulo que contenga los valores iniciales del juego como los cocos,obstáculo entre otros y un objeto de tipo **Player** que contenga la posición del usuario y del fantasma correspondiente.
 - Se encargara de la lógica de los fantasmas aplicada en la primera práctica. Hay que remarcar que en esta versión solo existe un fantasma por usuario.
 - Por medio de la conexión WebSockets gestiona la entrada en sala,inicio del juego y la actualización de los distintos elementos del juego.

Tecnologías Sera necesario emplear la API WebSockets y JavaScript en el cliente mientras que en el servidor utilizaremos NodeJS.

5.2. Diseño

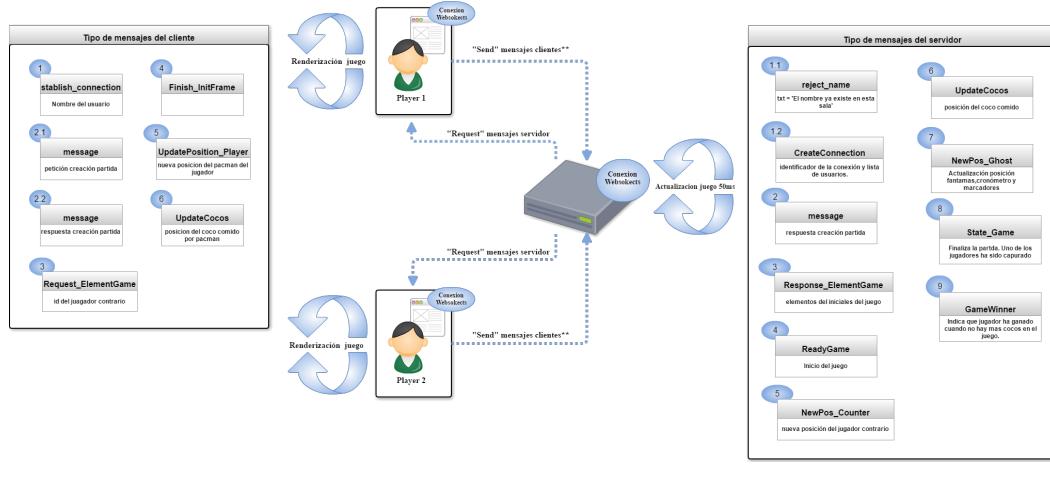


Figura 5.2: Esquema Pacman multijugador.

5.3. Desarrollo Servidor

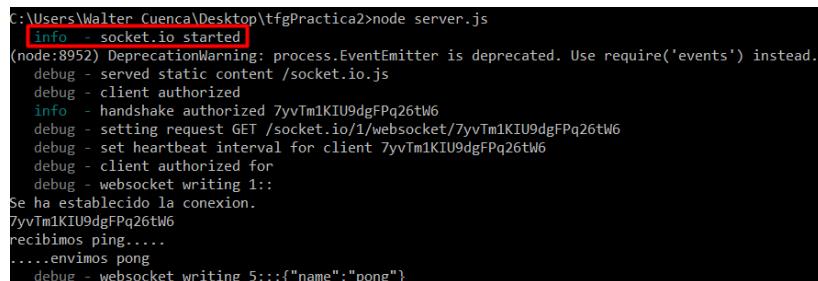
El servidor de la aplicación lo desarrollamos con NodeJS por lo que se importa las librerías **node-static** y **http** para la creación del servidor.

Con esto el servidor entrega el fichero inicial a los usuarios que se conecten pero queda importar la librería **socket.io** para crear la conexión WebSockets en el momento de recibir una petición.

```
var static = require('node-static');
var http = require('http');
var file = new(static.Server)();
var app = http.createServer(function (req, res) {
  file.serve(req, res);
  console.log('Server listening 8181');
}).listen(8181);
/* inst. socket.io */
var io = require('socket.io').listen(app);
```

Fragmento de Código 5.1: Definición del servidor.

La imagen 5.3 muestra la conexión de un cliente al servidor y el inicio de la conexión WebSockets.



```
C:\Users\Walter Cuenca\Desktop\tfgPractica2>node server.js
info - socket.io started
(node:8952) DeprecationWarning: process.EventEmitter is deprecated. Use require('events') instead.
debug - served static content /socket.io.js
debug - client authorized
info - handshake authorized 7yvTm1KIU9dgFPq26tW6
debug - setting request GET /socket.io/1/websocket/7yvTm1KIU9dgFPq26tW6
debug - set heartbeat interval for client 7yvTm1KIU9dgFPq26tW6
debug - client authorized for
debug - websocket writing 1:
Se ha establecido la conexión.
7yvTm1KIU9dgFPq26tW6
recibimos ping.....
.....enviamos pong
debug - websocket writing 5:::{ "name": "pong" }
```

Figura 5.3: Ejecución del servidor.

Modulo Game

Se crea el modulo **CoreServer.js** en NodeJs para separar la lógica y las variables que definen los elementos del juego. Este modulo primero define las variables que tienen relación con el aspecto del juego como son los obstáculos, cocos y propiedades del juego.

```
var map = [...
];
/* Jugadores */
var player1 = {'x':5,'y':15,'typePacman':1,numPasos:0};
var player2 = {'x':11,'y':3,'typePacman':2,numPasos:0};
```

```

/* */
var list_user=[];
var name_room = 'hallGame';

/* Cronometro */
var seconds = 0;
var min = 0;
var horas = 0;
var time = '00' + ':' + '00' + ':' + '00';
var _timer;
var list_cocos = [
];
var list_obstaculos = [
];
var properGame={
    'wCuad':40,
    'hCuad':40,
    'nColum':18,
    'nFila':19,
}
var shape_1 = [....];
var shape_2 = [....];

```

Fragmento de Código 5.2: Definición variables del Obj.Game

Por otro lado define el objeto **Player** para representar a cada jugador. El objeto contiene la posición y pasos del personaje e incluye la posición y lista de nodos del fantasma correspondiente.

```

function Player(x,y,typePacman,xGhost,yGhost) {
    this.info= {'x':x,
    'y':y,
    'typePacman':typePacman,
    'numPasos':0,
    'fantasma': {'x':xGhost,
    'y':yGhost,
    },
    };
    this.path = [];
}

```

Fragmento de Código 5.3: Definición Player Obj.Game

Para que el contenido del modulo sea accesible desde el servidor es necesario declarar los elementos como **module.exports**.

```

module.exports = {
    'map':map,
}

```

```

'shape_1':shape_1,
'shape_2':shape_2,
'properGame':properGame,
'list_obstaculos':list_obstaculos,
'list_cocos':list_cocos,
'name_room':name_room,
'_timer;':_timer,
'seconds':seconds,
'min':min,
'horas':horas,
'time':time,
'Player':Player
}

```

Fragmento de Código 5.4: Export elementos del Obj.Game

Tras esto volvemos al fichero **server.js** donde se declara el acceso al modulo a través de la variable **Game**. A partir de ella creamos la instancia de los jugadores y calculamos la lista de nodos de su correspondiente fantasma por medio de la función auxiliar **CreatePath()**.

```

var Game = requiere('./ghost.js')

/* posiciones iniciales */
var player1 = new Game.Player(5,15,1,1,1);
var player2 = new Game.Player(14,15,2,18,1);
player1.path = CreatePath(player1.info,player1.info.fantasma);
player2.path = CreatePath(player2.info,player2.info.fantasma);

```

Fragmento de Código 5.5: Instancia Jugadores del Servidor.

Lógica del Servidor

Hasta este momento solo se ha definido los elementos del juego pero no se ha visto como el servidor gestiona cada uno de los estados del juego hasta que este finaliza.

Aquí es donde **WebSockets** interviene por medio de la librería **socket.io**, ya que permite al servidor gestionar los mensajes que recibirá. A continuación, se explican los tipos de mensajes y su tratamiento.

Sala de juego

Definimos el evento **stablish_connection** que recibe el nombre del usuario para validar la entrada a la sala.

Primero comprueba el numero de usuario en la sala ya que esta restringido a dos. Si aun no esta completa comprueba que el nombre del usuario no exista por medio de la función **seekUser(name)**. Si existe el nombre envía un mensaje **reject_name**

en caso contrario envía un mensaje **CreateConnection** con el id de la conexión y la lista de usuario dentro de la sala.

Finaliza vinculando la conexión del usuario a la sala del juego y actualizando la lista de usuarios.

```
socket.on('stablish_connection',function(name) {
  var numClients = io.sockets.clients(Game.name_room).length;
  var existeName = seekUser(name);
  if(existeName) {
    var txt = 'El nombre ya existe en esta sala';
    socket.emit('reject_name',txt);
  }else{
    socket.emit('CreateConnection',socket.id,list_user);
    socket.username = name;
    socket.room = Game.name_room;
    socket.join(Game.name_room);
    if(list_user.length == 0){
      var user = {'user':name,'room':Game.name_room,'id':socket.id,
                  'posGame':player1.info,'pos':1,'score':0};
    }else{
      var user = {'user':name,'room':Game.name_room,'id':socket.id,
                  'posGame':player2.info,'pos':2,'score':0};
    }
    list_user.push(user);
    socket.broadcast.to(Game.name_room).emit('New_Joined',socket.id
      );
  }
});
```

Fragmento de Código 5.6: Definición evento stablish_connection

La figura 5.4 muestra la conexión del primer jugador mientras la figura 5.5 trata la del segundo jugador.

```
Numero de clientes:0
false
walter
[]
debug - websocket writing 5:::{ "name": "CreateConnection", "args": [ "7yvTm1KIU9dgFPq26tW6", [] ] }
debug - broadcasting packet
```

Figura 5.4: Petición sala 1^a jugador.

Petición de partida

La petición y respuesta a la creación de una partida la realizan los usuarios por lo que el servidor en este caso sera transparente para ellos. Por ello definimos el

```

Número de clientes:1
false
PEDRO
[ { user: 'walter',
  room: 'hallGame',
  id: 'zYjISGMJ4xgkcoVC8d3d',
  posName: { x: 5, y: 15, typePacman: 1, numPasos: 0, fantasma: [Object] },
  pos: { x: 5, y: 15, score: 0 } ]
debug - websocket writing 5:::{name:"CreateConnection","args":["AQYWhlyjpA2T5bb9u8d3e",{"user":"walter","room":"hallGame","id":"zYjISGMJ4xgkcoVC8d3d","posGame":{"x":5,"y":15,"typePacman":1,"numPasos":0,"fantasma":[{"x":1,"y":1}],"score":0}]}
debug - broadcasting packet
debug - websocket writing 5:::{name:"New Joined","args":["AQYWhlyjpA2T5bb9u8d3e"]}

```

Figura 5.5: Petición sala 2º jugador.

evento **message** que encamina los mensajes entre los usuarios por medio del valor **idDestino**.

```

socket.on('message', function(message) {
  io.sockets.socket(message.idDestino).emit('message', message);
});

```

Fragmento de Código 5.7: Definición evento message.

En la figura 5.7 se muestra como el servidor encamina los mensajes entre los usuarios.

```

Reenvío de mensaje al nodo destino.
{ typeMsg: 'Init',
  idOrigen: 'AQYWhlyjpA2T5bb9u8d3e',
  texto: 'PEDRO: Jugamos una partida ??',
  idDestino: 'zYjISGMJ4xgkcoVC8d3d'
  debug - websocket writing 5:::{name:"message","args":[{"typeMsg":"Init","idOrigen":"AQYWhlyjpA2T5bb9u8d3e","texto":"PEDRO: Jugamos una partida ??","idDestino":"zYjISGMJ4xgkcoVC8d3d"]}}

```

Figura 5.6: Petición de partida usuario.

```

Reenvío de mensaje al nodo destino.
{ typeMsg: 'replayInit',
  idOrigen: 'lZltW7ebdeUpMlHYg93lb',
  texto: 'si',
  idDestino: 'QYYcSeQIKup-2apE93lc'
  debug - websocket writing 5:::{name:"message","args":[{"typeMsg":"replayInit","idOrigen":"lZltW7ebdeUpMlHYg93lb","texto":"si","idDestino":"QYYcSeQIKup-2apE93lc"]}}

```

Figura 5.7: Respuesta de partida usuario.

Elementos del juego

Tras haber acordado jugar una partida ambos jugadores se define el evento **Request_ElementGame** para enviar los parámetros iniciales del juego.

Obtiene la información de los jugadores utilizando la función **getInfoUser(id)** pasándole cada uno de los identificadores.

Con la información anterior y las características del escenario crea la variable **elements** que se envía dentro del mensaje **Response_ElementGame**.

```

socket.on('Request_ElementGame', function(otherId) {
  var myInfo = getInfoUser(socket.id);

```

```
var counterInfo = getInfoUser(otherId);
var element = {
  'shape_1':Game.shape_1,
  'shape_2':Game.shape_2,
  'cocos':Game.list_cocos,
  'obstaculos':Game.list_obstaculos,
  'properGame':Game.properGame,
  'myInfo':myInfo,
  'counterInfo':counterInfo
}
socket.emit('Response_ElementGame',element);
});
```

Fragmento de Código 5.8: Definicion del evento Request_ElementGame

La figura 5.8 muestra el envío de la información a los jugadores.

Figura 5.8: Envío elementos del juego.

Inicio del juego

Para iniciar la partida definimos el evento **Finish_InitFrame**. Se encarga de validar que la petición de inicio de partida este solicitada por los dos jugadores para enviar el mensaje **ReadyGame** a los clientes.

En este punto se establece el bucle de actualización del juego por parte del servidor a los cliente por medio del evento timer **setInterval(UpdateGhost,800)**.

```
socket.on('Finish_InitFrame',function(){
    cooardiar +=1;
    if(cooardiar == 2){
        io.sockets.in(Game.name_room).emit('ReadyGame');
        setTimeout(function(){
            Game._timer = setInterval(UpdateGhost,800);
            CronoTime();
            },3000)
    }
});
```

Fragmento de Código 5.9: Definición del evento Finish_InitFrame

La función **UpdateGhost** es la parte mas importante de la lógica del servidor ya que se encarga de actualizar la posición de cada fantasma y obtener la puntuación de cada usuario para enviar un mensaje **NewPos_Ghost**.

También comprueba si alguno de los usuarios a colisionado con el fantasma lo que provoca que se envié el mensaje **State_Game** a los usuarios.

```

function UpdateGhost () {
    var elementsGame = [];
    /* */
    var hitGhost1 = playerHitGhost (player1); //true o false
    var hitGhost2 = playerHitGhost (player2); // true o false
    if (!hitGhost1 && !hitGhost2) {
        var coordGhost={'x':player1.path[0].x,'y':player1.path[0].y};
        setGhost_User(list_user[0].id,coordGhost);
        coordGhost={'x':player2.path[0].x,'y':player2.path[0].y};
        setGhost_User(list_user[1].id,coordGhost);
        elementsGame.push({'id':list_user[0].id,'x':player1.path
            [0].x,'y':player1.path[0].y,'score':list_user[0].score
            });
        player1.path.splice(0,1);
        elementsGame.push({'id':list_user[1].id,'x':player2.path
            [0].x,'y':player2.path[0].y,'score':list_user[1].score
            });
        player2.path.splice(0,1);
        io.sockets.in(Game.name_room).emit ('NewPos_Ghost',
            fantasmas,Game.time,scores);
    }

    if (hitGhost1 || hitGhost2){
        var msgPlayer1 = (hitGhost1 === true) ? 'KO' : 'OK';
        var msgPlayer2 = (hitGhost2 === true) ? 'KO' : 'OK';
        io.sockets.socket(list_user[0].id).emit ('State_Game',
            msgPlayer1);
        io.sockets.socket(list_user[1].id).emit ('State_Game',
            msgPlayer2);
        clearInterval(Game._timer);
    }
}

```

Fragmento de Código 5.10: Definición de la función UpdateGhost.

La figura 5.9 muestra el envío de la información actualizada del juego a los jugadores.

```
debug - websocket writing 5:::{name:"NewPos_Ghost","args": [{"id":"zYj1S60J4xgkcoVC8d3d","x":1,"y":2}, {"id":"AQWMyjpA275bb9u8d3e","x":17,"y":1}], "score":0}, {"id":"AQWMyjpA275bb9u8d3e","score":0}}}
debug - websocket writing 5:::{name:"NewPos_Ghost","args": [{"id":"zYj1S60J4xgkcoVC8d3d","x":1,"y":2}, {"id":"AQWMyjpA275bb9u8d3e","x":17,"y":1}], "score":0}, {"id":"AQWMyjpA275bb9u8d3e","score":0}}}
```

Figura 5.9: Envió actualización del juego.

Actualización elementos del juego

El movimiento del personaje de cada usuario es notificado al servidor por lo que se define el evento **UpdatePosition_Player**. Con la nueva posición actualiza la posición del personaje correspondiente por medio de la función **setPosition_User**. Ademas, evalúa el numero de pasos dado por el personaje ya que si el valor es 1 o -1 indica que se ha movido a una nueva casilla del mapa de juego y es necesario actualizar la lista de nodos que contiene la variable **path**.

Finalmente envía el mensaje **NewPos_Counter** con la nueva posición al usuario contrario.

```
socket.on('UpdatePosition_Player', function(newPosition) {
    setPosition_User(socket.id, newPosition);
    if(newPosition.numPasos == 1 || newPosition.numPasos == -1) {
        for (var i = 0; i < list_user.length; i++) {
            var user = list_user[i];
            if(user.id == socket.id) {
                var myInfo = getInfoUser(socket.id)
                if (user.pos == 1) {
                    player1.path = CreatePath(myInfo, myInfo.fantasma);
                } else{
                    player2.path = CreatePath(myInfo, myInfo.fantasma);
                }
            }
        }
        socket.broadcast.to(Game.name_room).emit('NewPos_Counter',
            newPosition);
    }
});
```

Fragmento de Código 5.11: Definición del evento UpdatePosition_Player.

La figura 5.10 muestra el envío de la nueva posición al usuario contrario.

```
{ x: 5.5, y: 15, typePacman: 1, numPasos: 0.5 }
debug - broadcasting packet
debug - websocket writing 5:::{name:"NewPos_Counter","args": [{"x":5.5,"y":15,"typePacman":1,"numPasos":0.5}]}]
```

Figura 5.10: Envió nueva posición usuario.

El movimiento del personaje provoca interacción con los cocos por lo que definimos

el evento **UpdateCocos_Player** que se encarga de eliminar el elemento de la lista de cocos y reenviar la información al otro jugador por medio del mensaje **UpdateCocos_Player**.

Por ultimo, comprueba si no existen mas cocos en el juego ya que si esto ocurre se envía el mensaje **GameWinner** a los jugadores informando quien a ganado.

```
socket.on('UpdateCocos', function(cocoPosition) {
    Game.list_cocos.splice(cocoPosition,1);
    socket.broadcast.to(Game.name_room).emit('UpdateCocos_Player',
        cocoPosition);
    setScore_User(socket.id,4);
    if(Game.list_cocos.length == 0) {
        ganador = gameFinish();
        io.sockets.in(Game.name_room).emit('GameWinner',ganador);
    }
});
```

Fragmento de Código 5.12: Definición del evento UpdateCocos.

La figura 5.10 muestra el envío del coco comido por un usuario al contrario.

```
debug - broadcasting packet
debug - websocket writing 5:::{ "name": "UpdateCocos_Player", "args": [99]}
```

Figura 5.11: Envío coco comido al usuario.

5.4. Desarrollo Cliente

Los jugadores que conectan a la url <http://localhost:8181/> en la que se encuentra el servidor que entrega el fichero **index.html**¹ al navegador estableciendo la conexión WebSockets con el servidor.

Lógica de comunicación

Ahora se muestra los distintos mensajes y eventos definimos en el cliente para establecer la comunicación.

Sala de juego

El usuario introduce el nombre en el **input** de la pagina, provocando que se envíe un mensaje **stablish_connection** con el nombre para validar que no existe en la sala.

¹Apéndice A

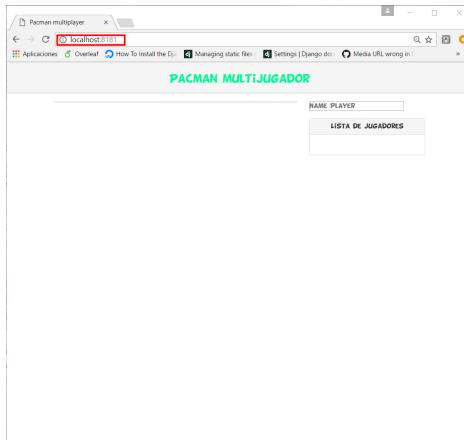


Figura 5.12: Pagina Inicio Pacman-Online.

```
function sendConnection() {
    socket.emit('stablish_connection', $('#player').val());
    $('#player').attr('disabled', true);
}
```

Fragmento de Código 5.13: Envío mensaje stablish_connection.

Se genera el evento **Reject_name** para tratar la respuesta al mensaje en caso de que el nombre no sea valido.

```
socket.on('reject_name', function(info) {
    $('#player').attr('disabled', false);
});
```

Fragmento de Código 5.14: Definición evento reject_name.

Por ultimo, se crea el evento **CreateConnection** para tratar en caso de ser afirmativa la respuesta al mensaje. Recibe el **id** de la conexión con el que se crea la instancia del objeto **Player** en la variable **myPacman** y la lista de usuario.

```
socket.on('CreateConnection', function(id, listUser) {
    listplayers = listUser;
    myPacman = new Player(id);
    for (var i = 0; i < listUser.length; i++) {
        var player = listUser[i];
        $('.list-group').append('<li id=user_'+i+' class=list-group-item>' + player.user + '</li>');
        $('#user_'+i).click(function() {requestGame(this) });
    }
});
```

Fragmento de Código 5.15: Definición evento CreateConnection.

La figura 5.13 muestra la petición de sala del primer usuario mientras que la figura 5.14 del segundo.

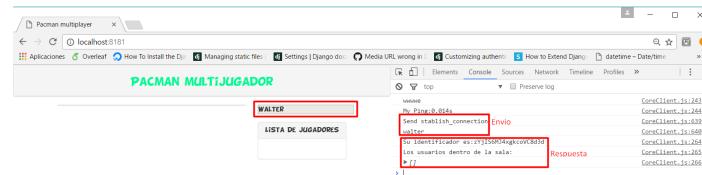


Figura 5.13: Petición/Respuesta sala 1^a jugador.

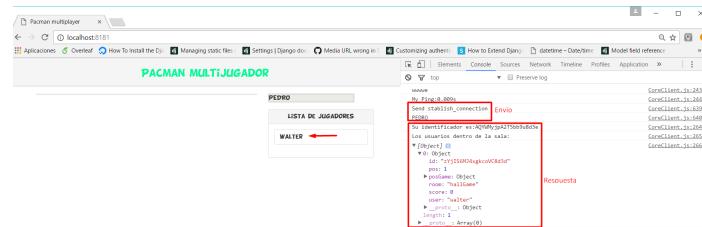


Figura 5.14: Petición/Respuesta sala 2^a jugador.

Petición de Partida

Tras entrar en la sala los usuarios pueden realizar una petición de partida a los usuarios existentes. Al seleccionar un nombre de la lista se active la función **RequestGame(this)**.

Esta función se encarga de obtener el id de la conexión del cliente seleccionado por medio de la función **SeekUser(name)** y genera un mensaje **message** cuyo cuerpo contiene el **id_origen**, **id_destino**, el texto de la petición y como subtipo de mensaje **Init**.

```
function requestGame(elemento) {
  var text=$('#player').val()+' Jugamos una partida ??';
  var destino = seekID($(elemento).text());
  var message = {
    'typeMsg':'Init',
    'idOrigen':myPacman.id,
    'texto':text,
    'idDestino':destino
  }
}
```

```
socket.emit('message', message)
$(`#li`).off('click');
}
```

Fragmento de Código 5.16: Definición función requestGame.

La imagen 5.15 muestra la petición de partida de un jugador a otro.

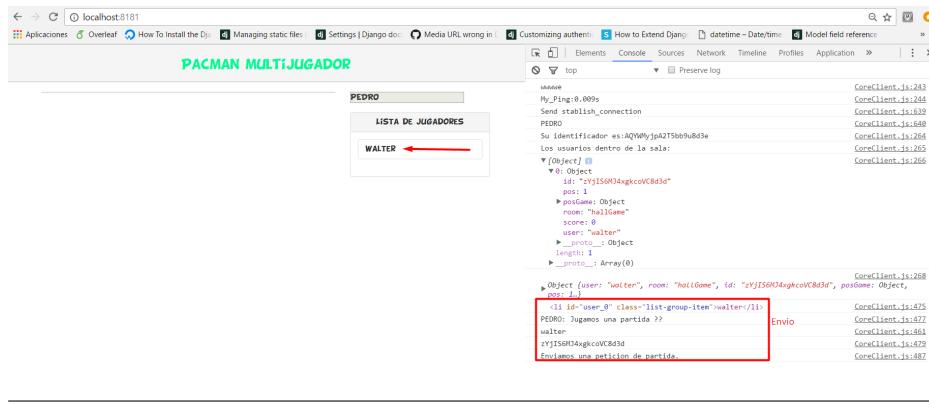


Figura 5.15: Envío petición partida.

Los mensajes de petición de partida se crean entre usuarios por lo que el servidor solo nos sirve como intermediario para encaminar el mensaje. Por ello es necesario definir el evento **message** para gestionar estos mensajes.

```
socket.on('message', function(message) {
  if(message.typeMsg == 'Init') {
    _message = message
    $('#peti').text(message.texto);
    $('#myModal').modal('show');
  } else if (message.typeMsg == 'replayInit') {
    if(message.texto == 'si') {
      $('#escena').show();
      CounterPacman = new Player(message.idOrigen);
      socket.emit('Request_ElementGame', CounterPacman.id);
    } else{
      $('#li').on('click', function() {requestGame(this)});
    }
  }
});
```

Fragmento de Código 5.17: Definicion evento message.

Para el mensaje tipo **Init** el evento se encarga de mostrar una ventana emergente con la petición recibida, figura 5.16.

El usuario receptor tiene la opción de pulsar si o no provocando que la función **ReplayGame(si/no)** gestione la respuesta. La función independientemente de la

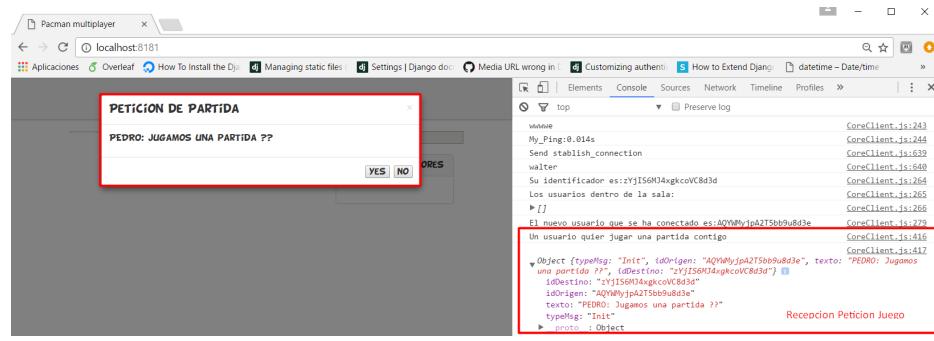


Figura 5.16: Recepción petición partida.

opción genera un mensaje **message** que contiene el **id_origen**, **id_destino**, la respuesta a la petición y como subtipo de mensaje **Replay_Init**.

Solo si la respuesta es afirmativa crea el mensaje **Request_ElementGame** dirigido al servidor para pedir los parámetros iniciales del juego.

```

function replayGame(contesta) {
  $('#myModal').modal('hide')
  var message = {
    'typeMsg': 'replayInit',
    'idOrigen': myPacman.id,
    'texto': contesta,
    'idDestino': _message.idOrigen
  };
  socket.emit('message', message)
  if(contesta == 'si'){
    $('#escena').show();
    CounterPacman = new Player(_message.idOrigen)
    socket.emit('Request_ElementGame', CounterPacman.id);
  }
}
  
```

Fragmento de Código 5.18: Definición función replayGame.

El usuario que envió el mensaje inicial recibe un mensaje **replayInit** para realizar las mismas tareas que el otro usuario, figura 5.17.

Presentación del juego

Tras el ultimo mensaje enviado por los clientes es necesario definir el evento **Response_ElementGame** para recibir el valor de los parámetros iniciales del juego. Los parámetros los obtiene de la variable **elements**, donde aquellos que tienen que ver con el escenario de juego se guardan en el objeto **GameArea** mientras que otros valores se guardan en **myPacman** y **CounterPacman** que corresponde al personaje del cliente y al del rival.

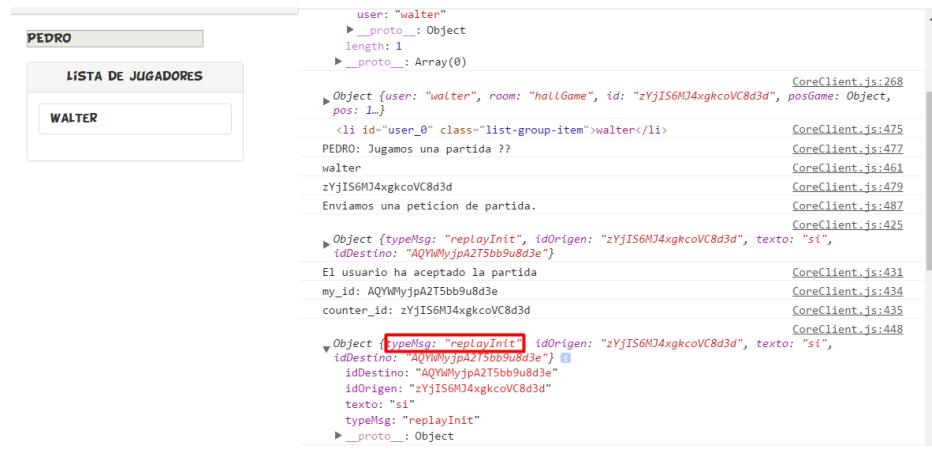


Figura 5.17: Recepción respuesta a la petición de partida.

```
socket.on('Response_ElementGame', function(elementsGame) {
    GameArea.shape_1 = elementsGame.shape_1;
    GameArea.shape_2 = elementsGame.shape_2;
    GameArea.list_obstaculos=elementsGame.obstaculos;
    GameArea.list_cocos=elementsGame.cocos;
    GameArea.properGame=elementsGame.properGame;

    myPacman.setMyghostposition(elementsGame.myInfo.fantasma);
    delete elementsGame.myInfo.fantasma;
    myPacman.setMyPosition(elementsGame.myInfo);

    CounterPacman.setMyghostposition(elementsGame.counterInfo.
        fantasma);
    delete elementsGame.counterInfo.fantasma;
    CounterPacman.setMyPosition(elementsGame.counterInfo);

    DrawScene();
    socket.emit('Finish_InitFrame');
});
```

Fragmento de Código 5.19: Definicion evento Response_ElementGame

Con esta información llamamos a la función **Drawscene()** que define las dimensiones del área del juego y dibujar el aspecto inicial del juego a través de la función **UpdateGame()**.

```
function DrawScene() {
    GameArea.canvas.width=(GameArea.properGame.nFila+2)*GameArea.
        properGame.wCuad;
    GameArea.canvas.height=(GameArea.properGame.nColum+4)*GameArea.
```

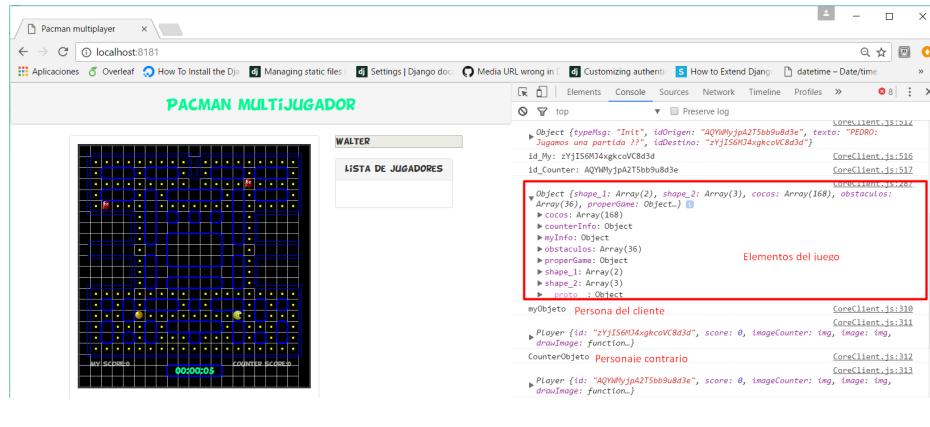
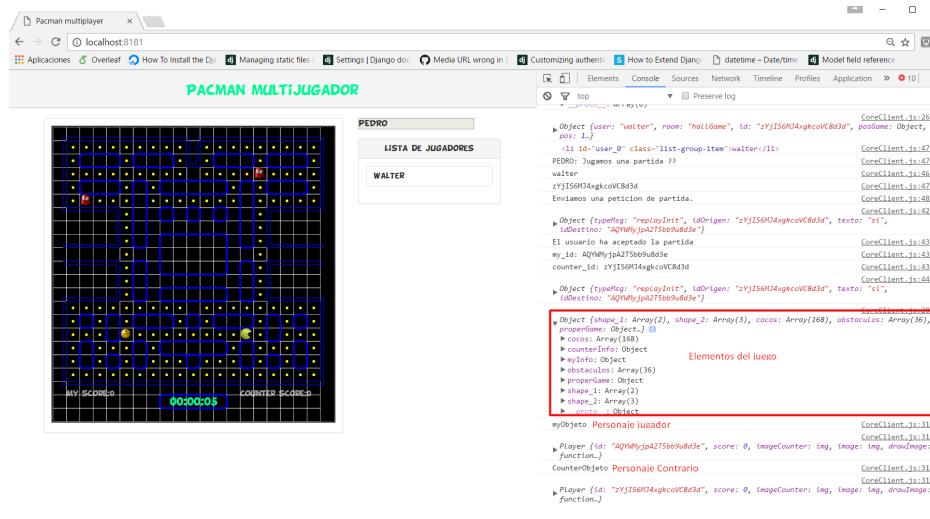
```

properGame.hCuad;
$( '#pnCanvas' ).show();
UpdateGame();
}

```

Fragmento de Código 5.20: Definición función DrawScene.

La figura 5.18 muestra la recepción de los parámetros y la visualización del juego para el primero usuario mientras la figura 5.19 se muestra el del segundo usuario.

Figura 5.18: Recepción parámetros iniciales 1^a usuario.Figura 5.19: Recepción parámetros iniciales 2^a usuario.

Tras realizar estas operaciones envía un mensaje **Finish_initFrame** para indicar al servidor que el cliente esta a la espera de empezar la partida.

Inicio del juego

Se crea el evento **ReadyGame** para recibir el mensaje de inicio de juego. En este momento se genera un evento **timeInterval** que ejecuta función **UpdateGame()** cada 1000/60 s provocando el inicio del juego.

```
socket.on('ReadyGame', function() {
  x = true;
  GameArea.frame = setInterval(UpdateGame, 1000/60);
});
```

Fragmento de Código 5.21: Definición evento ReadyGame.

Actualización del juego

El movimiento del personaje se realiza por medio de la función **NextPosPacman** que gestiona los eventos del teclado. Cada movimiento tiene que ser validado ya que esta nueva posición se envía al servidor.

Primero verifica que la nueva posición del jugador, variable **newCoord**, no sobrepase el contorno del juego por medio de la función **Game.hitt_counter(newCoord)** igual que con los obstáculos a través de función **GameArea.hitObject(newCoord)**. Si estas dos validaciones son correctas, envía un mensaje **UpdatePosition_Player** con la nueva posición.

```
function NextPosPacman(e) {
  var newCoord = {'x':0,'y':0};
  if(GameAcive) {
    var pas = 0;
    newCoord.x = myPacman.myposition.x;
    newCoord.y = myPacman.myposition.y;
    if(e.code == 'ArrowDown') {
      newCoord.y += 0.5;
      pas += 0.5;
    }else if(e.code == 'ArrowUp') {
      newCoord.y += -0.5;
      pas += -0.5;
    }else if(e.code == 'ArrowRight') {
      newCoord.x += 0.5;
      pas += 0.5;
    }else if (e.code == 'ArrowLeft') {
      newCoord.x += -0.5;
      pas += -0.5;
    }
    var HitContor = GameArea.hitt_counter(newCoord);
    if(!HitContor){
      var HitObstacle = GameArea.hitObject (newCoord);
      if(!HitObstacle){
        myPacman.myposition.x = newCoord.x;
```

```
myPacman.myposition.y = newCoord.y;
myPacman.myposition.numPasos += pas;
socket.emit('UpdatePosition_Player', myPacman.myposition);
if(myPacman.myposition.numPasos == 1 || myPacman.myposition.
    numPasos == -1) {
    myPacman.myposition.numPasos = 0;
}
var eat = GameArea.eatCoco(myPacman.myposition);
if(eat >= 0) {
    socket.emit('UpdateCocos', eat);
}
}
}
}
```

Fragmento de Código 5.22: Definición función NextPosPacman.

Ademas, comprueba si existe colisión con algún coco por medio de la función **GameArea.eatCoco(newCoord)**. En caso afirmativo envía un mensaje **UpdateCoco** con la posición del coco con el que se ha colisionado.

Las acciones anteriores las realizan los dos usuarios por lo que es necesario definir una serie de eventos para actualizar el estado del jugador contrario.

El primer evento es `socket.on('NewPos_Counter',function())` que recibe la nueva posición del usuario contrario.

```
socket.on('NewPos_Counter', function(newPosition){  
    CounterPacman.myposition = newPosition;  
});
```

Fragmento de Código 5.23: Definición evento NewPos_Counter.

El siguiente evento a definir es `socket.on('UpdateCocos_Players',function())` que recibe la posicion del coco que ha comido el usuario contrario.

```
socket.on('UpdateCocos_Player',function(cocoPosition){  
    GameArea.list_cocos.splice(cocoPosition,1);  
});
```

Fragmento de Código 5.24: Definición evento UpdateCocos_Player.

Por ultimo se define el evento `socket.on('NewPos_Ghost,function()')` que recibe actualizaciones periódicas del servidor con la posición de los fantasmas, el valor del cronometro y la puntuación de cada usuario.

```
socket.on('NewPos_Ghost', function(fantasmas,timers,scores) {
    GameArea.time = timers;
    for (var i = 0; i < fantasmas.length; i++) {
        ghost = fantasmas[i];
        core = scores[i];
```

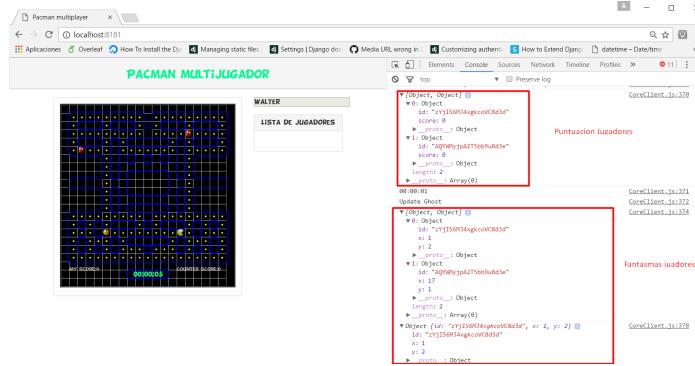
```

if (ghost.id == myPacman.id && core.id == myPacman.id) {
    myPacman.myGhost.x = ghost.x;
    myPacman.myGhost.y = ghost.y;
    myPacman.score = core.score;
} else{
    CounterPacman.myGhost.x = ghost.x;
    CounterPacman.myGhost.y = ghost.y;
    CounterPacman.score = core.score;
}
}
);

```

Fragmento de Código 5.25: Definición evento NewPos_Ghost.

En las figuras 5.20, 5.21 se muestra la recepción de los elementos del juego por cada usuario.

Figura 5.20: Recepción NewPos_Ghost 1^a usuario.Figura 5.21: Recepción NewPos_Ghost 2^a usuario.

Finalización del juego

Por ultimo, la finalización del juego lo determina el servidor ya que se encarga de comprobar si existe colisión con entre los jugadores y el fantasma. Por ello se define el evento **State_Game** que recibe **OK** o **KO** dependiendo si jugador que ha sido capturado por el fantasma o no.

```
socket.on('State_Game', function(msg) {
    GameArea.State_Game = msg;
});
```

Fragmento de Código 5.26: Definición evento State_Game.

La figura 5.22 muestra como uno de los jugadores es capturado por el fantasma por lo que pierde la partida.

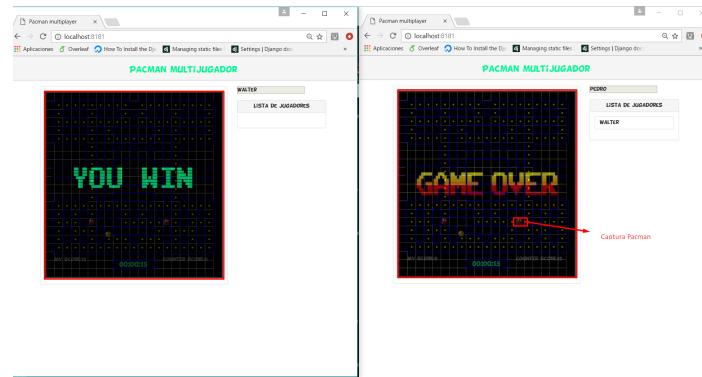


Figura 5.22: Aspecto clásico Pacman.

Otro evento necesario es **GameWinner** que recibe el ganador de la partida cuando todos los cocos del coco han sido comidos.

```
socket.on('GameWinner', function(userwinner) {
    GameArea.State_Game = userwinner;
});
```

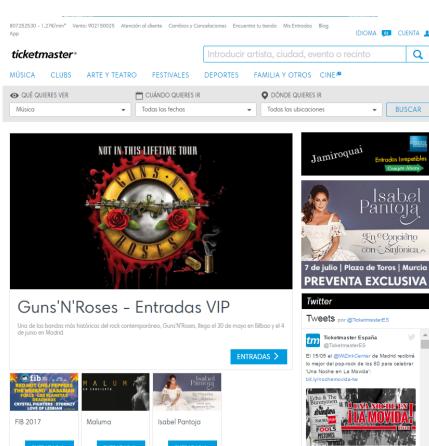
Fragmento de Código 5.27: Definición evento GameWinner.

Capítulo 6

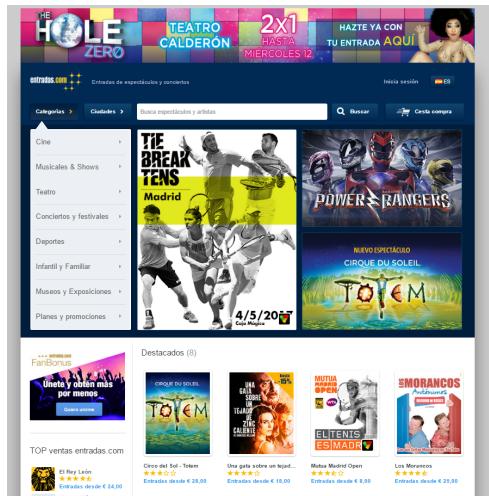
Tienda Web

6.1. Enunciado

En la actualidad los sitios web han empezado ha tener mayor presencia en Internet debido a la versatilidad y comodidad de los servicios que ofrecen a los usuarios por ejemplo TicketMaster.com o Entradas.com permiten a los usuarios adquirir entradas a distintos eventos sin necesidad de hacerlo personalmente ademas de estos ejemplos existen otros que prestan otro tipo servicio al usuario.



(a) Pagina TicketMaster.com



(b) Pagina Entradas.com

Figura 6.1: Portada Paginas Web.

Por ello, en esta tercera practica se pide desarrollar un sitio web Full-Stack, es decir, nos encargamos de crear la capa Front-End y Back-End del proyecto y conocer como interactúan entre si estas dos capas.

Funcionalidad Para gestionar el contenido de la Web sera necesario implementar como BBDD MySQL. Ademas los usuarios tienen que tener acceso a los siguientes

servicios de la aplicación y hacer uso de elementos multimedia (imágenes, audio y vídeo) para enriquecer la aplicación:

1. **Cantantes:** Se muestra información ,vídeo,discos e imágenes del cantante.
2. **Eventos:** Muestra información,localización,cantantes y entradas del evento.

Por otra parte tiene que ser capaz de gestionar los usuarios por medio de las siguientes funciones:

1. **Register:** Entrega un formulario al usuario para registrarse en la aplicación.
2. **Login:** Permite acceder al cliente siempre que se valide el contenido que rellene en el formulario de Login.
3. **Perfil Usuario:** Disponible para aquellos usuarios que hayan realizado el Login y debe mostrar la información que el usuario ha rellenado en la etapa de registro y un historial de las compras realizadas.
4. **Logout:** Permite cerrar la sesión actual del usuario.

Por último, es necesario proveer a la aplicación de un carrito de la compra basado en la sesión del usuario con el objetivo de tener un mecanismo de persistencia permitiendo realizar las siguientes acciones:

1. **Detalle del contenido:** Debe mostrar el contenido del carrito en una ventana individual donde se detalla cada uno de los productos.
2. **Actualizar contenido:** Se debe permitir la modificación del número de productos añadidos.
3. **Eliminar contenido:** Se debe permitir eliminar un producto que seleccione el cliente.

Apariencia Al tratarse de un sitio Web es necesario darle un aspecto ordenado y cuidado aparte de las funcionalidades que se ha descrito por lo que se recomienda utilizar la librería de Bootstrap.

Tecnologías Necesarias Es necesario emplear Django como tecnología core, MySQL como BBDD, WebServices , Mapas interactivos además de peticiones Ajax y formularios.

6.2. Desarrollo Back-End

Esta capa se encarga de buscar información en la BBDD de acuerdo a las peticiones que recibe y de entregar la información a los ficheros html correspondientes para que el navegador se encargue de cargarlos.

Primeramente se instalan los controladores de MySQL que permiten a Django acceder a la BBDD ejecutando los siguientes comandos `apt -get install python-dev`, `apt -get install libmysqlclient-dev` y `pip install MySQL-python`

Conexión Django-BBDD

La BBDD generada se incluye dentro del fichero `setting.py`

```
 DATABASES = { 'default': {
    'ENGINE': 'django.db.backends.mysql',
    'NAME': 'appBBDD',
    'USER': 'root',
}}
```

Fragmento de Código 6.1: añadimos la BBDD al entorno de Django.

Modelos

Hasta el momento solo se ha declarado la BBDD pero es necesario crear las tablas de la aplicación. La figura 6.2 describe el contenido de los modelos¹ diseñados y la su relación.

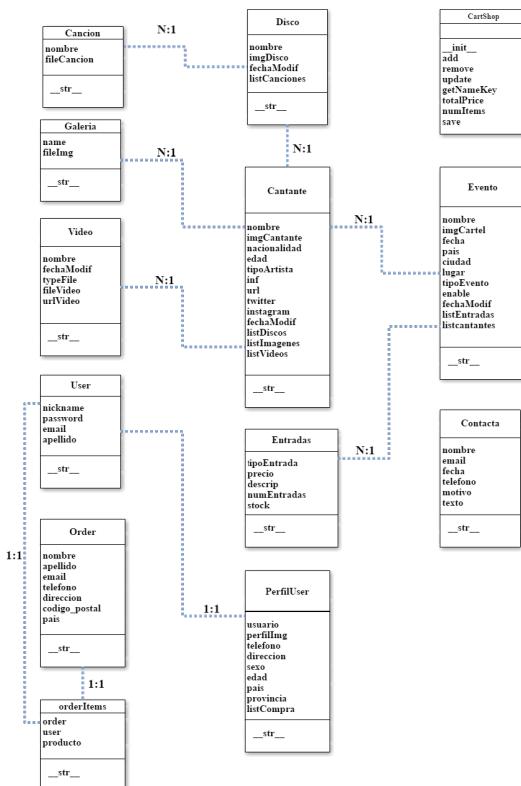


Figura 6.2: Diagrama modelos App.

Para incluir información dentro de los modelos disponemos del interfaz admin de

¹<https://github.com/RoboticsURJC-students/2015-TFG-Walter-Cuenca>

Django el cual abstrae la BBDD utiliza ademas de proporcionar un metodo mas amigable para el usuario. Se accede por medio de la url `128.0.0.1:WebMultimedia/admin/`, donde se visualizan los modelos incluidos en el fichero `admin.py`, figura 6.3.

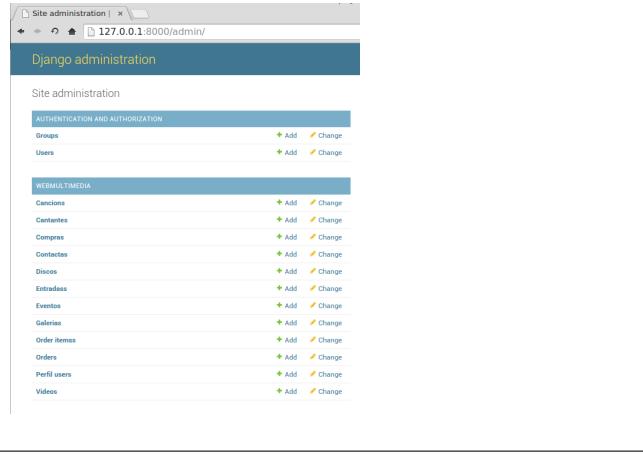


Figura 6.3: Interfaz admin Django

Formularios

El objetivo de los formularios es recolectar información proporcionada por los usuarios y validarla una vez lleguen al servidor. Por ello se genera un fichero `forms.py` con varias clases python como se ve en la figura 6.4.

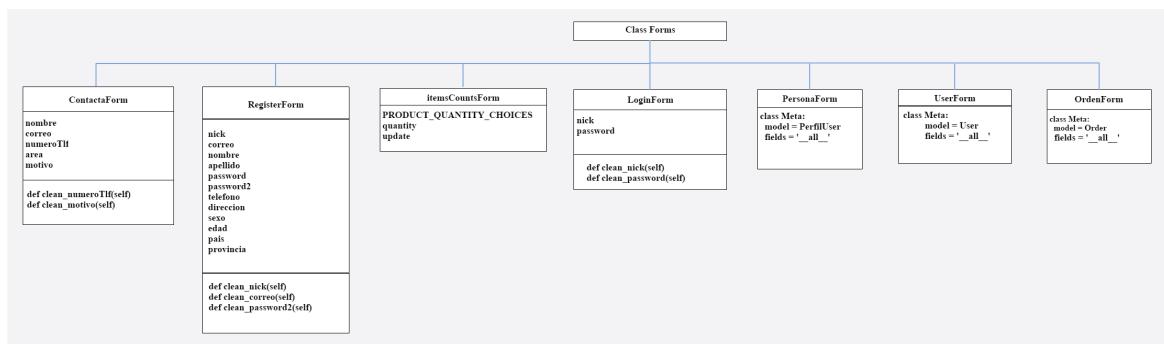


Figura 6.4: Diagrama formularios de la aplicación.

URL's

Pasamos a definir las distintas URLs de la aplicación para permitir el acceso a los usuarios. La figura 6.5 muestra las características que tiene cada URL así como

la vista que para tratar cada la petición.

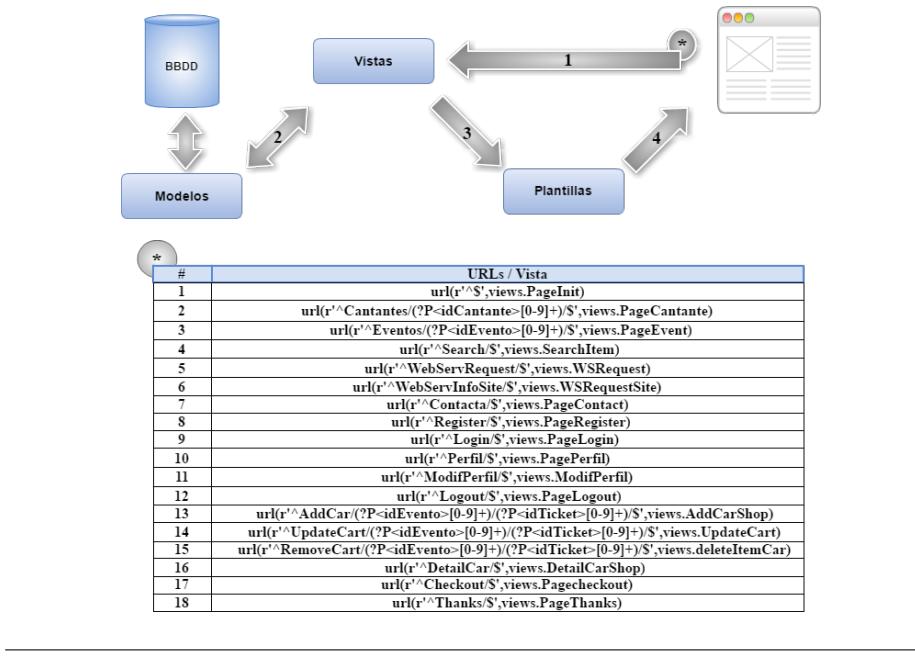


Figura 6.5: Diagrama URLs de la aplicación.

Vistas

Pasamos a detallar el ultimo componente de esta capa que se encarga de interactuar con la BBDD obteniendo y/o guardando información recibida al acceder a las distintas URLs de la aplicación. La figura 6.6 muestra las distintas vistas empleadas así como el parámetro que reciben en los casos que correspondan.

Las vistas utilizadas se han agrupado en cinco grupos según su funcionalidad, a continuación explicamos cada uno de ellos.

- 1. Home:** Se encarga de obtener los cantantes y eventos que se han incluido en la BBDD en los últimos 15 días.
- 2. Búsqueda:** Se encargan de buscar un determinado elemento en la BBDD como puede ser un Cantante, Evento o algún elemento de introducido por medio de la barra de búsqueda.
- 3. Gestión de Usuarios:** Se encargan de gestionar el registro, login/logout o el perfil de un determinado usuario.
- 4. Web Services Google:** Se encargan de gestionar peticiones al WebServices de Google Maps sobre la geolocalización de un lugar y servicios cercanos.

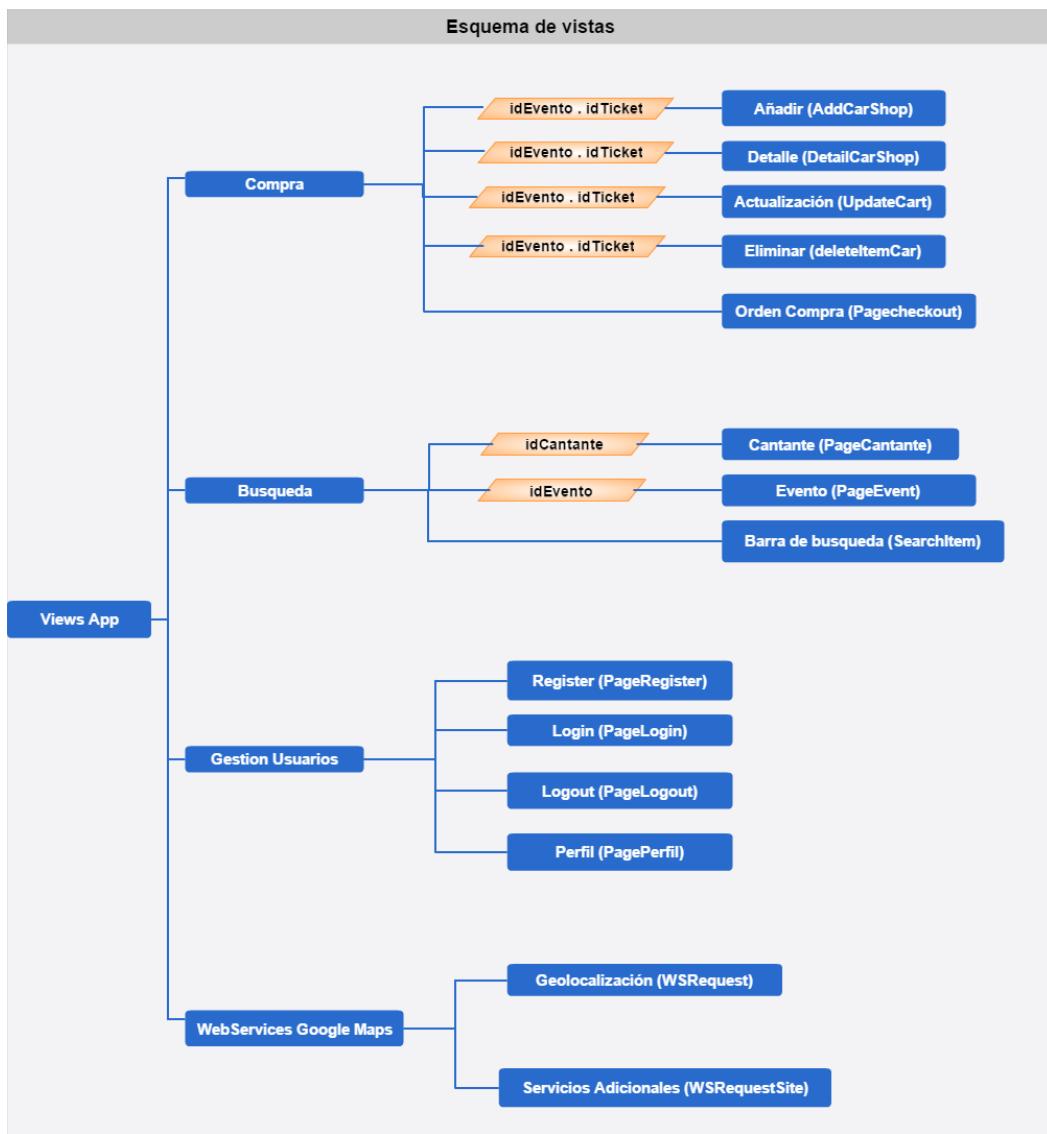


Figura 6.6: Diagrama vistas de la aplicación.

6.3. Desarrollo Front-End

Esta capa de la aplicación se encarga del diseño de los ficheros html donde se vuelca la información de las vistas. Para acceder a la información es necesario aplicar el lenguaje de plantillas de Django.

La figura 6.7 muestra cada uno de los ficheros creados para la aplicación tanto html como js.

Ahora pasamos a explicar el contenido de cada uno de los ficheros tanto la apariencia como la funcionalidad.

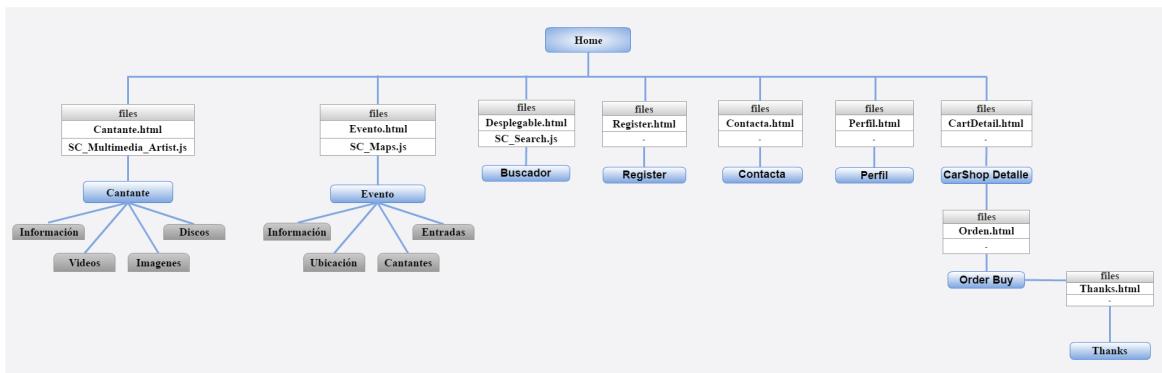


Figura 6.7: Diagrama ficheros Front-End de la aplicación.

Barra de Navegación

Es el elemento principal de la aplicación donde se encuentran los accesos a las ventanas de la aplicación. La figura 6.8 muestra la apariencia de este elemento así como los distintos accesos.



Figura 6.8: Barra de navegación.

En las siguientes subsecciones se muestra el aspecto y funcionalidad en el caso que corresponda de los distintos accesos.

Buscador

Permite a los usuarios buscar cantantes o eventos introduciendo el nombre en el buscador obteniendo como resultado un desplegable. La figura 6.9 muestra la apariencia.

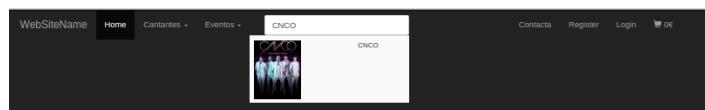


Figura 6.9: Buscador de la aplicación.

Funcionalidad: La función `sendInfo()` recupera la información introducida por el usuario y valida su longitud ya que tiene que ser mayor a dos caracteres. Tras la validación realizamos una llamada Ajax a la url `/WebMultimedia/Search/` por medio de la función `AjaxRequest(data)`.

```

function sendInfo() {
    texto = $('#textSearch').val();
    console.log(texto);
    if(texto.length > 2) {
        AjaxRequest(texto);
    } else{
        $("table#tbSearch").remove();
    }
}

function AjaxRequest(data) {
    $.ajax({
        type: "POST",
        url: '/WebMutimedia/Search/',
        data: {
            'textRequest' : data,
            'csrfmiddlewaretoken': $("input[name=csrfmiddlewaretoken]").val(),
        },
        success: resulSearch,
        dataType: 'html',
    });
}

```

Fragmento de Código 6.2: Petición Ajax buscador aplicación.

La función **resulSearch()** definida en la petición se encarga de recibir la respuesta e insertarla en la pagina.

```

function resulSearch(data, textStatus, jqXHR) {
    $('#listSearch').html(data);
}

```

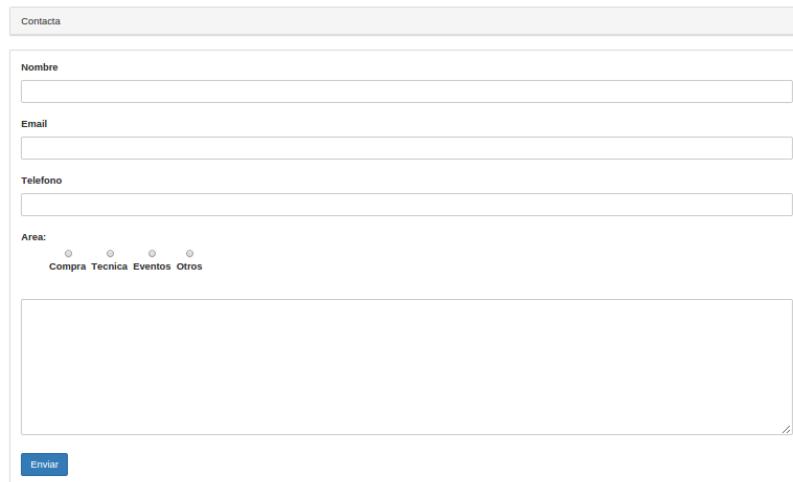
Fragmento de Código 6.3: Respuesta Ajax buscador aplicación.

Contacta

Al acceder al enlace se redirecciona al usuario a la url */WebMutimedia/Contacta* obteniendo como resultado un formulario del contacta como se ve en la figura 6.10.

Register

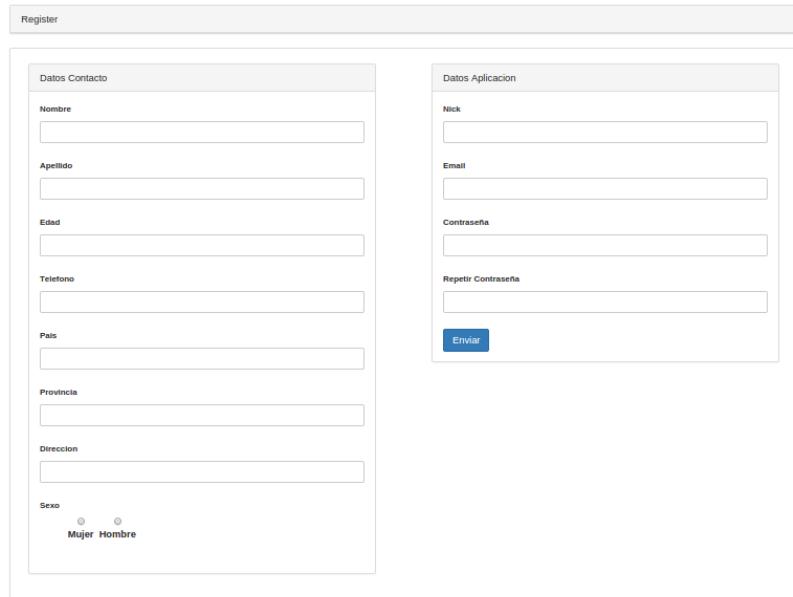
Al acceder al enlace se redirecciona al usuario a la url */WebMutimedia/Register* obteniendo como resultado el formulario de registro como se ve en la figura 6.11.



Formulario de contacto:

Nombre	<input type="text"/>
Email	<input type="text"/>
Teléfono	<input type="text"/>
Área:	<input type="radio"/> Compra <input type="radio"/> Técnica <input type="radio"/> Eventos <input type="radio"/> Otros
<input type="button" value="Enviar"/>	

Figura 6.10: Pagina Contacta.



Formulario de Registro:

Datos Contacto	
Nombre	<input type="text"/>
Apellido	<input type="text"/>
Edad	<input type="text"/>
Teléfono	<input type="text"/>
País	<input type="text"/>
Provincia	<input type="text"/>
Dirección	<input type="text"/>
Sexo	<input type="radio"/> Mujer <input type="radio"/> Hombre
Datos Aplicación	
Nick	<input type="text"/>
Email	<input type="text"/>
Contraseña	<input type="text"/>
Repetir Contraseña	<input type="text"/>
<input type="button" value="Enviar"/>	

Figura 6.11: Pagina Registro.

Login

Al pulsar el enlace se activa el formulario de la figura6.12 que contiene como campos el nombre del usuario y password que se enviara a la url */WebMultimedia/Login*.

Figura 6.12: Formulario Login.

Perfil

Al pulsar el enlace se redirecciona a la url */WebMultimedia/Perfil* obteniendo como resultado la figura 6.13 donde se muestra la información del usuario y las compras realizadas en la aplicación.

Figura 6.13: Pagina Perfil Usuario.

Logout

Al acceder a este enlace se redirecciona a la url *'/WebMultimedia/Logout'* que finaliza la sesión del usuario.

Home

La pagina principal de la aplicación se obtiene al acceder a la url *'/WebMultimedia/'* que muestra consta de un carrusel de imágenes y una sección de novedades de cantantes y eventos como se ve en la figura 6.14.

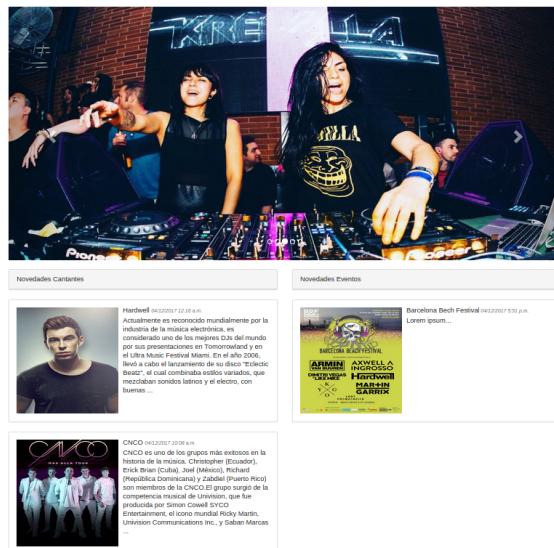


Figura 6.14: Pagina Principal.

Cantantes

El usuario selecciona un cantante del desplegable provocando la redirección a la url '/WebMultimedia/Cantante{idCantante}' que muestra la información del cantante que se distribuye en cuatro tabs que pasamos a explicar.

Información

Contiene información del cantante además de su foto como se ve en la figura 6.15.

Vídeos

Contiene una lista con los vídeos de los que dispone la aplicación sobre el cantante como muestra la figura 6.16.

Funcionalidad: Para permitir que los usuarios seleccionen un vídeo de la lista creamos la función '*loadVideo(file, tipo)*' y lo vinculamos a cada uno de los elementos. La función evalúa el tipo de archivo del que se trata ya que puede ser <iframe> o <video> y así reproducirlo a través del elemento adecuado.

```
function loadVideo(file, tipo) {
  if(tipo == 'iframe') {
    $('iframe').attr("src", file);
    $("video").hide();
    $("iframe").show();
  } else{
    var path ='/media/' +file;
```

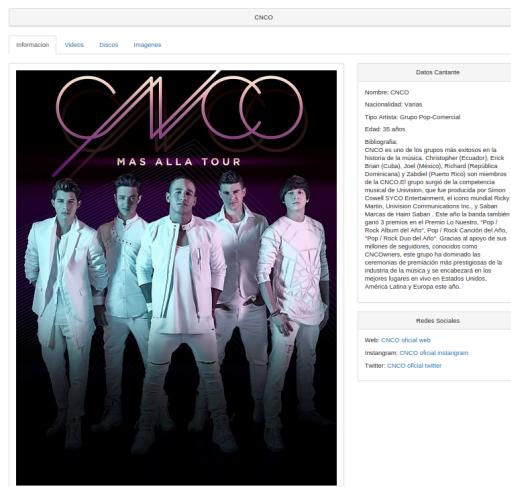


Figura 6.15: Pagina Cantante panel Información.

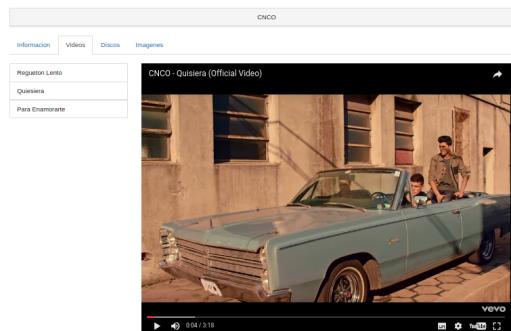


Figura 6.16: Pagina Cantante panel Información.

```

$( 'video' ).attr( "src", path );
$( "iframe" ).hide();
$( "video" ).show();
}
}

```

Fragmento de Código 6.4: Función carga de videos.

Discos

Contiene los discos relacionados con el cantante además de una lista de reproducción para escuchar las canciones de un disco tras ser seleccionado. La apariencia se muestra en la figura 6.17.

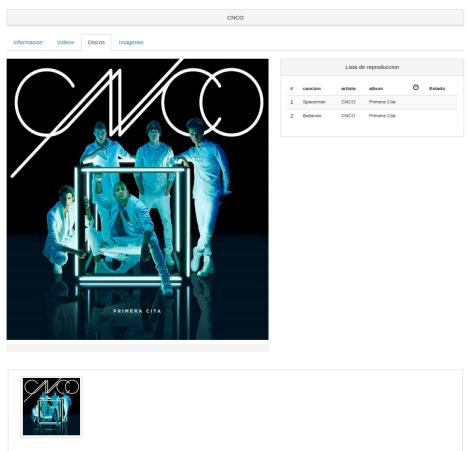


Figura 6.17: Pagina Cantante panel Discos.

Funcionalidad: Los usuarios al seleccionar provocan la llamada a la función **loadList**). La función carga la imagen del disco y las canciones en la lista de reproducción.

```
function loadList(elemento, imagen) {
    $('#ImgRepro').attr('src', '/media/' + imagen);
    var idElemento = $(elemento).attr("id");
    var divListSong = $('#' + idElemento).siblings('div');
    var idDivListSong = $(divListSong).attr("id");
    $("table#x").remove();
    $('#' + idDivListSong).children('table').clone().appendTo($('#x'));
}
```

Fragmento de Código 6.5: Función carga lista de reproducción.

Para reproducir una canción de la lista se emplea la función **loadSong()**. La función se encarga de obtener el elemento a reproducir al igual que el tiempo duración en la lista de producción.

```
function loadSong(name, idElement, urlSong) {
    var idTimer = 'timer_' + idElement;
    var state_Actual = 'state_' + idElement;
    var path = '/media/' + urlSong;
    var audio = document.getElementById('repro');
    if(!audio.paused) {
        audio.pause();
    }
    if (state_Actual != stado_Old && stado_Old != '') {
        document.getElementById(stado_Old).innerHTML = '';
    }
    audio.src = path;
```

```

audio.onloadeddata=function() {
  var str_Time = convertSeg_Min(audio.duration);
  document.getElementById(state_Actual).innerHTML =
    'Reproduciendo';
  document.getElementById(idTimer).innerHTML = str_Time;
  audio.play();
  stado_Old = state_Actual;
}
}

```

Fragmento de Código 6.6: Función reproducir canción.

Imágenes

Contiene un conjunto de imágenes en forma de galeria como se ve en la figura 6.18.

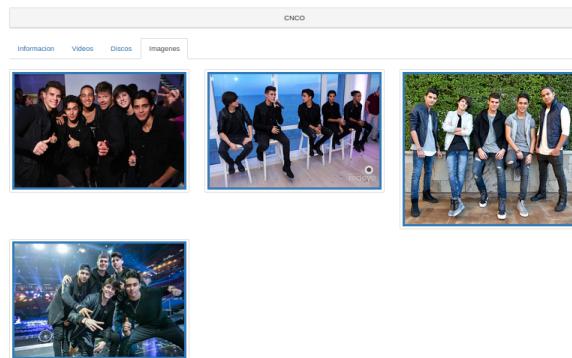


Figura 6.18: Pagina Cantante pestaña imágenes.

Eventos

Al seleccionar un evento del desplegable redirecciona al usuario a la url '/Web-Multimedia/Eventos/idevento/' que muestra la información en cuatro paneles que pasamos a explicar.

1. Información

Este panel contiene varia información sobre el evento como se ve en la figura 6.19.

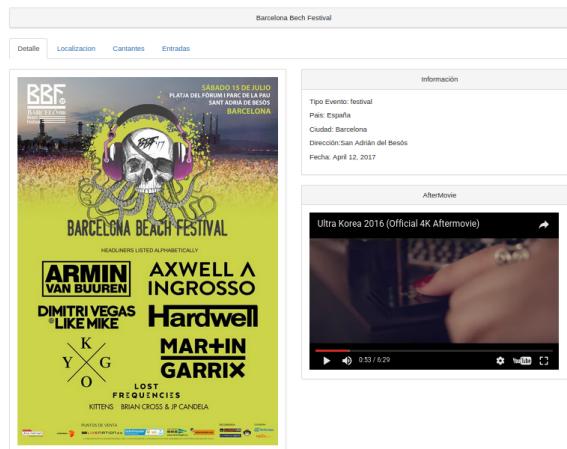


Figura 6.19: Pagina Evento panel Información.

2. Ubicación

Muestra la ubicación del evento y los servicios(hoteles,restaurantes,...) alrededor del lugar en un mapa como se ve en la figura 6.20.

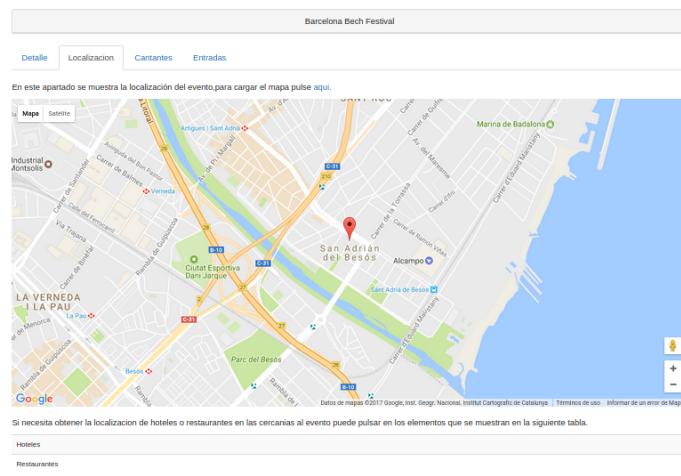


Figura 6.20: Pagina Evento panel ubicación.

Funcionalidad: Se realiza una llamada Ajax a la url `/WebMutimedia/WebServRequest/` con el nombre del lugar. Definimos la función `searchSuccess` para recuperar la respuesta del servidor.

```
$ .ajax({
  type: "POST",
  url: "/WebMutimedia/WebServRequest/",
```

```

data: {
  'site' : $('#direct').text(),
  'csrfmiddlewaretoken': $("input [name=csrfmiddlewaretoken]").val
    () ,
},
success: searchSucess,
dataType: 'html',
});

```

Fragmento de Código 6.7: Petición Ajax Ubicación Lugar.

La función **searchSucess** parsea a formato JSON el contiene de la variable **data** para obtener la información de las coordenadas accediendo a **geometry.location**.

```

function searchSucess(data, textStatus, jqXHR) {
  var infoRequest = JSON.parse(data);
  coordenadas = infoRequest.results[0].geometry.location;
  WarchMap();
}

```

Fragmento de Código 6.8: Respuesta Ajax Ubicación Lugar.

La función **WarchMap()** se encarga de crear el mapa realizando una instancia de **google.maps.Map()** que recibe como parámetro el lugar de la pagina donde se visualizara y sus propiedades que se definen en la variable **mapProp**. Finalmente con las coordenadas realiza una instancia de **google.maps.Marker** asociándolo al mapa con el objetivo de indicar claramente el lugar del evento.

```

function WarchMap() {
  var lat = parseFloat(coordenadas.lat);
  var long = parseFloat(coordenadas.lng);
  var mapProp = {
    center: new google.maps.LatLng(lat, long),
    zoom: 15,
    mapTypeId: google.maps.MapTypeId.ROADMAP,
  };
  map = new google.maps.Map(document.getElementById("Maps"), mapProp);
  ;
  var marker = new google.maps.Marker({
    position: { lat: parseFloat(coordenadas.lat), lng: parseFloat(
      coordenadas.lng) },
    draggable: true,
    animation: google.maps.Animation.BOUNCE,
    map: map,
    title: 'Concierto'
  });
}

```

Fragmento de Código 6.9: Creación Mapa con la Ubicación.

3. Cantantes

El panel contiene imágenes de los cantantes que tienen relación con el evento seleccionando como se puede ver en la figura 6.21.

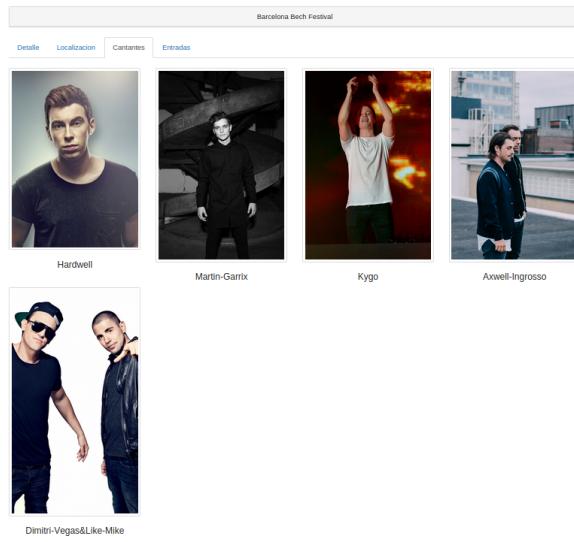


Figura 6.21: Pagina Evento panel Cantantes.

4. Entradas

El panel contiene las entradas de las que dispone el evento además de una pequeña descripción como se puede ver en la figura 6.22.

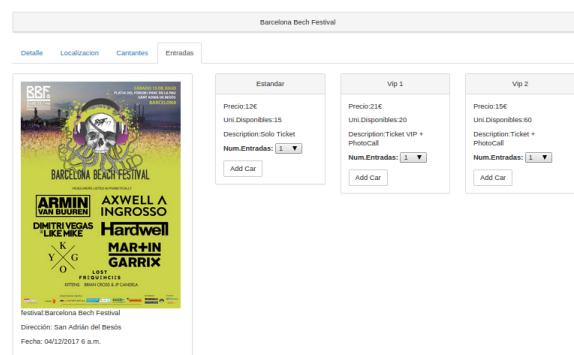


Figura 6.22: Pagina Evento pestaña Entradas.

Funcionalidad: Los usuarios añaden contenido al carrito de la compra por medio del botón **Add Car** disponible en cada tipo de entrada que redirecciona a la url

/WebMultimedia/AddCar/evento.id/ticket.id provocando que se añade este elemento al carrito de la compra del usuario.

Carrito de la Compra

Es necesario permitir a los usuarios visualizar el detalle de su compra por lo que al pulsar sobre el enlace de la barra de navegación accedemos a la url '*/WebMultimedia/DetailCar/*' mostrando cada uno de los productos como se ve en la figura 6.23.

Detalle compra						
Imagen	Producto	Cantidad	Update	Eliminar	Valor Total	
	Barcelona Beach Festival Vip 1	1	<input type="button" value="1"/> <input type="button" value="Update"/>	<input type="button" value="Remove"/>	21€	
	Barcelona Beach Festival Estandar	2	<input type="button" value="1"/> <input type="button" value="Update"/>	<input type="button" value="Remove"/>	24€	
Total:						45€
<input type="button" value="Checkout"/>						

Figura 6.23: Pagina Detalle Carrito.

Funcionalidad:Dentro del detalle de la compra es necesario proveer de varias acciones que explicamos a continuación para que el usuario pueda realizar modificaciones del contenido de su carrito.

1. **Actualizar:** Al seleccionar un nuevo numero de entradas y pulsar '**Update**' se redirecciona la petición a la url '*WebMultimedia/UpdateCar/idEvento/idTicket/*' que se encarga llevar acabo la acción.
2. **Eliminar:** Al pulsar '**Remove**' se redirecciona la petición a la url '*WebMultimedia/RemoveCart/idEvento/idTicket/*'
3. **Checkout:** Permite pasar a tramitar la orden de compra de los producto redireccionando la acción a la url '*WebMultimedia/Checkout/*'.

Orden de Compra

Esta ventana es visible cuando el usuario desea terminar la compra del contenido de su carrito ya que muestra un resumen de los elementos añadidos y un formulario con los datos del usuario para generar la orden . El aspecto de esta pagina se muestra la figura 6.24.

Orden de compra	Su orden
Nombre <input type="text"/>	Producto Barcelona Beach Festival-Barcelona Beach Festival
Apellido <input type="text"/>	Precio 21€ Vip 1 x 1
Email <input type="text"/>	Barcelona Beach Festival-Barcelona Beach Festival
Teléfono <input type="text"/>	Entrenar x 2
Dirección <input type="text"/>	Total 42€
Código Postal <input type="text"/>	
País <input type="text"/>	
<input type="button" value="Enviar"/>	

Figura 6.24: Pagina Orden Compra.

Capítulo 7

VideoConferencia usando WebRTC

7.1. Enunciado

Las aplicaciones multimedia en tiempo real permiten conectar a distintos usuarios a través de la red e intercambiando información y contenido de audio, vídeo. Un ejemplo de este tipo de aplicación es Skype que ha tenido mucho éxito ya que permite a sus usuarios establecer videollamadas e intercambiar información una vez se haya instalado el software.



Figura 7.1: Skype interfaz videollamada.

En esta ultima practica se pide desarrollar una aplicación web similar a Skype que permita a los usuarios conectarse entre si a través de la url de la aplicación.

Requisitos Los usuario dentro de la aplicación podrán seleccionar los elementos multimedia (audio y vídeo) que desean compartir y se les permitirá seleccionar o crear una sala donde realizar la videoconferencia. Dentro de la sala dispondrán de un chat y la opción de intercambiar ficheros por medio de WebRTC.

Como mecanismo adicional sera necesario el desarrollo de un servidor de señalización para la fase de comunicación inicial de WebRTC.

Tecnologías En el desarrollo de la práctica es necesario utilizar NodeJS, WebSockets, API File y WebRTC.

7.2. Diseño

La figura 7.2 muestra el esquema utilizado en el desarrollo de esta práctica. Contiene cada uno de los elementos que intervienen desde el proceso de señalización hasta que se convierte en una comunicación entre pares.

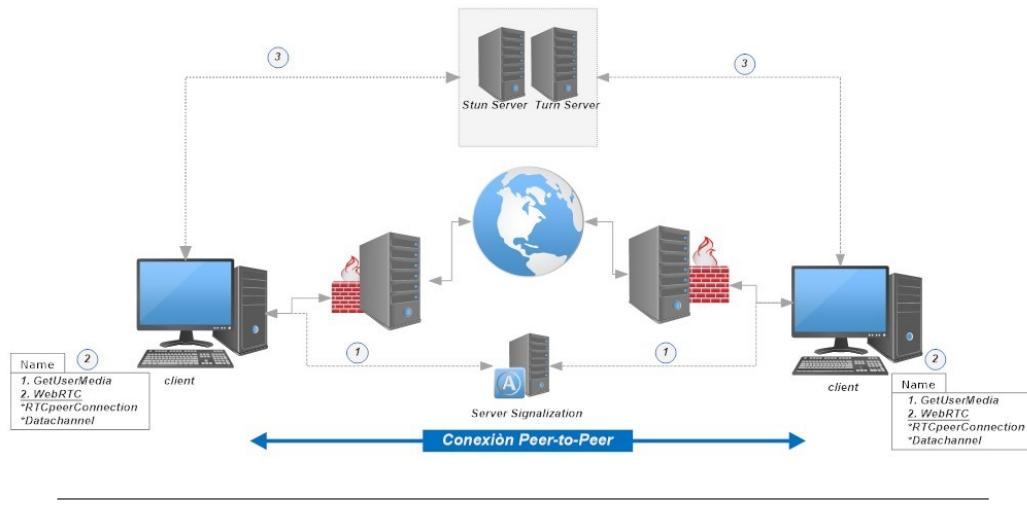


Figura 7.2: Creación oferta cliente.

La topología de comunicación final entre los miembros de una sala se indica en la figura 7.3.

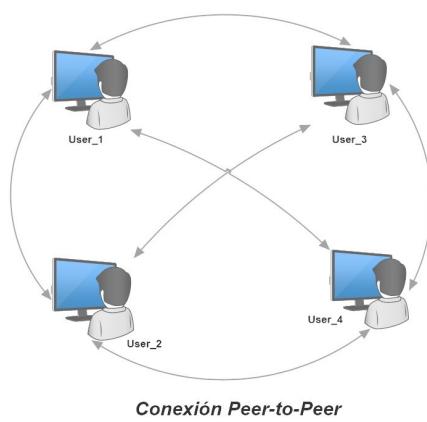


Figura 7.3: Conexión Peer-to-Peer final.

7.3. Desarrollo Servidor Señalización

Para crear el servidor nos apoyamos en las librerías `node-static`, `http` y `socket.io`. La figura 5.3 muestra al servidor encendido con la instancia de WebSockets operativa.

```
var static = require('node-static');
var http = require('http');
var file = new(static.Server)();
var app = http.createServer(function (req, res) {
    file.serve(req, res);
}).listen(8181);
var io = require('socket.io').listen(app);
```

Fragmento de Código 7.1: Creación Servidor de Señalización.

Las tareas que realiza el servidor se divide en Inicio de Conexión y Mensajes señalización que se detallan a continuación.

Inicio de conexión

Esta etapa se encarga de gestionar el acceso de los usuarios en una sala de la aplicación. Se define el evento `InfoRoom` que envía un mensaje `ReplayInfoRoom` con la lista de salas existentes.

```
socket.on('infoRoom', function() {
    socket.emit('ReplayInfoRoom', listRooom);
});
```

Fragmento de Código 7.2: Request/Replay lista de salas existentes.

La figura 7.4 muestra como el servidor realiza el envío de este mensaje.

```
C:\Users\waltercito\Desktop\tfgPractica4\Chat_multi_usuario>node server.js
Servidor escuchando en el puerto:8181
  info - socket.io started
  debug - served static content /socket.io.js
  debug - client authorized
  info - handshake authorized X_PYhV3DxQ-4IVXLff7_
  debug - setting request GET /socket.io/1/websocket/X_PYhV3DxQ-4IVXLff7_
  debug - set heartbeat interval for client X_PYhV3DxQ-4IVXLff7_
  debug - client authorized for
  debug - websocket writing 1:
Se ha establecido una conexión con un browser.
Recibimos un mensaje InfoRoom.
Enviamos un mensaje ReplayInfoRoom:
  debug - websocket writing 5:::{ "name": "ReplayInfoRoom", "args": [ [ { "name": "streaming", "state": "" } ] ] }
```

Figura 7.4: Request/Replay lista de salas del Servidor.

Además define el evento `Establish_connect` para obtener el nombre del usuario y el de sala a la que el usuario quiere acceder. Con el nombre de la sala comproueba su existencia por medio de la función `getRoom(nameRoom)` en caso de

no existir la guarda en la lista de salas y finaliza comprobando que la sala no este llena. Finaliza enviando un mensaje `CreateStream` con el identificador de conexión y un mensaje `NewJoined` al resto de usuario de la sala para que conozcan la existencia del nuevo miembro.

```
socket.on('stablish_connection', function(name, room) {
  if(!getRoom(room)){
    setRoom(room, '');
  };
  var numClients = io.sockets.clients(room).length;
  if(numClients < 3){
    socket.username = name;
    socket.room = room;
    socket.join(room);
    socket.emit('CreateStream', socket.id);
    socket.broadcast.to(room).emit('New_Joined', socket.id);
  }else{
    socket.emit('RejectStream', socket.id);
  }
});
```

Fragmento de Código 7.3: Request/Replay del establecimiento de conexión.

La figura 7.5 muestra como el servidor realiza el envío de este mensaje.

```
Node.js command prompt - node server.js
info  - socket.io started
debug - served static content /socket.io.js
debug - client authorized
info  - handshake authorized BaZ9vRqB2IhCT5Zphump
debug - setting request GET /socket.io/1/websocket/BaZ9vRqB2IhCT5Zphump
debug - set heartbeat interval for client BaZ9vRqB2IhCT5Zphump
debug - client authorized for
debug - websocket writing 1::
Se ha establecido una conexión con un browser.
Recibimos un mensaje InfoRoom.
Enviamos un mensaje ReplayInfoRoom:
debug - websocket writing 5:::{ "name": "ReplayInfoRoom", "args": [ [ { "name": "streaming", "st
debug - emitting heartbeat for client BaZ9vRqB2IhCT5Zphump
debug - websocket writing 2::
debug - set heartbeat timeout for client BaZ9vRqB2IhCT5Zphump
debug - got heartbeat packet
debug - cleared heartbeat timeout for client BaZ9vRqB2IhCT5Zphump
debug - set heartbeat interval for client BaZ9vRqB2IhCT5Zphump
Recibimos un mensaje stablish_connection.
**user: walter
**room: prueba
Comprobación número de clientes: 0
debug - websocket writing 5:::{ "name": "CreateStream", "args": [ "BaZ9vRqB2IhCT5Zphump" ] }
debug - broadcasting packet
```

Figura 7.5: Request/Replay conexión al Servidor.

Mensajes señalización WebRTC

El momento en que se quiere establecer la conexión WebRTC el servidor tiene como función encaminar los mensajes entre los participantes de la conexión. Cada uno de los mensajes de señalización viajan dentro del mensaje message por lo que se crea un evento con este nombre que consulta el id destino para encaminar el mensaje correctamente.

```
socket.on('message', function(message, room) {
  io.sockets.socket(message.id_dest).emit('message', message);
});
```

Fragmento de Código 7.4: Mensajes de señalizacion.

La figura 7.6 encamina la oferta, la figura 7.7 encamina la respuesta y la figura 7.8 encamina los IceCandidate.

```
Reenvio de mensaje al nodo destino.
{ id_origen: '5kJjLJVGceTNeh08jgfG',
  id_dest: 'DiBsxSj_bEjTropijgfF',
  message:
    { type: 'answer',
      sdp: 'v=0\r\no=- 5055778763332693161 2 IN IP4 127.0.0.1\r\ns=-\r\nnt=0 0\r\nna=group:BUNDLE audio video data\r\nna=
msid-semantic: WMS jbymQZhoiZkkKkHJXg0VYWL6VbgQpy8cZav\r\nnm=audio 9 UDP/TLS/RTP/SAVPF 111 103 104 9 0 8 106 105 13 1
26\r\nnc=IN IP4 0.0.0.0\r\nna=rtcp:9 IN IP4 0.0.0.0\r\nna=ice-ufrag:rBs\r\nna=ice-pwd:Q9YEenNWishN3wa0vTnye/000\r\nna=fing
erprint:sha-256 DB:ED:29:B9:EC:3D:49:66:8B:FF:11:EF:F6:65:4F:2B:A1:E3:C3:59:8B:D8:51:EE:51:5C:A1:60:1C:85:6A:45\r\nna=
```

Figura 7.6: Mensaje Answer.

```
Reenvio de mensaje al nodo destino.
{ id_origen: 'DiBsxSj_bEjTropijgfF',
  id_dest: '5kJjLJVGceTNeh08jgfG',
  message:
    { type: 'offer',
      sdp: 'v=0\r\no=- 3179353706946817949 2 IN IP4 127.0.0.1\r\ns=-\r\nnt=0 0\r\nna=group:BUNDLE audio video data\r\nna=
msid-semantic: WMS SSLaCsdjwFD0UaJgAxRsZdi2zj6oGZ8DBDdN\r\nnm=audio 9 UDP/TLS/RTP/SAVPF 111 103 104 9 0 8 106 105 13 1
26\r\nnc=IN IP4 0.0.0.0\r\nna=rtcp:9 IN IP4 0.0.0.0\r\nna=ice-ufrag:rBs\r\nna=ice-pwd:Q9YEenNWishN3wa0vTnye/000\r\nna=fing
erprint:sha-256 DB:ED:29:B9:EC:3D:49:66:8B:FF:11:EF:F6:65:4F:2B:A1:E3:C3:59:8B:D8:51:EE:51:5C:A1:60:1C:85:6A:45\r\nna=
```

Figura 7.7: Mensaje Offer.

Tras terminar el envío de los mensajes de señalización el servidor es transparente entre los usuarios ya que empieza la comunicación Peer-to-Peer.

7.4. Desarrollo Cliente Peer-to-Peer

Los usuarios se conectan al servidor a través de la url <http://localhost:8181> en un navegador que recibe el fichero inicial de la aplicación. Cuando el fichero se carga se crea la instancia de Websockets y se pide al usuario que introduzca el nombre que utilizará en la aplicación.

```
Reenvio de mensaje al nodo destino.
{ id_origen: 'DiBsxBj_bEjTropijgff',
  id_dest: '5kJjLJVGceTNeh08jgfG',
  message:
    { type: 'iceCandidate',
      label: 1,
      id: 'video',
      candidate: 'candidate:288073040 1 udp 2122255103 2001::5ef5:79fd:1868:1ede:d1e4:e4e4 61342 typ host generation 0
ufrag CKHe network-id 1 network-cost 50' } }
  debug - websocket writing 5:::{ "name": "message", "args": [ {"id_origen": "DiBsxBj_bEjTropijgff", "id_dest": "5kJjLJVGceTNeh08jgfG", "message": { "type": "iceCandidate", "label": 1, "id": "video", "candidate": "candidate:288073040 1 udp 2122255103 2001::5ef5:79fd:1868:1ede:d1e4:e4e4 61342 typ host generation 0 ufrag CKHe network-id 1 network-cost 50" } ] }
```

Figura 7.8: Mensaje Icecandidate.

Inicio de conexión

El usuario al acceder a la aplicación envía un mensaje `infoRoom` al servidor para obtener la lista de salas disponibles por lo que se define el evento `ReplayInfoRoom` para recibir la lista de salas disponibles.

```
socket.on('ReplayInfoRoom', function(listRoom) {
  for(var i = 0; i < listRoom.length; i++) {
    var room = listRoom[i];
    $('#listRoom').append('<li><a id=' + room.name + '>' + room.name + '</a></li>');
    $('#' + room.name).click(function() {
      nameRoom = $(this).text();
      attachmentElements();
    });
  }
});
```

Fragmento de Código 7.5: Creación desplegable de salas.

Dentro de la aplicación el usuario puede seleccionar los elementos multimedia que desea compartir. Tras la selección de los elementos selecciona la sala que desea unirse enviando un mensaje `stablish_connection` con el nombre del usuario y el de la sala.

```
socket.emit('stablish_connection', name, nameRoom);
```

Fragmento de Código 7.6: Envío mensaje inicio conexión.

Para tratar la respuesta al mensaje se define el evento `CreateStream`.

```
socket.on('CreateStream', function(id) {
  my_id = id;
});
```

Fragmento de Código 7.7: Recepción respuesta inicio conexión.

Mientras que para los otros usuarios se define el evento `New_Joined` con el identificador de conexión del nuevo usuario para iniciar el proceso de señalización

```
socket.on('New_Joined', function(id) {
  id_newUser = id;
  create_connection(id_newUser);
});
```

Fragmento de Código 7.8: Incluir elementos multimedia remotos 1.

La figura 7.9 muestra el envío y la recepción de cada uno de los mensajes mencionados en esta sección.

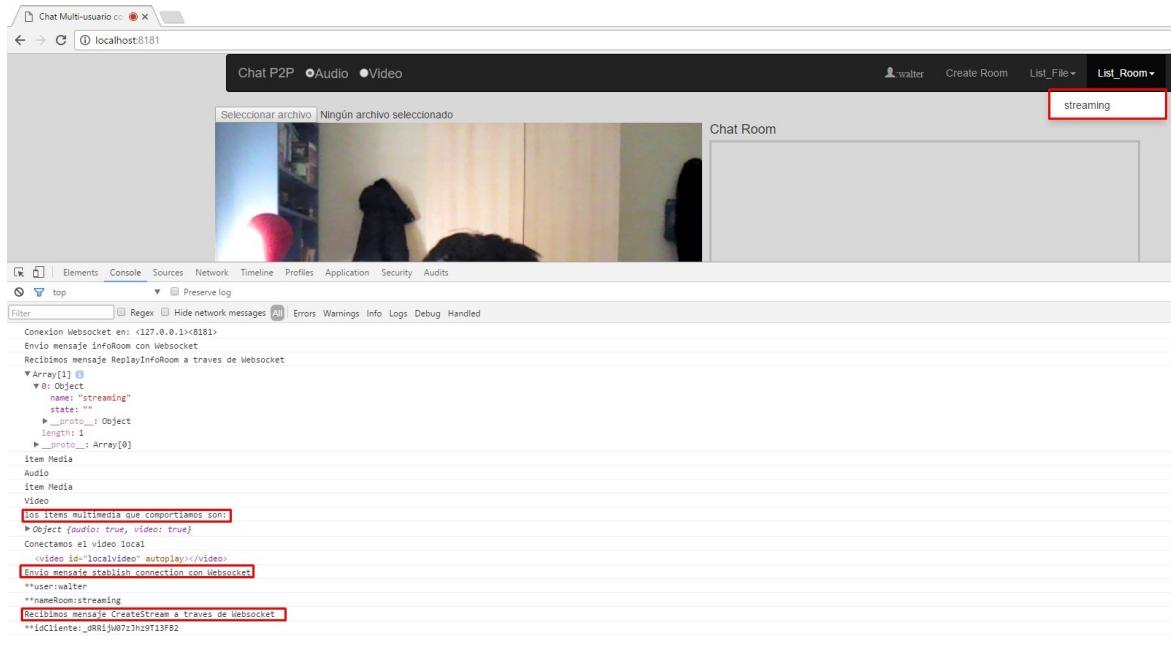


Figura 7.9: Inicio de conexión.

Proceso señalización WebRTC

A partir del ultimo mensaje empieza el proceso de señalización que se divide en tres etapas: Oferta, Respuesta y Icecandidate. Es necesario definir la configuración del protocolo ICE en la variable pc_config antes de iniciar este proceso.

```
var pc_config = {'iceServers': [{url: 'stun:stun.l.google.com:19302'}]};
```

Fragmento de Código 7.9: Configuración del protocolo ICE.

Ofera

La función `create_connection` inicia el proceso de oferta generando la instancia del objeto `RTCPeerConnection()` con la configuración del protocolo ICE

definida en la variable `pc_config`. Además vincula el flujo de vídeo local y el remoto a través del método `addStream()` y del evento `onaddstream`.

También define el canal de datos a través del método `createDataChannel` con los eventos necesarios para trabajar con este canal. Por ultimo genera la oferta a través del método `createOffer()` guardando su descripción de sesión con el método `setLocalDescription` y enviando este mensaje al nuevo usuario.

```

function create_connection(id) {
  var pc = new RTCPeerConnection(pc_config, {});
  var num_user = 'user_'+ list_user.length;
  new_remote(num_user);
  pc.addStream(streaming);

  pc.onaddstream = function(event) {
    var video = document.querySelector('#'+num_user);
    if(webrtcDetectedBrowser == 'firefox'){
      video.mozSrcObject = event.stream;
    }else{
      video.srcObject = event.stream;
    }
    video.play();
  };

  var sendChannel = pc.createDataChannel("sendDataChannel",
    {reliable: true});
  list_send.push(sendChannel);
  sendChannel.onopen = ChannelOpen;
  sendChannel.onclose = ChannelClose;
  sendChannel.onmessage = ChannelReceive;

  pc.createOffer(function(sessionDescription) {
    pc.setLocalDescription(sessionDescription);
    var message = create_msg(my_id,id(newUser,sessionDescription));
    socket.emit('message',message);
    },function(err){console.log(err);},{}
  );

  pc.onicecandidate = SendICEcandidate;
  list_user.push({id:id(newUser),peer:pc,data:sendChannel});
}

```

Fragmento de Código 7.10: Vinculamos vídeo local/remoto a RTCPeerConnection.

La figura 7.10 muestra la creación de los elementos mencionados anteriormente dentro del objeto `RTCPeerConnection`.

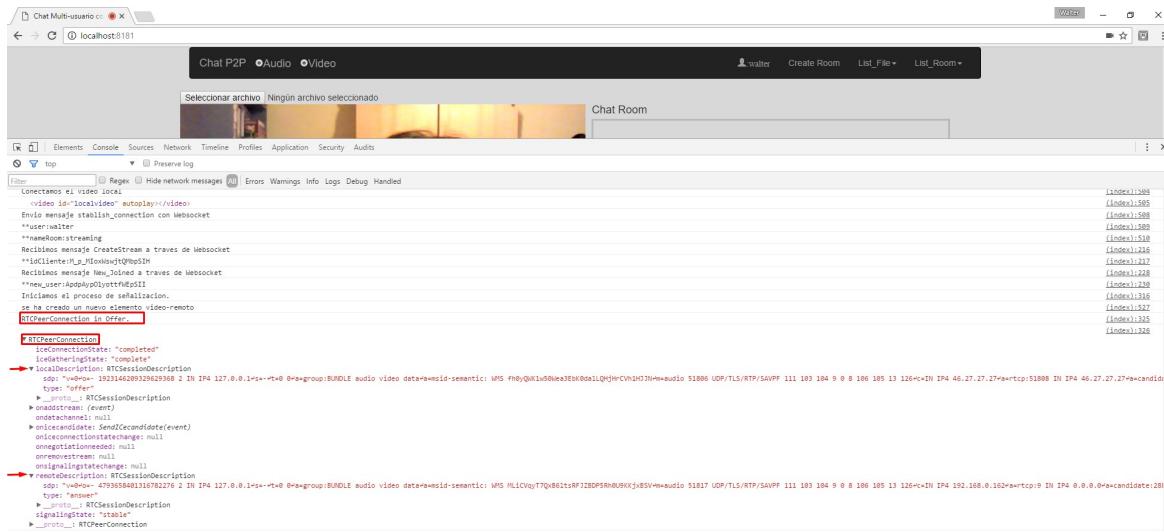


Figura 7.10: Creación oferta cliente.

Answer

EL otro usuario recibe un mensaje de oferta con la descripción de sesión del otro usuario. Al igual que en la oferta crea una instancia `RTCPeerConnection` en la que define los eventos `onaddstream`, `ondatachannel`, `onicecandidate`. Además crea un nuevo objeto `RTCSessionDescription` con la descripción recibida que se guarda como descripción remota por medio del método `setRemoteDescription`.

Finaliza creando el mensaje de respuesta con el método `createAnswer` añadiendo su descripción de sesión a través del método `setLocalDescription` y envía dicho mensaje al cliente.

```
if(message.message.type == 'offer') {
    var pc = new RTCPeerConnection(pc_config, {});
    id_newUser= message.id_origen;
    pc.addStream(streaming);
    pc.setRemoteDescription(new RTCSessionDescription(message.
        message));
    pc.onicecandidate = SendICecandidate;
    pc.onaddstream = function(event){
        var num_user = 'user_'+ list_user.length;
        new_remote(num_user);
        var video = document.querySelector('#'+num_user);
        if(webRTCDetectedBrowser == 'firefox'){
            video.mozSrcObject = event.stream;
        }else{
            video.srcObject = event.stream;
        }
        video.play();
        list_user.push({id:message.id_origen, peer:pc, data:receiveChannel
    }
}
```

```

    });
}

pc.ondatachannel= function(event) {
    list_send.push(event.channel);
    var receiveChannel = event.channel;
    receiveChannel.onmessage = ChannelReceive;
    receiveChannel.onopen = ChannelOpen;
    receiveChannel.onclose = ChannelClose;
}

pc.createAnswer(function(sessionDescription) {
    pc.setLocalDescription(sessionDescription);
    var msg = create_msg(my_id,message.id_origen,sessionDescription
    );
    socket.emit('message',msg);
},function(err){
    console.log(err);
},{});
}

```

Fragmento de Código 7.11: Recepción de la oferta.

La figura 7.11 muestra como se crea la instancia RTCPeerConnection con los eventos que se han mencionado.

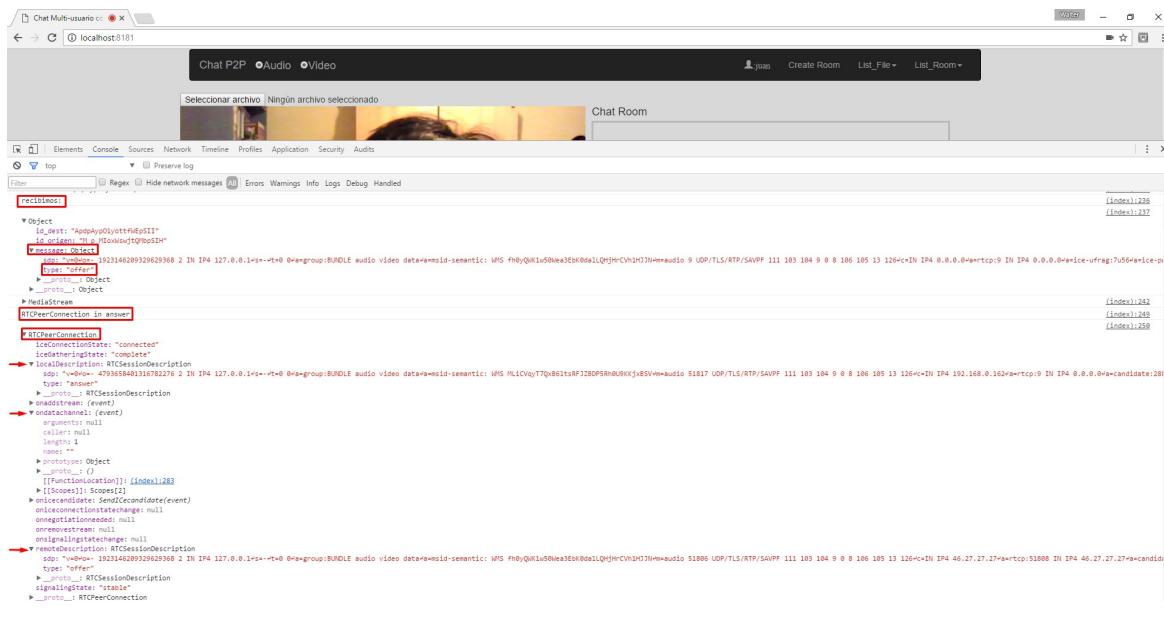


Figura 7.11: Creación de la respuesta.

Icecandidate

Tanto el usuario local y como el remoto tras la generación del objeto RTCPeerconnection necesitan obtener información de la Ip:Puerto disponibles para la conexión por

medio del protocolo ICE. Cuando se encuentra un candidato se ejecuta el evento `onicecandidate` que compone el mensaje con la información y lo envía a través de un mensaje de tipo `message`.

```
pc.onicecandidate = SendIceCandidate;
function SendIceCandidate(event) {
  if(event.candidate) {
    var ice = {type: 'iceCandidate',
      label: event.candidate.sdpMLineIndex,
      id: event.candidate.sdpMid,
      candidate: event.candidate.candidate
    };
    var msg = {id_origen:my_id,id_dest:id_newUser,message:ice}
    socket.emit('message',msg);
  }
}
```

Fragmento de Código 7.12: Envío candidatos.

Los usuarios que reciben la información anterior generan un objeto `RTCIceCandidate` para llamar a la función `addIcecandidate` que se encarga de buscar dentro de la lista de conexiones la correspondiente y así guardar el objeto creado a través del método `addIceCandidate`.

```
var candidate = new RTCIceCandidate({
  sdpMLineIndex:message.message.label,
  candidate:message.message.candidate
});
addIceCandid(message.id_origen,candidate);
function addIceCandid(id,message) {
  for(var i=0;i<list_user.length;i++) {
    var user = list_user[i];
    if(user.id == id) {
      user.peer.addIceCandidate(message);
    }
  }
}
```

Fragmento de Código 7.13: Recepción de candidatos.

Al conocer los usuarios la información de red y la descripción de sesión de cada uno esta etapa termina dando lugar a la conexión Peer-to-Peer.

Conexión WebRTC

En esta etapa los usuarios intercambiando flujo de vídeo y audio en una conexión entre pares lo que permite que el canal de datos (`RTCDATAChannel`) se encuentre operativo para enviar información por él como se describe a continuación.

Envió de cadena de texto

A través del chat de la sala se puede enviar cadena de caracteres mediante la función `send_data()`. La función genera un mensaje que contiene el flag `chat` y la cadena de texto para enviarlo por los distintos canales de comunicación disponibles.

```
function send_data(elemento) {
    var msg = $(elemento).val();
    var data = JSON.stringify({info:'chat',data:name+':'+msg});
    for(var i=0;i<list_send.length;i++){
        list_send[i].send(data)
    }
}
```

Fragmento de Código 7.14: Envío datos del chat.

La figura 7.12 es un ejemplo del envío de la cadena de texto tecleada por un usuario y enviada a los demás.

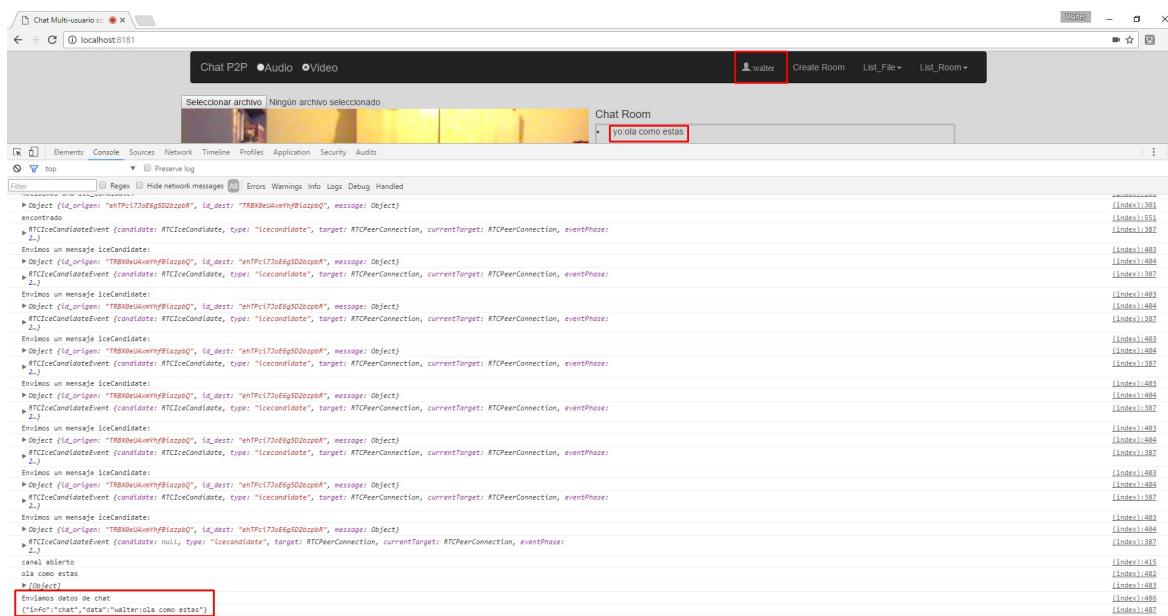


Figura 7.12: Envío de caracteres del chat por RTCDatChannel.

La recepción se realiza por medio de la función `WriteChat(_data)` que se encarga de presentar dentro del chat la información que se ha recibido de otro usuario.

```
function WriteChat (_data) {
    var line = document.createElement('li');
    var textnode = document.createTextNode(_data.data);
    line.appendChild(textnode);
    $('#texto').append(line);
}
```

Fragmento de Código 7.15: Recepción datos del fichero.

Envío de ficheros

Los usuarios disponen de un input de tipo file con el que carga el fichero que desea compartir. Tras seleccionar el fichero se ejecuta la función `processFiles(file)` que obtiene el contenido del fichero a través del objeto `FileReader()`.

```
function processFiles(file) {
  var files = file[0];
  type = files.type;
  name_fich = files.name;
  var reader = new FileReader();
  reader.onload = function (e) {
    var data_file = reader.result;
    data_encrypt = arrayBufferToBase64(data_file);
    send_chucky();
  };
  reader.readAsArrayBuffer(files);
}
```

Fragmento de Código 7.16: Lectura del fichero.

El envío de los datos se realiza por medio de la función `send_chucky()` en pequeños fragmentos de longitud fija ya que no sabemos la longitud del archivo y con el fin de no saturar el canal lo enviamos de esta forma. Cada envío está formado por el flag `file` y el fragmento del archivo correspondiente, una vez se ha enviado se programa el siguiente envío mediante el evento timer `setTimeout(sendChuncky, time)`.

Al enviar el fragmento final del fichero se añade información adicional del fichero como el nombre y tipo de fichero ya que esta información es necesario para que el usuario receptor pueda reconstruir el fichero.

```
function send_chucky() {
  var last = false;
  fin = inicio + size_data;
  if(fin < data_encrypt.length) {
    var data = JSON.stringify({info:'file', data:data_encrypt.slice(
      inicio, fin)});
    for(var i=0;i<list_send.length;i++) {
      var user = list_send[i];
      user.send(data);
    }
    inicio = fin;
    setTimeout(send_chucky, 100);
  }else{
    last = true;
    var more_info ={type:type, name:name_fich};
```

```
var data = JSON.stringify({info:'file',end:last,data:  
    data_encrypt.slice(inicio, data_encrypt.length),more:  
    more_info});  
for(var i=0;i<list_send.length;i++){  
    var user = list_send[i];  
    user.send(data);  
}  
inicio = 0;  
}  
}
```

Fragmento de Código 7.17: Envío de fragmentos del archivo.

La figura 7.13 muestra la obtención de información del fichero seleccionado y el envío de cada fragmento del fichero hasta finalizar.

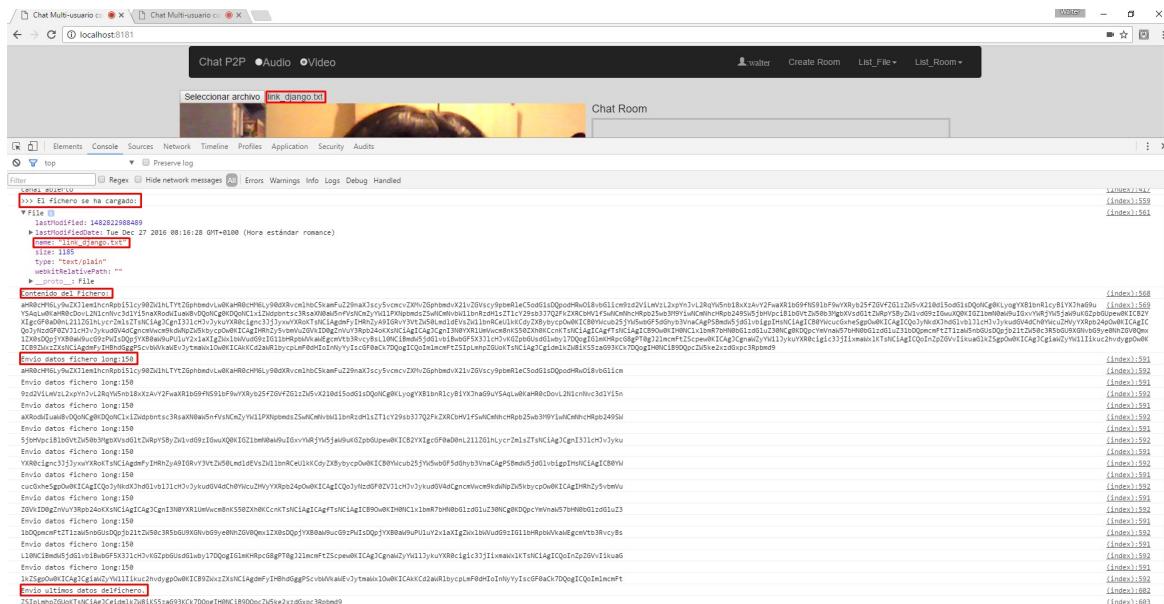


Figura 7.13: Envío información del fichero con RTCDataChannel.

El usuario receptor acumula cada uno de los fragmentos que recibe hasta obtener el ultimo fragmento mediante la función `BuildField()`. Al obtener el ultimo fragmento pasa a reconstruir el documento a través dentro de una etiqueta `<a>`, donde el atributo `href` esta formado por el tipo de archivo concatenado a los fragmentos del archivos.

```
function BuildFile(_data) {
blob += _data.data;
if (_data.end != undefined) {
if (_data.more.type == 'text/plain') {
var link = document.createElement('a');
link.href = 'data:' + data.more.type +';base64,' + blob;
```

```
link.target = '_blank';
link.download = _data.more.name;
var textnode = document.createTextNode(_data.more.name);
link.appendChild(textnode);
$("#listFile").append(link);
}
blob =' ,';
}
}
```

Fragmento de Código 7.18: Recepción y reconstrucción del fichero .

La figura 7.14 muestra la recepción de cada fragmento enviado por un usuario y la reconstrucción del fichero. *figur*.

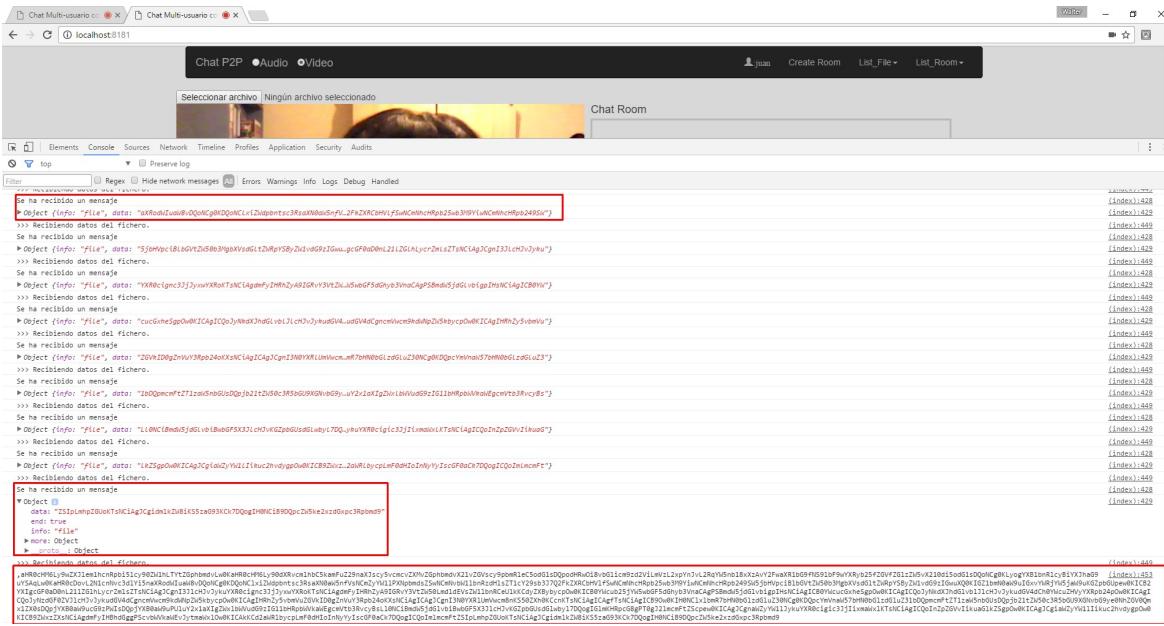


Figura 7.14: Recepción y reconstrucción del fichero.

Capítulo 8

Conclusiones

Este capítulo recoge las conclusiones y futuros trabajos una vez los desarrollos del TFG han concluido y poniendo en balance si se los requisitos iniciales han sido cubiertos.

8.1. Conclusiones

Si echamos una mirada atrás se puede ver que se ha creado satisfactoriamente cada uno de los enunciados y el desarrollo de las prácticas destinadas al entorno docente en las que utilizamos múltiples tecnologías web que se marco como objetivo principal. Dentro de este objetivo nos marcamos cuatro subjetivos, los cuales también hemos cumplido:

1. La aplicación del juego del ComeCocos se ha conseguido crear el juego a través de tecnologías Web del clientes sin necesidad de plugins externos.
2. La aplicación multijugador del ComeCocos se ha conseguido implementar la comunicación entre el servidor del juego y los clientes permitiendo establecer una partida entre jugadores satisfactoriamente.
3. La aplicación de un sitio Web de una tienda se ha desarrollado con Django y utilizando como BBDD MySQL.
4. La aplicación de Videoconferencia con WebRTC se ha desarrollado correctamente permitiendo a los usuarios conectarse entre si a través de un navegador permitiendo intercambiar flujo de vídeo, audio o ficheros por medio de tecnologías Web.

Se puede encontrar tanto esta memoria, como el repositorio del código, vídeos, explicaciones, ejemplos y resultados obtenidos en la mediawiki oficial del proyecto¹

¹<http://jderobot.org/Walter-tfg>

8.2. Trabajos futuros

Este TFG habré las puertas al uso de tecnologías Web de ultima generación. A medida que los desarrollos fueron avanzando se veían puntos de mejora en cada una de las practicas ya sea de forma visual o aportando mayor funcionalidad aunque no influida en llegar a los objetivos marcados por lo que la mejora de cada una de las practicas puede ser una linea de futuros trabajos.

Aunque una linea de trabajo atractivo para el futuro seria fusionar lo aprendiendo con estas practicas para crear un proyecto que abarque estas tecnologías en un solo desarrollo. Esto podría ser un juego multijugador como el ComeCocos en el que se utiliza en el lado del cliente JS y HTML5 y como mecanismos de comunicación multimedia WebRTC apoyándose en su canal de datos para información del juego entre los usuarios además que de esta forma permitiría a los jugadores verse mientras hacen uso de la aplicación. A esto habría que sumarle la utilización de WebSockets en la fase inicial de WebRTC y enriquecer la apariencia del juego con WebGL.

Bibliografía

- [1] Curso de CSS3 de W3C http://www.w3schools.com/css/css3_intro.asp
- [2] Página oficial WebGL <http://get.webgl.org/>
- [3] Edgar Barrero. Desarrollo de una aplicación web para sistema domótico. Trabajo Fin de Grado, Grado en Ingeniería de Sistemas Audiovisuales y Multimedia, Universidad Rey Juan Carlos, 2013-2014.
- [4] Mediawiki Edgar Barrero (Surveillance 5.1) <http://jderobot.org/Aerobeat-colab>
- [5] Aitor Martínez. Drone con WebRTC. Trabajo Fin de Grado, Grado en Ingeniería Telemática, Universidad Rey Juan Carlos, 2015-2016.
- [6] MediaWiki Aitor Martínez (JdeRobotWebClients) [wikihttp://jderobot.org/Aitormf-tfg](http://jderobot.org/Aitormf-tfg)
- [7] Iván Rodríguez-Bobada Martín. Tecnologías web en plataforma robotica Jde-Robot. Grado en Ingeniería de Sistemas Audiovisuales y Multimedia, Universidad Rey Juan Carlos, 2015-2016.
- [8] MediaWiki Iván Rodríguez-Bobada Martín (DronWebRTC) [wikihttp://jderobot.org/Irodmar-tfg](http://jderobot.org/Irodmar-tfg)
- [9] Mediawiki Walter Cuenca (Prácticas docentes de desarrollo web.) <http://jderobot.org/Walter-tfg>
- [10] Repositorio de Walter Cuenca (Prácticas docentes de desarrollo web.) <https://github.com/RoboticsURJC-students/2015-TFG-Walter-Cuenca>
- [11] The HTML 5 JavaScript <http://html5index.org/index.html>
- [12] Tutorial Canvas https://developer.mozilla.org/es/docs/Web/Guide/HTML/Canvas_tutorial
- [13] Etiquetas de audio/vídeo HTML5 https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Video_and_audio_content

- [14] Tutorial API File <https://www.html5rocks.com/en/tutorials/file/dndfiles/>
- [15] Introducción WebSockets <http://html5index.org/index.html>
- [16] Ejemplo WebSockets <http://blog.teamtreehouse.com/an-introduction-to-websockets>
- [17] Tutorial WebRTC <https://www.tutorialspoint.com/webrtc/webrtcTutorial.pdf>
- [18] Tutorial JavaScript <http://www.vc.ehu.es/jiwotvim/ISOFT2009-2010/Teoria/BloqueIV/JavaScript.pdf>
- [19] Tutorial Jquery <http://www.cav.jovenclub.cu/comunidad/datos/descargas/jQuery.pdf>
- [20] Web oficial Jquery <http://contribute.jquery.org/documentation/>
- [21] Transparencias Bootstrap UCM <https://www.fdi.ucm.es/profesor/jpavon/web/26-Bootstrap.pdf>
- [22] Web oficial Bootstrap <http://getbootstrap.com/>
- [23] Libro NodeJS Introducción a Nodejs a través de Koans ebook
- [24] Teoría Base de Datos <https://si.ua.es/es/documentos/documentacion/office/access/teoria-de-bases-de-datos.pdf>
- [25] Teoría WebServices http://usershop.redusers.com/media/blfa_files/lpcu104/capitulogratis.pdf
- [26] Teoría tipos de WebServices <http://vis.usal.es/rodrigo/documentos/sisdis/teoria/4-serviciosWeb.pdf>
- [27] Libro Django <http://bibing.us.es/proyectos/abreproy/12051/fichero/libros%252Flibro-django.pdf>