Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Intensificación en Telemática

Tecnologías para la creación de aplicaciones web interactivas y dinámicas

Autor: Alejandro Manuel López Rodríguez

Tutor: Francisco José Fernández Jiménez

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Tecnologías para la creación de aplicaciones web interactivas y dinámicas

Autor:
Alejandro Manuel López Rodríguez

Tutor:
Francisco José Fernández Jiménez
Profesor colaborador

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Tecnologías para la creación de aplicaciones web interactivas y dinámicas

Autor: Alejandro Manuel López Rodríguez

Tutor: Francisco José Fernández Jiménez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A mi familia y amigos

Agradecimientos

En primer lugar, quisiera agradecer a mi tutor Francisco, por haber estado ahí cuando lo necesitaba y por proporcionarme las herramientas necesarias para realizar el trabajo.

En segundo lugar a mi familia, que sin ellos todo esto no hubiera sido posible. A mi padre, que me ha ayudado con la corrección del trabajo; a mi madre, fuente de inspiración; a mis hermanos Diana y Miguel, que me han proporcionado ideas y mejoras del mismo. También doy las gracias a mi abuela y tío, que siempre están ahí para lo que haga falta y me han apoyado desde el comienzo de esta etapa, que ya acaba.

Por otro lado, quisiera agradecer a todos mis compañeros de clase, que hemos sufrido esta carrera desde el primer curso, pero a la vez disfrutado.

También a la gente que he conocido en Sevilla en estos cuatro años, a Dani, Loren, Pablo, Rafa y compañía.

Por último, agradecer a mis compañeros incondicionales Fran y Jesús, que siempre hemos estado juntos desde el comienzo.

Alejandro Manuel López Rodríguez

Sevilla, 2016

Resumen

El objetivo de este trabajo es el análisis de tecnologías que permiten la creación de aplicaciones web interactivas en tiempo real. Para ello se estudiará el modelo push, haciendo hincapié en el protocolo WebSocket. Se estudiarán dos Frameworks para su manejo, como son Atmosphere y Orbiter.

También se estudiarán otros protocolos que mejoran la experiencia de usuario en la red, como es HTTP/2.

Por otro lado, otro de los objetivos del proyecto es tomar contacto con aplicaciones basadas en Java, la familiarización con proyectos web dinámicos y el aprendizaje de tecnologías como Java Server Faces (JSF).

Por último, se realizará una aplicación web práctica para poner en vigor todo lo aprendido durante esta investigación.

Abstract

The aim of this work is the analysis of technologies that allow the creation of interactive web applications in real time. For this, the push model will be studied, emphasizing in the WebSocket protocol. Also, the Frameworks Atmosphere and Orbiter will be useful to manage comunications between clients.

In addition, other protocols that improve the user experience in the network, as is HTTP/2 will be studied too.

On the other hand, another objective of the project is to make contact with Java-based applications, familiarization with dynamic web projects and learning technologies such as Java Server Faces (JSF).

Finally, a practical web application will be made to enforce everything learned during this investigation.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Los científicos estudian el mundo tal como es;
los ingenieros crean el mundo que nunca ha sido

Theodore von Karman, 1881

En la actualidad las aplicaciones web tienen un papel esencial y la interactividad con ellas es algo muy habitual, ya sea para consultar el correo electrónico, ya sea para realizar una compra online o incluso para planear unas vacaciones. Por tanto, es conveniente que la forma en la que el usuario interactúe con la web sea rápida, eficiente, además de que tenga una respuesta activa y visual.

Existen aplicaciones web muy dinámicas, donde la usabilidad y la velocidad de navegación son muy buenas; a la vez que existen algunas páginas donde este aspecto es bastante mejorable. Por consiguiente, un buen diseño, donde se tenga constancia de estos aspectos, es esencial; así, se deben usar las tecnologías necesarias para que su funcionamiento sea óptimo, a la vez de eficiente en la utilización de recursos (ancho de banda, memoria, carga en el servidor, etc.).

1.1 Introducción al modelo push

El modelo clásico de comunicación entre cliente y servidor está basado en las peticiones en el lado del cliente, y las respuestas por parte del servidor, como se observa en la siguiente imagen:

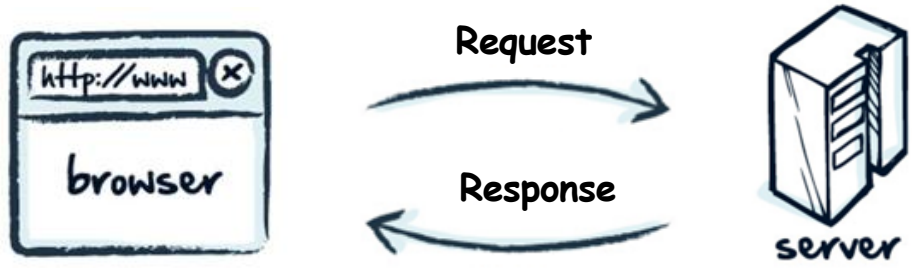


Figura 1 – Modelo clásico de comunicación cliente-servidor

Sin embargo, esta forma de comunicación no es del todo eficiente para aplicaciones interactivas en las que intervengan muchos clientes; ya que serían necesarias múltiples y continuas peticiones al servidor para observar los cambios en todos los clientes.

Un claro ejemplo de este aspecto podría ser un grupo de chat, al que se puedan conectar varios clientes. Cada vez que un cliente envíe un mensaje, el servidor procesará el mismo y actualizará el contenido de la página mostrándolo, siendo necesario una nueva petición (refresco de página) por parte del resto de clientes para observar dicho mensaje.

Por tanto, este esquema de comunicación entre cliente y servidor es mejorable para aplicaciones web interactivas.

El modelo push, también conocido como server push, es una apuesta novedosa que trata de mejorar estas cuestiones y conseguir, así, una comunicación efectiva entre el servidor y los clientes. En este esquema, será el servidor el que realice las peticiones a los clientes, agilizando de esta manera el proceso, como se ve en la siguiente imagen:

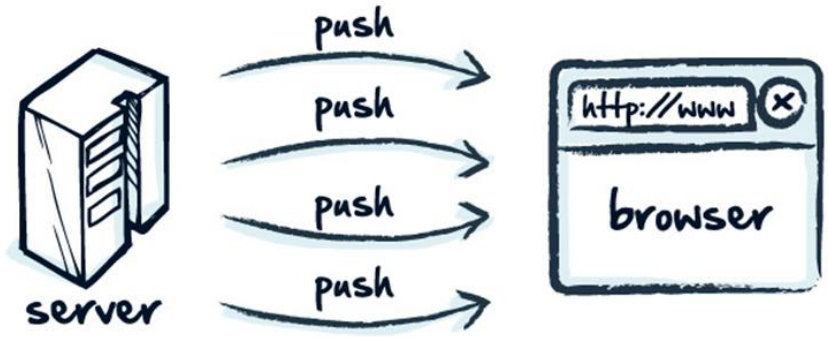


Figura 2 – Modelo push

Los servicios push siguen el modelo publicador-suscriptor, en el que cada cliente se subscribe a uno o varios canales de información, y ante un nuevo contenido o evento en el mismo, el servidor envía la información a los usuarios suscritos a dicho canal. Con este método se observa que el número de peticiones por parte del cliente es menor y, por tanto, mejora el rendimiento de la aplicación.

Volviendo al ejemplo del grupo de chat anterior, usando este nuevo modelo push, cada vez que un cliente envíe un mensaje, el servidor notificará a todos los usuarios conectados al chat con dicho mensaje, siendo innecesarias las peticiones por parte de los clientes. Así conseguimos una aplicación a tiempo real, eficiente, y ante todo interactiva.

Otros ejemplos actuales que usan este modelo son juegos interactivos en tiempo real, el correo electrónico [1], o la mensajería online [2], basada en notificaciones push, donde WhatsApp podría ser un ejemplo claro.

1.2 Objetivos

El objetivo del trabajo es realizar un análisis de las tecnologías push más actuales, enfocándolo principalmente en la especificación WebSockets. Además de una investigación del protocolo HTTP/2 para mejorar la interactividad de una aplicación web.

Para ello se realizará una pequeña aplicación web que usará este tipo de tecnologías. La implementación se realizará usando JSF (Java Server Faces), que facilita el diseño de la interfaz de usuario.

1.3 Estructura del trabajo

El trabajo está dividido en cuatro partes principales.

En la primera parte se detallarán las implementaciones push más extendidas actualmente, así como sus principales antecedentes, haciendo hincapié en el protocolo WebSocket.

Posteriormente, la segunda parte se centrará en un estudio de HTTP/2, además de realizar una comparativa con sus versiones anteriores.

La tercera parte se dedicará al análisis de la estructura de una aplicación que siga el modelo push en la actualidad; según las propuestas de la IETF y W3C, para la parte del servidor y cliente, respectivamente.

Por último, en la cuarta parte se creará una aplicación web usando JSF. Se hará uso de la especificación WebSockets.

1.4 Introducción a las tecnologías

Durante el proyecto se analizarán diversas tecnologías. Este apartado tiene como objetivo situar todas y cada una de ellas en el tiempo, así como una breve descripción de cada una de ellas.

En la siguiente lista se observa una breve decripción de cada una de ellas:

- **HTTP/1.0**: La primera versión de este protocolo. Muy simple, tan solo 66 páginas.
- **HTTP/1.1**: Segunda versión de este protocolo. Aumentaba su dificultad y añadía muchas funcionalidades y sutilezas.
- **Long Polling**: Una de las primeras técnicas que trataban de solventar la deficiencia del modelo petición-respuesta clásico de aplicación web.
- **Comet**: Modelo de aplicación web en el que una petición HTTP mantenida permite al servidor enviar datos al navegador, sin que sean explícitamente solicitados por el mismo.
- **Server-Sent-Events (SSE)**: Tecnología en la que el navegador recibe actualizaciones automáticas desde el servidor a través de una conexión HTTP.
- **WebSockets**: una de las apuestas más novedosas hoy día para implementar un modelo push en una aplicación web. Gracias a ellos se facilita el paso de mensajes entre el navegador y los servidores web.
- **HTTP/2**: Siguiente versión del protocolo HTTP, que incluye muchas novedades para mejorar el aprovechamiento de la red, simplificación de mensajes, etc.
- **WEBPUSH IETF**: Propuesta de la IETF para la implementación de un modelo push en una aplicación.
- **W3C PUSH API**: Propuesta de la W3C para la implementación del modelo push en una aplicación web.

En el siguiente diagrama se observan todas ellas cronológicamente:

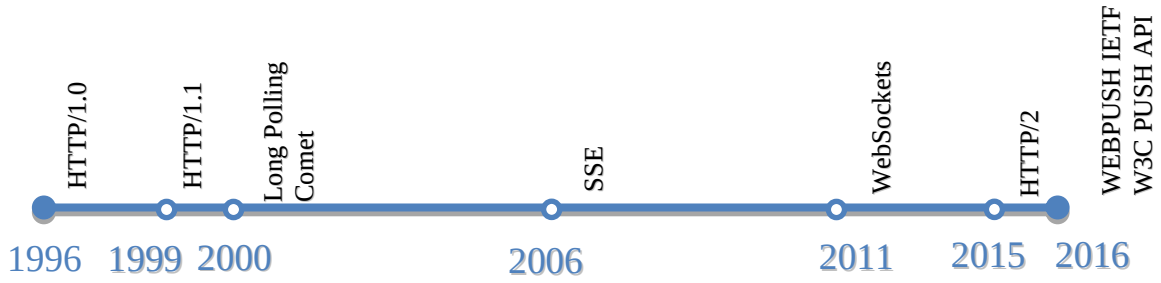


Figura 3 – Nacimiento de tecnologías

2 IMPLEMENTACIONES PUSH MÁS EXTENDIDAS ACTUALMENTE

No basta tener un buen ingenio, lo principal es aplicarlo bien.

René Descartes

En este capítulo se analizarán y compararán las diferentes implementaciones más populares en la actualidad para una aplicación que siga el modelo push.

En primer lugar, se hará un resumen del principal antecedente a esta tecnología, como es el Long Polling.

Posteriormente, se analizarán otros modelos muy populares como Server Sent Events y Comet para aplicaciones web.

Por úlimo se hará un resumen de la especificación WebSockets, como modelo basado para realizar la aplicación web en JSF del último capítulo.

2.1 Antecedentes, Long Polling

La técnica conocida como Long Polling, fue una de las pioneras en intentar solventar la deficiencia del modelo petición-respuesta clásico de aplicación web, donde el cliente siempre es el primero en iniciar la transacción de datos con su petición.

El funcionamiento básico de este método consiste en que el cliente realiza una única petición al servidor para iniciar la solicitud de información. El servidor mantiene esta conexión abierta mientras tenga datos para el envío y usará esta conexión para enviar datos al cliente. Una vez que el servidor finalice el envío de datos, se cierra la conexión. Inmediatamente el cliente inicia otra solicitud de conexión con el servidor, para así emular una aplicación a tiempo real. [3]

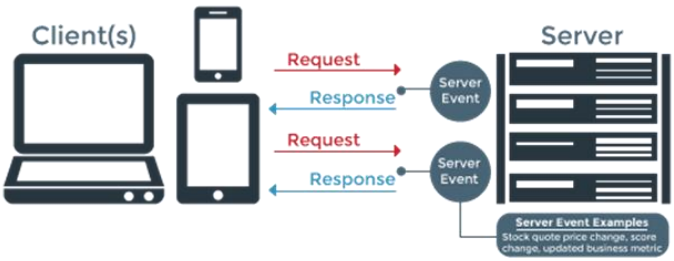


Figura 4 – Modelo basado en Long Polling

Además es importante destacar, como se observa en la imagen, que el servidor responde en función de eventos, notificando al cliente su cambio de estado.

Con este sistema se consigue un modelo de aplicación casi a tiempo real, sacrificando otros aspectos como el envío de muchas peticiones por parte del cliente, aumentando así la carga tanto en el cliente como en el servidor. Por consiguiente, no resulta ser una técnica eficiente. [4]

Actualmente, existen multitud de páginas que siguen utilizando este método, ya que suele tener una fácil implementación, aunque la tendencia se decanta por otras tecnologías más eficientes como WebSockets. [5]

2.2 Comet

Comet es un modelo de aplicación web en el que una petición HTTP mantenida permite al servidor enviar datos al navegador, sin que sean explícitamente solicitados por el mismo.

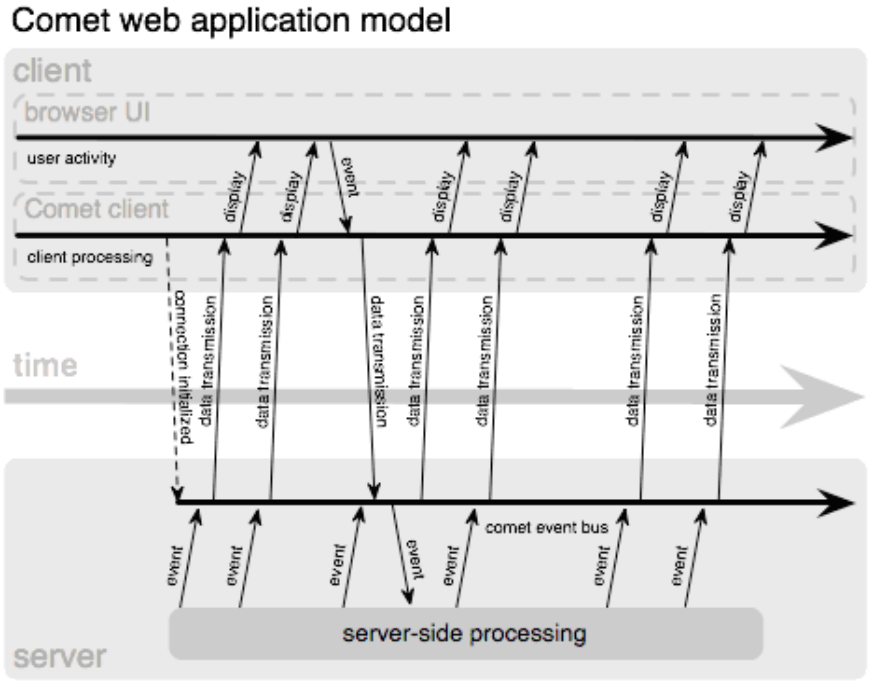


Figura 5 –Modelo basado en Comet

Este mismo término es conocido a través de otros nombres, entre los que destacan Ajax Push, Reverse Ajax, HTTP Streaming, HTTP Server Push o Two-way-web, entre otros.

El término Comet realmente se emplea para abarcar un conjunto de técnicas, que entre todas consiguen dicha funcionalidad. Todos estos métodos se basan en características incluidas por defecto en los diferentes navegadores, como el uso de JavaScript.

Este modelo difiere del clásico modelo de la web, en el que un navegador solicita una página web completa en un determinado momento; en lugar de esto, se basa en refrescar solo las partes o componentes que sean necesarios de la página, manteniendo los datos cacheados en memoria en sincronización con el servidor. [6]

Las primeras implementaciones basadas en Comet se remontan hacia el año 2000, con los proyectos Pushlets [7] y Lightstreamer [8].

Muchas de estas implementaciones se apoyan en Ajax (asynchronous JavaScript and XML). Actualmente existen dos variantes principales donde englobar el modelo Comet, que son el Streaming y el Ajax Long polling. [9]

2.2.1 Streaming Comet

Una aplicación que use Streaming Comet abre una única conexión persistente entre el navegador y el servidor para así intercambiar datos y los eventos Comet.

Estos eventos son manejados e interpretados en lado del cliente cada vez que el servidor envía un nuevo evento, sin que ninguno de los extremos cierre la conexión. [6]

Entre las principales técnicas para la implementación del Streaming Comet están las siguientes:

- **Hidden iframe**. Es una técnica básica para aplicaciones web dinámicas que permite usar elementos HTML ocultos, concretamente inline frames; es decir se permite embeber un documento HTML dentro de otro.

- **XMLHttpRequest (XHR)**. Es la principal herramienta de las aplicaciones Ajax para una comunicación navegador-servidor. Proporciona una forma fácil de obtener información de una URL sin tener que recargar la página completa. [10]

2.2.2 Ajax Long Polling

Debido a que ninguna de las técnicas de Streaming funciona en todos los navegadores actuales sin efectos negativos, se plantean modelos Comet que implementen la técnica Long Polling antes comentada, para así conseguir la funcionalidad en la mayoría de navegadores. Esta técnica es fácil de implementar en los navegadores que soportan XHR.

Como resumen de esta técnica, el cliente realiza una petición al servidor, que permanece abierta hasta que el servidor termine de enviar los datos correspondientes. Posteriormete el navegador manda otra petición para los posibles eventos en el servidor siguiente. (Mirar apartado 2.1).

Algunas de las técnicas que se emplean para la implementación de aplicaciones con Ajax Long Polling son las siguientes:

- **XMLHttpRequest long polling**. Funciona igual que XHR. El navegador inicia una petición asíncrona al servidor, manteniéndose mientras este tenga datos para el envío. Posteriormente, cuando termina de procesar la respuesta, el navegador inicia otra petición XHR, para recibir futuras respuestas de eventos.
- **<script> tag long polling**. Al igual que pasa con el método iframes anterior, el objetivo de esta técnica es incluir una etiqueta script en la página que permitirá la ejecución de un script. El servidor usará este script para enviar y recibir datos.

2.3 Server-Sent Events

Server-Sent Events (SSE) es una tecnología en la que el navegador recibe actualizaciones automáticas desde el servidor a través de una conexión HTTP. La API esta estandarizada por W3C, como parte de HTML5. [11]

El primer uso de esta tecnología se remonta al año 2006, en la que el navegador Opera la implementó de forma experimental, como una característica llamada Server-Sent Events. [12]

Este estándar describe cómo los servidores pueden enviar información al navegador una vez que la conexión se ha establecido. Se usa principalmente para el envío de mensajes o flujos de datos continuos al navegador. Uno de los principales objetivos es el streaming, que se consigue gracias a la API de JavaScript EvenSource, donde un cliente solicita una URL para recibir un flujo de eventos.

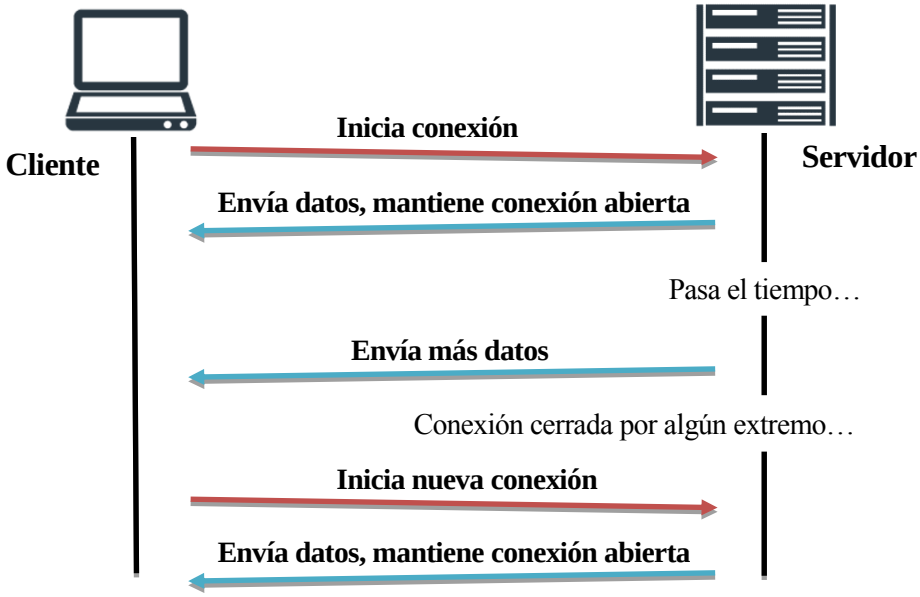


Figura 6 – Modelo basado en SSE [13]

Actualmente existen múltiples frameworks y librerías que permiten el desarrollo de aplicaciones basadas en SSE. Entre los más destacados están los siguientes:

Tabla 1 – Librerias para la implementación de SSE

Framework	Descripción	Sitio Web
jEaSSE	Framework para aplicaciones basadas en java	https://github.com/mariomac/jeasse
EvenSource	Framework para aplicaciones basadas en Swift	https://github.com/inaka/EventSource
TRVSEventSource	Framework para aplicaciones basadas en Objetive-C	https://github.com/travisjeffery/TRVSEventSource
Django-see	Framework para aplicaciones basadas en Python	https://github.com/niwinz/django-sse
Totaljs	Framework para aplicaciones basadas en NodeJS	https://github.com/totaljs
Hoa\Eventsource	Framework para aplicaciones basadas en PHP	https://github.com/hoaproject/Eventsource

2.3.1 Soporte en navegadores

En la siguiente imagen se muestra en verde las versiones que soportan SSE y en rojo los navegadores que no lo soportan. [14]

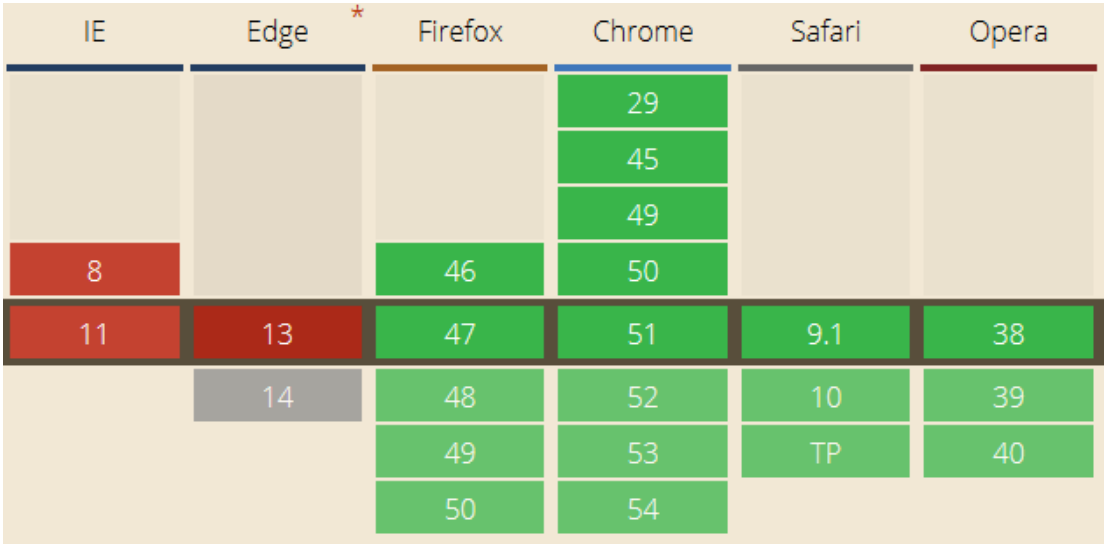


Figura 7 –Soporte de SSE en principales navegadores

*Microsoft Edge no lo soporta en la versión actual, en la siguiente se desconoce si se aceptará.

2.4 WebSockets

Los WebSockets, tal y como se ha mencionado antes, son una de las apuestas más novedosas hoy día para implementar un modelo push en una aplicación web. El IETF estandarizó este protocolo en el año 2011 en el RFC 6455 [15], mientras que W3C es el encargado de definir la API para el manejo en aplicaciones Web [16].

Gracias a ellos se facilita el paso de mensajes entre el navegador y los servidores web.

En los siguientes apartados se describirán con mayor detalle la base teórica, funcionamiento y formas de implementación de los WebSockets.

2.4.1 Visión General

La principal ventaja del protocolo WebSocket con respecto a sus alternativas es que proporciona un flujo de mensajes full-dúplex sobre una única conexión TCP, por lo que solo será necesario un socket TCP.

Esta tecnología está diseñada para ser implementada en navegadores y servidores web principalmente, aunque siempre se puede implementar en cualquier otra aplicación que siga el modelo cliente servidor.

Una de las principales características de esta tecnología es la mejora de la interactividad entre los navegadores y los sitios web, facilitando la transferencia de datos entre los mismos. Esto se consigue permitiendo el envío de datos desde el servidor al navegador sin que sean solicitados previamente, además del paso bidireccional de mensajes sobre la conexión TCP abierta.

Las comunicaciones de este protocolo se realizan sobre el puerto 80/TCP. Esto es una gran ventaja ya que la mayoría de aplicaciones bloquean todas las conexiones no-web (puertos diferentes del 80) con un Firewall, por motivos de seguridad.

La especificación del protocolo WebSocket define dos nuevos esquemas URI (Uniform Resource Identifier) para las conexiones; estos son ws y wss, para las conexiones sin cifrar y cifradas respectivamente.

Por último, es importante subrayar que el protocolo WebSocket está actualmente soportado por la mayoría de navegadores, incluyendo Google Chrome, Firefox, Safari, Opera, Edge e Internet Explorer. Es destacable también que es necesario el soporte de WebSockets en la parte del servidor de la aplicación Web.

2.4.2 Breve historia

Cabe destacar que este protocolo se inició con el nombre de TCPConnection en la especificación de HTML5, como una referencia a la API de conexión a través de socket TCP. Sin embargo, en 2008, se decidió cambiarle el nombre y así sacar la primera versión de este protocolo, conocida como WebSocket.

En diciembre de 2009, Google Chrome fue el primer navegador en incorporar soporte a este estándar, manteniendo el protocolo WebSocket activado por defecto. Posteriormente, en febrero de 2010, el desarrollo del protocolo pasó a manos de la IETF.

Y ya en diciembre de 2011 se terminó la RFC 6455 -The WebSocket Protocol- pasando a estar soportado y activo por la mayoría de navegadores actuales.

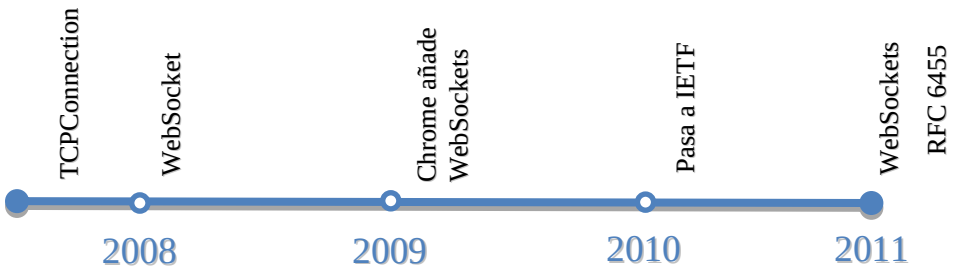


Figura 8 – Evolución WebSockets

2.4.3 Implementación en los navegadores

Actualmente el protocolo WebSocket está implementado en los principales navegadores.

En las siguientes tablas se muestra la evolución de las primeras versiones de los navegadores que empezaron a adoptarlo:

Tabla 2 – Primeras versiones de navegadores (PC) que soportan WebSockets

Protocolo	Fecha	Internet Explorer	Firefox	Chrome	Safari	Opera	Edge
hixie-75	4/2/2010			4	5.0.0		
hixie-76 **hybi-00**	6/5/2010 23/5/2010		4.0 (deshabilitado)	6	5.0.1	11.00 (deshabilitado)	
hybi-07	22/4/2011		6				
hybi-10	11/7/2011		7	14			
RFC 6455	Diciembre 2011	10	11	16	6	12.10	13

También se muestra la evolución de las versiones de los navegadores de dispositivos móviles que empezaron a usar el protocolo:

Tabla 3 – Primeras versiones de navegadores (móvil) que soportan WebSockets

Protocolo	Fecha	Firefox Android	Chrome Móvil	Safari iOS	Opera Móvil	Navegador Android
hixie-75	4/2/2010		4	5.0.0		
hixie-76 **hybi-00**	6/5/2010 23/5/2010		6	5.0.1	11.00 (deshabilitado)	
hybi-10	11/7/2011	7	14			
RFC 6455	Diciembre 2011	11	16	6	12.10	4.4

2.4.4 Negociación del protocolo

La negociación o Handshake del protocolo tiene como objetivo el establecimiento de una conexión WebSocket. Para ello el cliente envía una solicitud tipo Upgrade para la negociación, a la que el servidor le envía una respuesta.

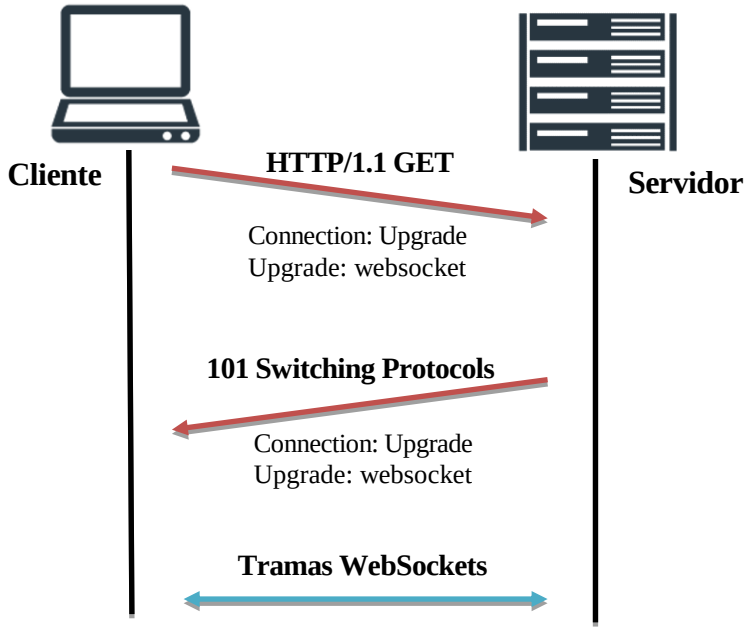


Figura 9 – Negociación WebSockets

Un ejemplo de petición sería el siguiente (Usando Firefox Development Tools):

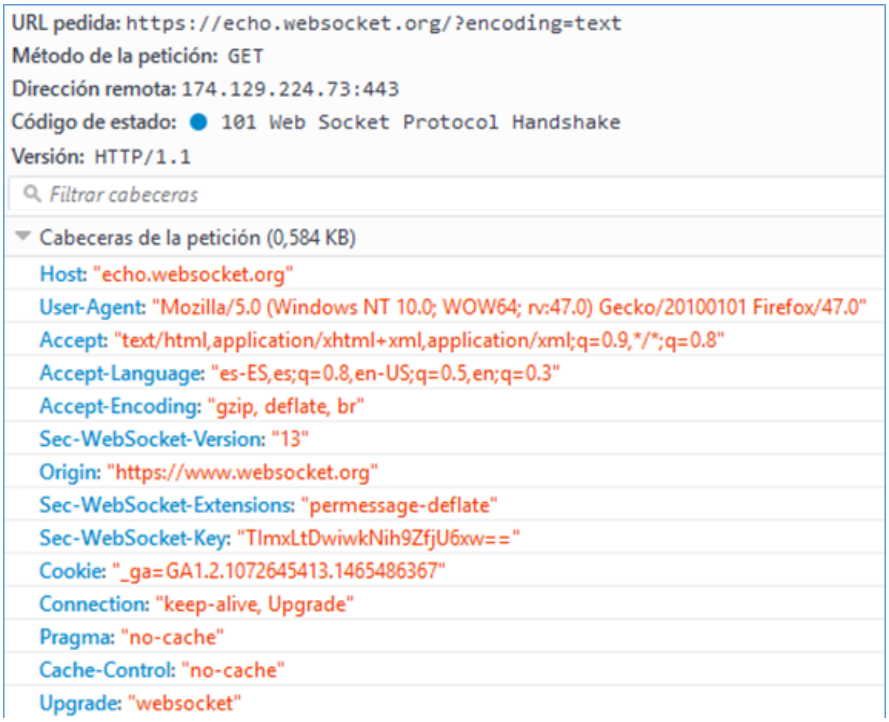


Figura 10 – Cabecera de petición WebSockets

Mientras que la respuesta del servidor es la siguiente:

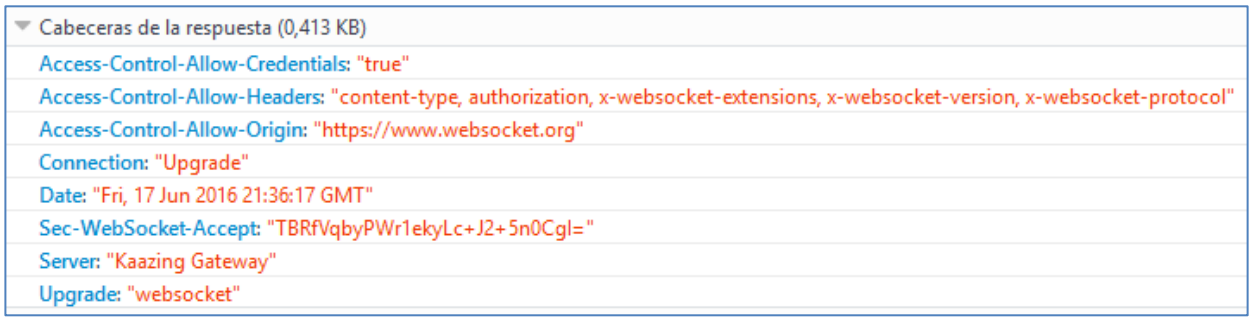


Figura 11 – Cabecera de respuesta WebSockets

Destacamos que esta negociación usa el protocolo HTTP, lo que permite a un servidor manejar todas las conexiones HTTP y de WebSockets en el mismo puerto. Una vez establecida la conexión, se usa otro protocolo diferente a HTTP para la comunicación, basado en la comunicación bidireccional full-dúplex.

Como vemos en las cabeceras de negociación, el tipo que se usa es **Connection:** Upgrade y **Upgrade:** websocket; también en la petición tenemos que indicar el host al que va dirigida la petición, con **Host:** servidor.ejemplo.com. Otro parámetro necesario en navegadores es el origen, para asegurar una conexión deseada y segura, **Origin:** https://www.websocket.org.

Otro aspecto fundamental es el parámetro **Sec-WebSocket-Key:** TlmxLttDwiwkNih9ZfjU6xw==, al que el servidor responde con **Sec-WebSocket-Accept:** TBfVqbyPWr1ekyLc+J2+5n0Cgl= para garantizar la seguridad en los envíos y evitar el cacheo proxy de conexiones anteriores. Además de la versión del protocolo usada, **Sec-WebSocket-Version:** 13.

Una vez establecida la conexión, las tramas WebSocket de datos o de texto pueden enviarse en ambos sentidos entre el cliente y el servidor en modo full-dúplex. Los datos se encapsulan con una pequeña cabecera y la carga útil (payload) del mensaje.

Las transmisiones WebSockets, conocidas como "mensajes", pueden dividirse en varias tramas de datos. Con esto se permite enviar mensajes cuando hay disponibles datos iniciales, pero la longitud total del mensaje es desconocida. Con extensiones del protocolo, esto puede servir para multiplexar varios flujos de mensajes simultáneamente (por ejemplo para evitar monopolizar el socket con una única carga muy grande).

2.4.5 Intercambio de datos

Una vez se ha establecido la conexión, los datos se transmiten en ambas direcciones entre cliente y servidor, consiguiendo así un modelo a tiempo real.

Gracias a herramientas como Chrome Developer Tools o Firefox Developer Tools con el complemento WebSocket Monitor, se puede observar en tiempo real el paso de estos mensajes, así como la apertura o cierre de conexiones WebSockets.

En el siguiente ejemplo se ven los mensajes intercambiados con un servidor de eco, que devuelve el mismo texto enviado por el cliente.

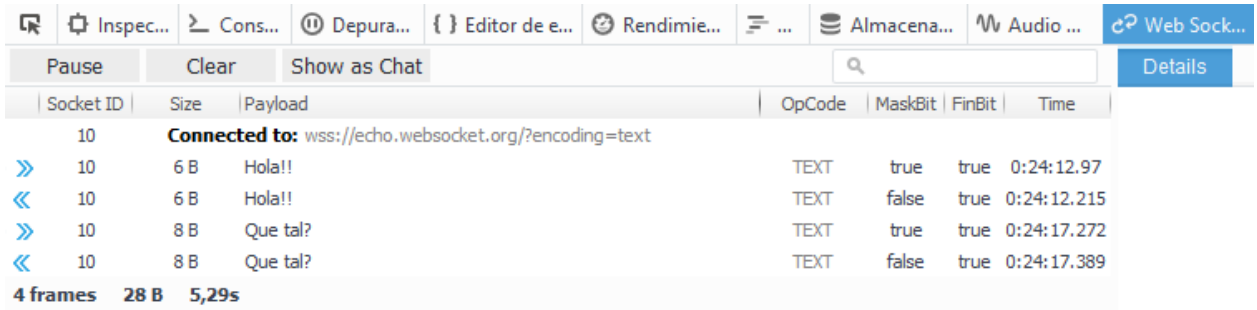


Figura 12 – Mensajes intercambiados WebSockets

También existen algunas otras opciones como ver las cabeceras de cada mensaje, el Payload, y diferentes vistas tipo chat, etc.

Figura 13 – Detalles de un mensaje WebSocket

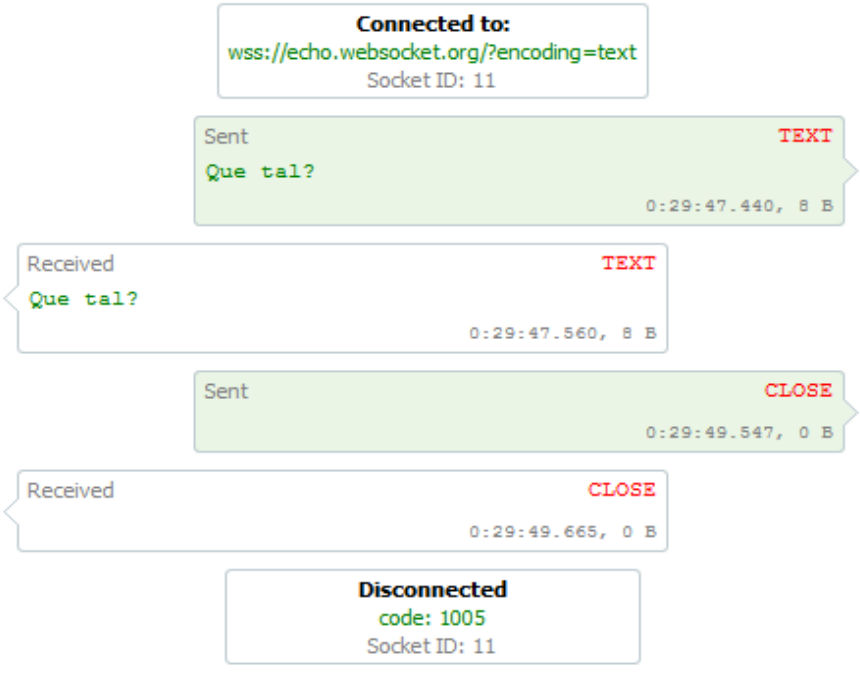


Figura 14 – Vista en modo Chat WebSockets

2.5 Tabla comparativa

En la siguiente tabla se muestran las principales características de los diferentes modelos mostrados anteriormente. [5]

Tabla 4 – Comparativa tecnologías

	Long-polling/Comet	**Server-Sent Events**	**WebSockets**
Soporte en el navegador	Soportado en la mayoría de navegadores	Soportado en la mayoría, excepto IE y Edge	Soportado en la mayoría de navegadores
Carga en el servidor	Crea un proceso idle por usuario, ocupando CPU y memoria	Parecido a Long polling, a diferencia que no necesita cerrar la conexión tras el envío de la respuesta	Mejor solución posible, óptima utilización
Carga en el cliente	Depende de la implementación, aunque requiere carga en el cliente ya que necesita enviar múltiples peticiones	Mínima, implementado en el navegador por defecto	Mínima, implementado en el navegador por defecto
Tiempo real	Casi tiempo real, retardos entre petición y respuesta	Retardo por defecto de 3s, configurable	Sí
Complejidad en la implementación	Fácil en general	Fácil, con la ayuda de algunos Frameworks	Fácil, gracias al uso de Frameworks como Spring o Atmosphere

3 HTTP/2

Grandes descubrimientos y mejoras implican
invariablemente la cooperación de muchas mentes

Alexander Graham Bell

Este apartado se centrará en describir brevemente la evolución histórica de este protocolo, las características más importantes del protocolo HTTP/2, las ventajas que supone su utilización, además de una comparativa con su versión anterior, HTTP/1.1.

3.1 HTTP hoy

Actualmente HTTP es un protocolo usado por todo el mundo en Internet. Se han realizado grandes inversiones en protocolos e infraestructuras para sacarle el máximo partido. Así, a la hora de crear alguna aplicación, es más fácil hacerla funcionar sobre HTTP que construir alguna tecnología nueva.

En sus inicios, HTTP fue concebido como un protocolo simple y sencillo. Su primera versión, HTTP/1.0 en el RFC 1945, surgió en el año 1996 en una especificación de 60 páginas. Sin embargo, tres años más tarde, se publicó HTTP/1.1 en el RFC 2616, creciendo en extensión hasta las 176 páginas, aumentando considerablemente su dificultad, además de que incluía una gran variedad de detalles y sutilezas.

3.2 Problemas de HTTP/1.1

3.2.1 Uso inadecuado de TCP

Uno de los principales problemas de HTTP/1.1 es que nunca ha conseguido aprovechar la ventaja y rendimiento que ofrece TCP. Así, para reducir los tiempos de recarga de las páginas, los clientes HTTP y los navegadores tienen que encontrar soluciones muy creativas.

Entre estas razones destacamos que la cantidad de información que debe ser consumida ha crecido considerablemente en los últimos 5 años. En el siguiente gráfico, se observa un gran incremento en el tamaño de transferencia (verde) y el número de total de peticiones (rojo) para servir los sitios web más populares del mundo:

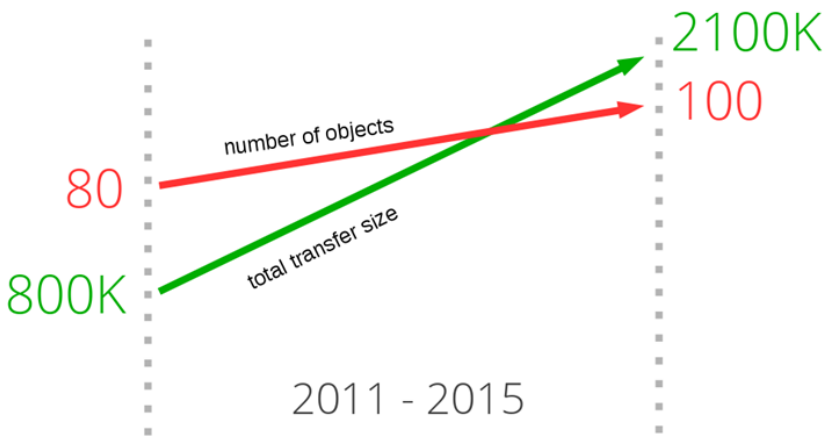


Figura 15 – Evolución del tamaño de transferencia y número de peticiones [17]

3.2.2 Mala latencia

Otro de los principales problemas que plantea HTTP/1.1 es que es muy sensible a la latencia. Considerando que el ancho de banda ha aumentado notoriamente en los últimos años, no se ha conseguido la misma mejora reduciendo la latencia.

En el siguiente gráfico vemos un ejemplo práctico de esta cuestión; se observa que cuanto menor es el RTT (tiempo de ida y vuelta de un paquete de datos), menor es el tiempo que tarda en cargar la página:

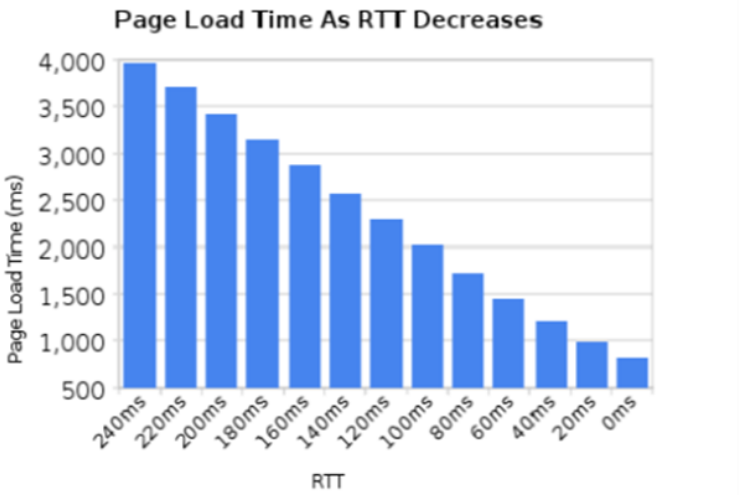


Figura 16 –Tiempo de carga de una página en función de RTT [17]

En enlaces de alta latencia, como es el caso de las tecnologías móviles actuales, es muy complicado conseguir una buena y sobre todo rápida experiencia de usuario en la web, incluso contando con un gran ancho de banda. Por otro lado, en enlaces de baja latencia, como los que necesitan los juegos o algunos tipos de vídeo, no se envía únicamente un único flujo de datos, por lo que este aspecto es fundamental.

3.2.3 Bloqueo del primero de la fila

Por último, el otro gran problema que se le plantea a HTTP/1.1 es el llamado "Bloqueo del primero de la fila" (head of line blocking), cuando se usa HTTP Pipelining. Este método consiste en enviar varias solicitudes consecutivas, sin haber recibido sus respuestas.

En la siguiente imagen se observa una comparativa de este aspecto:

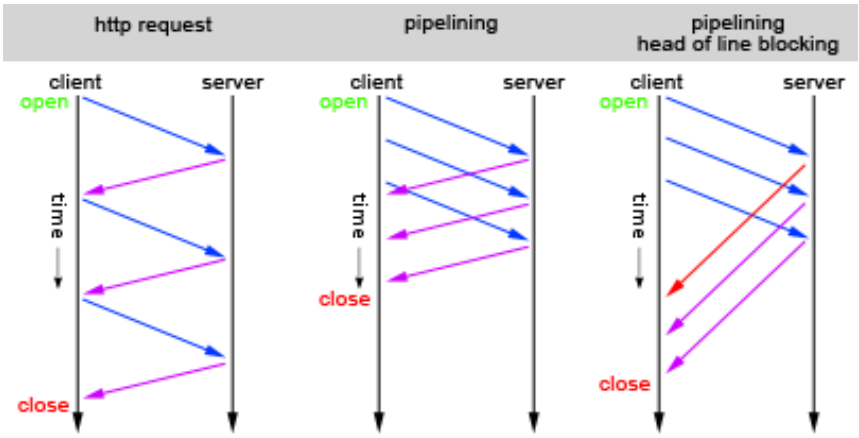


Figura 17 – Bloqueo del primero de la fila [18]

Por tanto se puede concluir que con la utilización de HTTP Pipelining, si existe un usuario que tenga malos requisitos de red o que reenvíe múltiples paquetes, producirá un bloqueo en la conexión, perjudicando la experiencia del resto de usuarios en la web. [19]

3.3 Primeras soluciones

Ante los problemas descritos anteriormente, se describen varias técnicas para solventarlos. Algunas de ellas son inteligentes y muy útiles, mientras que otras son muy engorrosas a la vez que ineficientes.

3.3.1 Spriting

Esta técnica se utiliza en imágenes y consiste en la unión de varias imágenes pequeñas en una única imagen más grande. Posteriormente con CSS o JavaScript se van recortando para así obtener las deseadas.

Con esta medida se consigue mayor velocidad, ya que una única descarga es mucho más rápida que pequeñas descargas individuales.

3.3.2 Inlining

Este es otro truco para evitar enviar imágenes individuales; se consigue usando diferentes URL "data" desde un fichero CSS:

```
.icon1 {
 background: url(<data>) no-repeat;
}
```

3.3.3 Concatenación

Con esta técnica se consigue agrupar ficheros JavaScript en un único archivo. Se obtiene gracias a algunas herramientas de front-end, y se obtiene como resultado una única descarga de fichero en lugar de varias. Sin embargo, esta técnica, normalmente, resulta engorrosa para los desarrolladores.

3.3.4 Sharding

También llamado fragmentación. Consiste en servir los distintos elementos de tu servicio desde el mayor número de servidores posible.

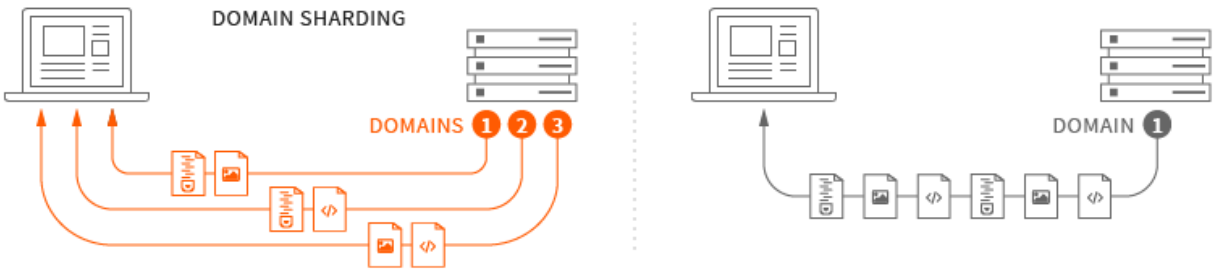


Figura 18 – Sharding como alternativa [20]

Inicialmente, en la especificación de HTTP/1.1 se fijaba un número máximo de dos conexiones TCP por dominio. Para no saltarse la especificación, ciertos sitios se inventaron nuevos nombres de dominio para así conseguir tener más conexiones y lograr bajar el tiempo de carga.

Hoy día se ha eliminado dicha restricción y los clientes actuales usan fácilmente unas 6-8 conexiones por dominio. Con esta limitación, algunos sitios siguen usando esta técnica para aumentar el número de conexiones, ya que la utilización de un elevado número de conexiones asegura un buen rendimiento de HTTP.

3.5 Actualizando HTTP

Se observa que la versión HTTP/1.1 es mejorable, sobre todo para:

- Reducir los tiempos de RTT.
- Arreglar el Pipelining y el problema de bloqueo de primero de fila.
- Parar la necesidad y deseo de seguir aumentando el número de conexiones para cada host.
- Mantener todas las interfaces actuales existentes. Compatibilidad.

El grupo HTTPbis se encargó de actualizar la especificación HTTP/1.1. Este trabajo fue terminado a comienzos de 2014 y resultó en la serie RFC 7320. Todo esto llevaría a la versión HTTP/2.

3.5.1 HTTP/2 empieza con SPDY

SPDY es un protocolo encabezado y desarrollado por Google. Ha sido desarrollado en abierto y todo el mundo ha sido invitado a participar, pero obviamente Google es el mayor beneficiado al controlar una implementación muy popular de navegador y una significativa parte de la población de servidores con bastantes servicios muy conocidos.

SPDY surge con el objetivo de reducir el tiempo de carga de páginas web aprovechando TCP; utiliza solo una única conexión para manejar varias peticiones HTTP a la vez, de manera concurrente. Además, usa la capa SSL para incrementar la seguridad del mismo.

Cuando el grupo HTTPbis comenzó a trabajar en HTTP/2, SPDY había demostrado ser un concepto que funcionaba. Por eso, la primera versión de HTTP/2 surgió de SPDY/3 con un poco de búsqueda y reemplazo.

Figura 19 – Evolución HTTP [21]

3.6 Conceptos de HTTP/2

Entre las principales características y restricciones que se impusieron a la hora de crear HTTP/2 destacan las siguientes:

- Mantener los paradigmas HTTP. Se mantiene como un protocolo que envía peticiones al servidor a través de TCP.
- Las URLs http:// y https:// no cambiarán. No puede existir un nuevo esquema, ya que existe una gran cantidad de contenido usando estas URLs.
- Servidores y Clientes HTTP/1.X se mantendrán durante décadas, pudiendo realizar un proxy hacia servidores HTTP/2.
- Por tanto, los proxies podrán mapear funcionalidades HTTP/2 a clientes HTTP/1.1 una a una.
- Eliminar o reducir partes opcionales del protocolo. Esto no es en sí un requisito, sino una forma de trabajar que llegaba desde SPDY y el equipo de Google. Asegurándose de que todo es obligatorio se evita una implementación inadecuada que más adelante pueda convertirse en una trampa.
- No más versiones menores. Se decidió que tanto clientes como servidores serían o no compatibles con HTTP/2. Si aparece una necesidad de extender el protocolo o modificar las cosas, entonces habrá nacido HTTP/3. No hay más versiones menores en HTTP/2.

3.6.1 HTTP/2 para esquemas URI existentes

Como se ha mencionado con anterioridad, los esquemas URI existentes no podrán ser modificados. De esta manera, hay que conseguir una forma nueva para solicitar al servidor que vamos a usar un protocolo determinado, en nuestro caso HTTP/2.

HTTP/1.1 conseguía este objetivo con el campo "Upgrade" de su cabecera, utilizando una petición del cliente al servidor. Sin embargo, esta forma no era aceptada por el equipo SDPY, ya que era necesaria una petición (round-trip) y que, además, no era muy eficiente. Por esto, se desarrolla una nueva extensión TLS para abordar la negociación de manera significativa. Conocida como NPM (Next Protocol Negotiation), que consiste en que el servidor comunica al cliente que protocolos conoce, así el cliente puede proceder y utilizar su protocolo preferido.

3.6.2 HTTP/2 para HTTPS

Debido a que SPDY solo funciona sobre TLS, se ha intentado con mucho esfuerzo que TLS sea obligatorio para HTTP/2, pero sin llegar a un consenso, por lo que se ha publicado con TLS opcional. Aun así, dos de los fabricantes más destacados han dicho que solo implementarán HTTP/2 sobre TLS: la iniciativa de Mozilla Firefox y Google Chrome, dos de los navegadores principales hoy día.

3.6.3 Negociación HTTP/2 sobre TLS

Tal y como hemos comentado antes, SPDY usa NPN para negociar con los servidores TLS. Sin embargo, para HTTP/2 no era un estándar adecuado, por lo que fue llevado a la IETF y surgió ALPN (Aplication Layer Protocol Negotiation).

La principal diferencia de ALPN con respecto a NPN es quién decide. Con ALPN el cliente le indica al servidor el listado de preferencia y el servidor escoge el que quiere. Mientras que con NPN es el cliente quien toma la decisión.

3.6.4 HTTP/2 sobre HTTP

El principal método de usar HTTP/2 sobre HTTP es que el cliente solicite al servidor el protocolo con la cabecera "Upgrade". Sin embargo, este procedimiento de actualización usa peticiones "round-trip", por lo que no es eficiente.

Aunque algunos representantes de navegadores han declarado que no implementarán este modo de comunicarse, Internet Explorer o Curl si lo harán.

3.7 El protocolo HTTP/2

Una vez comentado brevemente los antecedentes, historia y conceptos; es preciso indagar en los asuntos más específicos del protocolo. [17]

3.7.1 Binario

Es una de las características más novedosas e impactantes de este protocolo. Cabe la opción preguntarse, ¿por qué no usar un protocolo en formato texto/ASCII en lugar de binario? Así podríamos hacer consultas a mano con telnet, que un humano interactúe directamente…

La respuesta a esta pregunta es bien sencilla, HTTP/2 usa el formato binario para hacer más sencillo el entramado (framing), ya que uno de los aspectos más complicados en HTTP/1.1 es detectar el principio y fin de la trama, eliminando espacios en blanco, diferentes formas de escribir la misma cosa, etc. Así se consigue hacer mucho más fácil la separación de las partes de las tramas.

Otro de los motivos por lo que no preocupa mucho el hecho de que esté en binario es que va sobre la capa TLS, por lo que los datos no viajan de cualquier manera, sino cifrados. Por eso será necesario el uso de algún inspector de tráfico tipo Wireshark para la depuración y determinación de lo que está ocurriendo a nivel HTTP/2.

Por tanto, las tramas de HTTP/2 están en binario. Pueden enviarse 10 tipos distintos de trama, todas comenzando de la misma forma:

- Tamaño.
- Tipo.
- Flags.
- Identificador de flujo.
- Carga útil.

Existen 10 tipos de trama definidos en la especificación HTTP/2. En la siguiente imagen se muestra con mayor detalle el formato de trama y los tipos existentes:

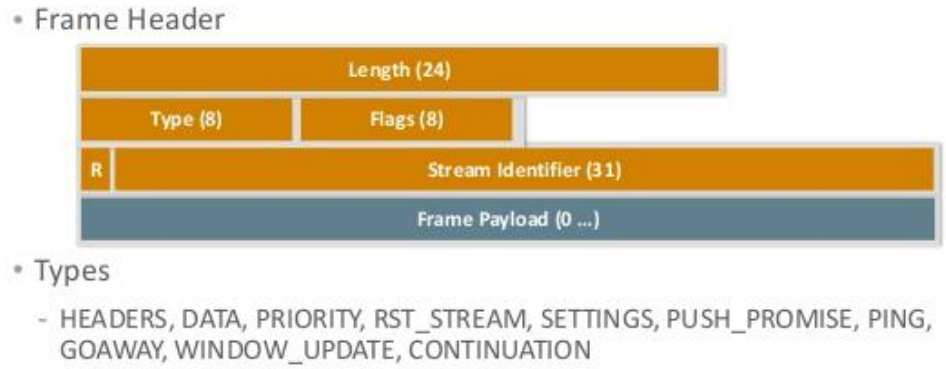


Figura 20 – Tipos y formato de trama

3.7.2 Flujos multiplexados

Como hemos comentado antes, cada trama binaria está asociada a un flujo gracias a su identificador de flujo. Un flujo sería algo así como una asociación lógica, es decir una secuencia independiente de tramas, bidireccional entre el cliente y servidor dentro de una conexión HTTP/2.

Dentro de una conexión HTTP/2 pueden existir, por tanto, varios flujos abiertos concurrentes. Siendo establecidos y usados unilateralmente por el cliente o el servidor, asimismo pueden ser cerrados por cualquiera de los dos puntos.

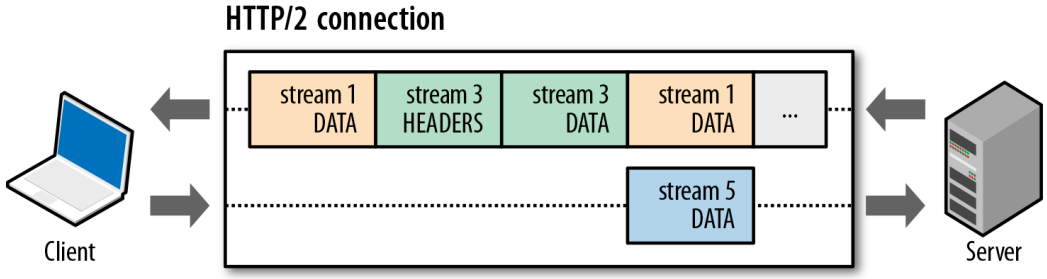


Figura 21 – Flujos multiplexados HTTP/2

Esta característica de multiplexación significa que los paquetes de distintos flujos pueden mezclarse en la misma connexion para, posteriormente, separarse en el destino. Cabe destacar que los receptores procesan las tramas en el mismo orden que le llegan, por lo que el orden de los flujos es algo significativo.

3.7.3 Prioridades

Cada flujo tiene asociado una prioridad, usada para indicar qué flujo es más importante en caso de contar con restricciones de recursos.

Usando la trama PRIORITY se pueden hacer que unos flujos dependan de otros, y así tendrán una jerarquía padre-hijo.

Destaca también que estas prioridades pueden ser cambiadas dinámicamente en tiempo real. Un ejemplo sería una página cargada de imágenes, en las que se cambie de prioridad de solicitud de las mismas a medida que el usuario hace scroll sobre la página.

3.7.4 Compresión de cabeceras

HTTP es un protocolo sin estado, por lo que cada petición debe indicar al servidor los detalles necesarios para que sea atendida. Así evitamos que el servidor tenga que almacenar gran cantidad de información de peticiones anteriores. HTTP/2 sigue este mismo modelo. Sin embargo, esto convierte a HTTP en repetitivo. Cuando un cliente solicita muchos recursos de un mismo servidor, como imágenes, habrá una serie de peticiones que serán idénticas. Este es el motivo por lo que HTTP/2 incluye compresión de cabeceras.

Otro argumento a favor de la compresión es el tamaño de las peticiones. En HTTP/1.1, las peticiones han crecido hasta tal punto de superar la ventana TCP inicial, lo que hace que el envío sea muy lento al necesitar recibir un ACK de vuelta.

Sin embargo, resulta que las compresiones HTTPS y SPDY han resultado ser vulnerables a ataques BREACH y CRIME; que consisten básicamente en que el atacante inserta texto conocido en el flujo, pudiendo, así, ver lo que cambia en el resultado comprimido, y saber lo que se está enviando.

HTTPbis intenta crear un protocolo que a pesar de hacer compresión de cabeceras no sea vulnerable a estos ataques, por lo que se introduce HPACK, Header Compresion for HTTP/2, que se especifica en otro estándar por separado (RFC 7541). Este nuevo formato, junto a otras técnicas como un bit que indica que no se comprima alguna cabecera específica o desplazamiento de tramas, hace mucho más complicado romper esta compresión.

3.7.5 Reset

Otra de las novedades de HTTP/2 es la capacidad de parar un envío de un mensaje. En HTTP sería necesario cerrar la conexión TCP y volverla a abrir, lo que resultaría no eficiente.

Esta acción se realiza con la trama RST_STREAM; primero se para el mensaje y se comienza uno nuevo, así de simple, y todo manteniendo la misma conexión TCP.

3.7.6 Server push

Esta es una de las funcionalidades más novedosas, ya que el servidor puede predecir lo que el cliente le va a solicitar en un futuro y así mandárselo sin una petición previa.

También es conocido como “cache push”. Su funcionamiento es el siguiente: el cliente solicita un recurso X determinado, el servidor determina que es muy probable que también solicite el recurso Y, por lo que se lo manda para que lo tenga en caché por si lo necesita.

3.7.7 Control de flujo

Cada flujo individual sobre HTTP/2 tiene su propia ventana de flujo asociada, sobre la que el otro extremo puede enviar información. Así se consigue que cada extremo envíe solo la información que quepa en la ventana del flujo. Cabe destacar que este control de flujo solo está disponible para las tramas DATA.

3.8 HTTP/2 Actualmente

Una vez descrito HTTP/2, nos preguntamos, ¿Cómo está a día de hoy su implantación? ¿será adoptado en un futuro con éxito?

A día de hoy, HTTP/2 no está ampliamente desplegado ni usado. Es complicado predecir cómo serán las cosas dentro de unos años, pero tras las mejoras que supone con respecto a su anterior versión podemos hacer una estimación.

3.8.1 HTTP/2 en el desarrollo web

Durante años, como se ha comentado anteriormente, los desarrolladores web y sus entornos de desarrollo iban solventando los problemas de HTTP/1.1 realizando pequeños trucos y herramientas, tales como el "spriting", el "inlining" o el "sharding".

Sin embargo, el uso de estos trucos perjudica a HTTP/2; de manera que mientras existan clientes de los dos tipos, es complicado alcanzar el máximo potencial de HTTP/2 en el desarrollo web. Por lo que pasará un tiempo antes de que se adopte unívocamente.

3.8.2 Implementaciones HTTP/2

Desde el inicio, han existido numerosas implementaciones de servicios sobre HTTP/2. Encabezados por Firefox, Twitter, Google, etc, se comienzan a ofrecer estos servicios.

Apache [22] y Nginx [23], entre los servidores más populares, también permiten el soporte de este protocolo en sus últimas releases (2.4.17 para Apache y 1.9.5 para Nginx). –Ver detalle de implementación en Anexo A-.

3.9 Soporte de HTTP/2 para WebSockets

Tras una investigación de este protocolo, se ha llegado a la conclusión de que HTTP/2 no soporta WebSockets. Hace dos años, la IETF sacó un borrador para implementar WebSockets sobre est protocolo [24], sin embargo se quedó en un intento, ya que no se ha sacado una versión oficial y no tiene pinta de que se saque por el momento. En su defecto, WebSockets funciona sobre HTTP/1.1 y lo seguirá haciendo por el momento.

Por tanto, esto significa que la nueva especificación HTTP/2 no va a dejar obsoleta a los WebSockets, ya que no incluye ningún soporte oficial para su implementación.

3.10 Comparativa HTTP/1.1 y HTTP/2

3.10.1 Introducción a la prueba

En este apartado se va a realizar una comparativa a nivel práctico de HTTP/1.1 con HTTP/2. Para ello se ejecutará una pequeña aplicación web, donde se realizará la petición de un conjunto de imágenes usando ambas versiones.

El conjunto de estas imágenes pequeñas formará una imagen mayor, como se aprecia en la siguiente fotografía:

Figura 22 –Imagen formada por múltiples imágenes

Como resultado final se comparará el tiempo de carga de la imagen total.

3.10.2 Implementación en Apache

Para realizar dicha prueba se montará un servidor apache con el módulo http2 incorporado (en Anexo A se encuentra información de instalación y configuración).

Posteriormente, se cargará la página activando y desactivando este módulo, para así comparar los resultados obtenidos.

Cabe destacar que el módulo http2 se ha montado para que funcione sobre HTTPS única y exclusivamente. Por consiguiente, si se quiere usar bastará con solicitar la página con HTTPS. En caso contrario, usaremos simplemente HTTP para obtener los resultados de carga usando HTTP/1.1.

3.10.3 Resultados

Para obtener los resultados de tiempo de carga se usará la siguiente función JavaScript, que utilizará la interfaz performanceTiming para obtener los tiempos de:

- Apertura de conexión: window.performance.timing.connectStart
- Carga completa de página: window.performance.timing.domComplete

```
  function showtimes() {
        var tiempo = 'Tiempo de carga de la página:<br>'

        tiempo = 'Carga completa: ' + (window.performance.timing.domComplete -
window.performance.timing.connectStart) + 'ms<br>'

        document.getElementById('tiempocarga').innerHTML = tiempo
  }
```

Una vez completada la carga de la página con los diversos métodos obtenemos los siguientes resultados:

- Tiempo de carga total con HTTP/1.1: 3.566 segundos
- Tiempo de carga total con HTTP/2: 0.77 segundos

Por tanto, se ha conseguido una mejora de casi 5 veces el tiempo de carga total.

Figura 23 – Carga de imagen con HTTP/1.1

Figura 24 –Carga de imagen con HTTP/2

Este experimento está basado en el realizado en [25]. Existen otros experimentos también muy interesantes como el realizado en [26].

4 ESTRUCTURA BASADA EN MODELO PUSH

Para el optimista, el vaso está medio lleno.
Para el pesimista, el vaso está medio vacío.
Para el ingeniero, el vaso es el doble de grande de lo que debería ser.

Anónimo

En esta sección se analizarán las diferentes propuestas de la IETF y W3C para la estructura de una aplicación que siga el modelo push en la actualidad.

Consta de dos partes principales, la primera se centra en el análisis de la comunicación en el lado del servidor mientras que la segunda se basa en la parte del cliente, que será el navegador web, como se observa en la siguiente imagen:

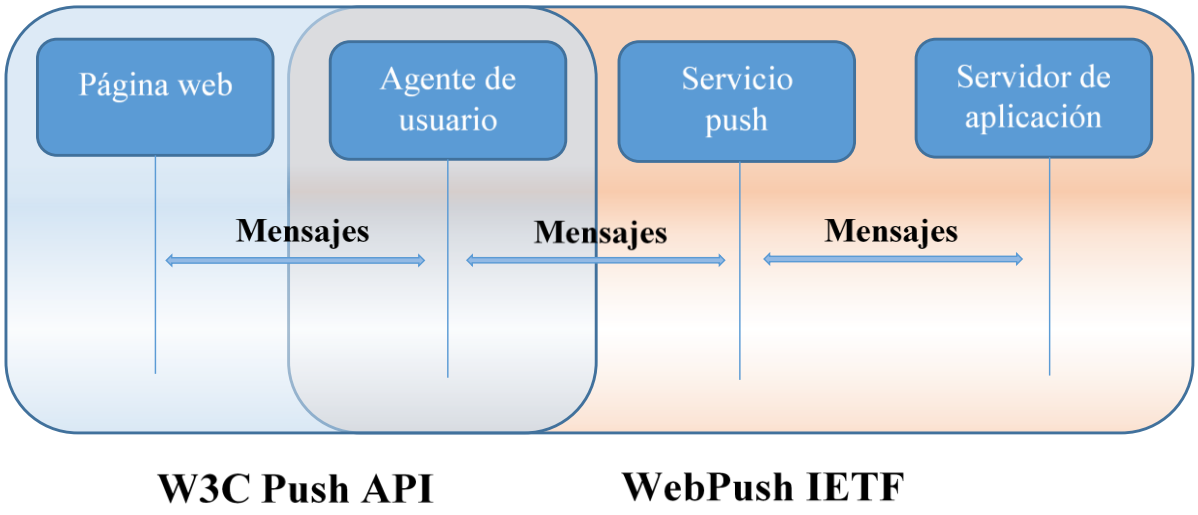


Figura 25 – Estructura de una aplicación que sigue el modelo push

4.1 Propuesta WebPush de la IETF

4.1.1 Introducción

La IETF (Internet Engineering Task Force) define un protocolo para abordar este modelo de eventos en tiempo real. Este protocolo usará HTTP/2 para procesar estos eventos (mensajes, llamadas…) y así hacer push hacia los clientes de forma inmediata. [27]

Cabe destacar que la mayoría de estos intercambios de mensajes pueden realizarse con HTTP/1.1, por ello en los ejemplos explicativos se usará este formato. Sin embargo, HTTP/2 es necesario para algunos mensajes, por lo que cuando se use en ejemplos, se indicará previamente.

Este protocolo asegura un uso más eficiente de la red y de las radios (WiFi), ya que la comunicación por radio consume una gran parte de la energía en un dispositivo inalámbrico.

Tal y como es citado en el estándar:

> *"Uncoordinated use of persistent connections or sessions from multiple applications can contribute to unnecessary use of the device radio, since each independent session independently incurs overheads."*

Todas y cada una de las sesiones iniciadas por los clientes influyen en el rendimiento y uso de la radio, por lo que se propone un modelo en el que se agrupen todos estos eventos en una única sesión, evitando así duplicidad de costes, es decir, un único servicio que distribuya estos eventos a las aplicaciones conforme van llegando.

Basándose en la W3C Web Push API donde se detalla el uso del servicio push en aplicaciones web -se explicará en un apartado posterior- se describirá un protocolo que puede ser usado para solicitar la entrega de un mensaje push a un agente de usuario, crear nuevas suscripciones de envío de mensajes push y monitorizar los nuevos mensajes push.

4.1.2 Definiciones

Aplicación: conjunto compuesto de emisor y receptor de mensajes push. Muchas de las aplicaciones tienen componentes que se ejecutan en el cliente y otras que van en el servidor.

Servidor de aplicación: componente de la aplicación que se ejecuta en el servidor y solicita el envío de mensajes push.

Agente de usuario (UA): receptor de mensajes push; puede ser un dispositivo o software.

Servicio push: conjunto de elementos que envía mensajes push a los UAs. Será el encargado de crear las suscripciones. La URL del servicio push es configurada en los UA.

Mensaje push: mensaje enviado desde el servidor de aplicación al UA a través de un servicio push.

Suscripción de mensajes push: contexto de envío de mensaje establecido entre el UA y el servicio push, además de ser compartido con el servidor de aplicación. Permite leer o eliminar el acceso a la propia suscripción de mensajes. Un UA recibe mensajes push usando dicha suscripción. Cada suscripción de mensajes push tiene exactamente un recurso push asociado.

Set de suscripción de mensajes push: contexto de envío de mensaje establecido entre el UA y el servicio push que contiene múltiples suscripciones dentro. Permite leer o eliminar el acceso al conjunto de suscripciones del set. Un UA recibe mensajes de todas las suscripciones correspondientes al set. Se le asocia un enlace de tipo "urn:ietf:params:push:set" para identificarlo.

Recurso push: Componente que usa un servidor de aplicación para solicitar el envío de mensajes push. Se le asocia un enlace de tipo "urn:ietf:params:push" para identificarlo.

Recurso de mensaje push: recurso creado por el servicio push para identificar los mensajes aceptados para el envío. Este recurso es eliminado por el UA para indicar el asentimiento del mensaje push.

Recurso de suscripción de recibo: componente usado por el servidor de aplicación para recibir los recibos de mensajes push. Un enlace de tipo "urn:ietf:params:push:receipt" lo identifica.

Recibo de mensaje push: confirmación de envío de mensaje enviada desde el servicio push al servidor de aplicación. Indica que el UA ha recibido y asentido el mensaje push.

4.1.3 Visión general

Un modelo general para un servicio push define tres actores básicos: el agente de usuario (UA), el servicio push y la aplicación, tal y como se puede observar en la imagen siguiente:

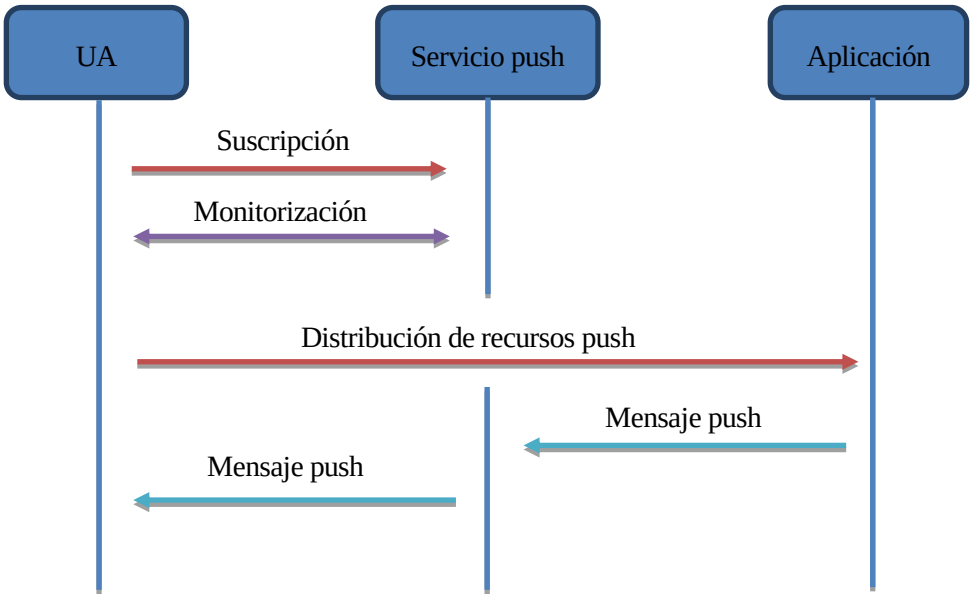


Figura 26 – Actores principales modelo push

En primer lugar, el UA crea un mensaje de suscripción para posteriormente ser distribuido a su servidor de aplicación. Este proceso será la base de todas las futuras comunicaciones entre actores.

Para ofrecer un mayor control, este proceso se divide en dos tipos de recursos independientes, cada uno con una funcionalidad distinta:

1. El recurso de suscripción, que es usado para recibir los mensajes correspondientes a una suscripción o eliminar una suscripción, es privado para el UA.
2. El recurso push, que es el encargado de enviar mensajes a una suscripción, es público y compartido por el UA a su servidor de aplicación.

Otro aspecto importante a tener en cuenta es que en una aplicación se pueden crear múltiples suscripciones, al igual que muchas aplicaciones pueden usar la misma suscripción.

Cabe destacar que cada suscripción tiene un tiempo de vida limitado pudiendo, también, ser eliminada por el servicio push o el propio UA.

Otros aspectos importantes como el recurso de suscripción de recibo se explicará con mayor detalle en el apartado 4.1.7.3.

4.1.4 Conexión al servicio push

El servicio push comparte el mismo puerto por defecto que HTTPS, el 443/TCP; pero también debe permitir anunciarse en el puerto 1001, usando los servicios alternativos HTTP.

4.1.5 Suscripción de mensajes push

El UA envía una solicitud POST a dicho servicio push, para crear una nueva suscripción. Algo parecido a lo siguiente:

```
POST /subscribe HTTP/1.1
Host: push.example.net
```

La respuesta a esta petición tendrá la siguiente forma:

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
 rel="urn:ietf:params:push"
Link: </subscription-set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
 rel="urn:ietf:params:push:set"
Location: https://push.example.net/subscription/LBhhw0OohO-Wl4Oi971UG
```

Donde se debe incluir:

- Un URI para un nuevo recurso de suscripción de mensajes push, en el campo Location de la cabecera.
- Un URI para el recurso push correspondiente a la suscripción de mensajes push, usando un enlace de tipo "urn:ietf:params:push".
- Un URI opcional para el set (conjunto) de recursos de suscripciones, usando un enlace de tipo "urn:ietf:params:push:set". Si se proporciona, el UA debería usar este enlace al set de suscripciones para recibir los mensajes push, en lugar de su enlace individual de suscripción.

Donde los identificadores son generados de forma aleatoria por el servicio push.

Por otro lado, para mejorar la eficiencia del servicio push, también es conveniente agrupar las suscripciones en un set de suscripciones. Por este motivo, el UA debe incluir en futuras peticiones el mismo set de suscripciones devuelto por la primera petición que realizó, tal y como se muestra en la siguiente solicitud:

```
POST /subscribe HTTP/1.1
Host: push.example.net
Link: </subscription-set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
 rel="urn:ietf:params:push:set"
```

Y el servicio push le devolverá, así, el mismo enlace al set de suscripciones, además de los nuevos enlaces para el recurso push y de suscripción creados.

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:52 GMT
Link: </push/YBJNBIMwwA_Ag8EtD47J4A>;
 rel="urn:ietf:params:push"
Link: </subscription-set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy>;
 rel="urn:ietf:params:push:set"
Location: https://push.example.net/subscription/i-nQ3A9Zm4kgSWg8_ZijVQ
```

4.1.6 Solicitud de envío de mensajes push

El servidor de aplicación solicita el envío de mensajes push enviando una solicitud HTTP POST al recurso push previamente distribuido por el UA a la aplicación. El mensaje push iría incluido en el cuerpo de esta solicitud:

```
POST /push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
TTL: 15
Content-Type: text/plain;charset=utf8
Content-Length: 36

iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB
```

Una respuesta afirmativa (201 Created) indicaría que el mensaje push ha sido aceptado. También se debe incluir el URI de recurso de mensaje push (previamente creado en la respuesta de la solicitud) en el campo Location:

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:56:55 GMT
Location: https://push.example.net/message/qDIYHNcfAIPP_5ITvURr-d6BGt
```

4.1.6.1 Protocolo de solicitud de envío de mensajes push

El servidor de aplicación puede incluir en el campo Prefer de la cabecera el término "respond-async", que solicita una confirmación del servicio push cuando el mensaje ha sido enviado y asentido por el UA.

Un ejemplo es el siguiente:

```
POST /push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
Prefer: respond-async
TTL: 15
Content-Type: text/plain;charset=utf8
Content-Length: 36

iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB
```

Una respuesta afirmativa (202 Acepted) indicaría que el mensaje push ha sido aceptado. También se debe incluir el URI de recurso push (previamente creado en la respuesta de la solicitud) en el campo Location. El servicio push también debe añadir un URI para el recurso de suscripción de recibo (en el apartado 4.1.7.3 se explicará esta cuestión con mayor detalle), en un enlace del tipo "urn:ietf:params:push:receipt":

```
HTTP/1.1 202 Accepted
Date: Thu, 11 Dec 2014 23:56:55 GMT
Link: </receipt-subscription/3ZtI4YVNBnUUZhuoChl6omUvG4ZM>;
 rel="urn:ietf:params:push:receipt"
Location: https://push.example.net/message/qDIYHNcfAIPP_5ITvURr-d6BGt
```

También destacamos que ante consecutivas solicitudes de recibo con el mismo origen, el servidor de aplicación puede indicar el recurso de suscripción de recibo en un enlace de tipo "urn:ietf:params:push:receipt". El servicio push también debe devolver el mismo recurso de recibo en su respuesta. Esto da la opción al servicio push de agregar los recibos.

4.1.6.2 TTL del mensaje

Cabe destacar que el servidor de aplicación debe incluir el campo TTL (Time-To-Live) en la cabecera de esta solicitud, indicando en segundos cuánto tiempo se puede retener el mensaje en el servicio push.

En caso de omission de este campo, el servicio push debe devolver una respuesta con código no válido (400 Bad Request).

En caso de que el TTL expire, el servicio push debe eliminar este mensaje.

Un mensaje con TTL=0 se envía inmediatamente al UA si está disponible para recibirlo, posteriormente el servicio push elimina este mensaje. Si el UA no estuviera disponible, se eliminaría de todas formas y no sería enviado.

4.1.6.3 Prioridad de mensajes push

Es frecuente que en dispositivos que funcionan con baterías se pase a estado inactivo cuando su uso no sea necesario; esto permite que el consumo de energía sea menor y así se prolongue la duración de la misma. Por este motivo, para evitar consumir recursos ante mensajes triviales, el servidor de aplicación puede comunicar la prioridad de estos mensajes y el tipo de prioridad de mensaje que quiere recibir cada UA del servicio push. Así, se añade un campo a la cabecera indicando la prioridad del mensaje.

Tabla 5 – Prioridades de mensajes push

Prioridad	Estado del dispositivo	Escenario de aplicación
Very-low	Pantalla encendida y con WiFi	Anuncios
Low	Pantalla encendida o con WiFi	Actualizaciones típicas
Normal	Pantalla apagada, sin WiFi	Mensajes de Chat o notificación de calendario
High	Con batería baja	Llamadas entrantes o alarmas

El servicio push no debe reenviar esta prioridad al UA. Y el UA solo debe incluir el campo prioridad cuando solo quiera solicitar mensajes push de ese nivel o superior.

4.1.6.4 Actualización de mensajes push

Un mensaje push almacenado en un servicio push puede ser actualizado con nuevo contenido, y, así, mientras el UA está desconectado, se evita que le lleguen mensajes redundantes o anticuados.

Cabe destacar que no todos los mensajes pueden ser reemplazados; solo pueden serlo aquellos que estén asignados a un *topic* (tema). De esta manera, un mensaje con un *topic* actualiza otro mensaje con el mismo *topic*.

Este campo *topic* se encuentra en la cabecera del mensaje y es una cadena de menos de 32 caracteres. Su funcionamiento es el siguiente:

1. Una solicitud de actualización de mensaje crea un nuevo recurso de mensaje push.
2. Se elimina cualquier recurso de mensaje con el mismo *topic*. Así conseguimos que no haya problemas con los asentimientos del mensaje anterior.
3. El servicio push debe eliminar todos los recibos de asentimiento del mensaje a reemplazar.

Por ejemplo, el siguiente mensaje:

```
POST /push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV HTTP/1.1
Host: push.example.net
TTL: 600
Topic: "upd"
Content-Type: text/plain;charset=utf8
Content-Length: 36

ZuHSZPKa2b1jtOKLGpWrcrn8cNqt0iVQyroF
```

actualizará el contenido de un mensaje anterior con el mismo topic, eliminará el recurso del mensaje anterior y se le enviará al UA una respuesta con el nuevo recurso de mensaje creado, con su URI correspondiente:

```
HTTP/1.1 201 Created
Date: Thu, 11 Dec 2014 23:57:02 GMT
Location: https://push.example.net/message/qDIYHNcfAIPP_5ITvURr-d6BGt
```

Donde el campo Topic no es enviado al UA, al igual que la prioridad.

4.1.7 Recepción de mensajes push de una suscripción

El UA solicita la entrega de nuevos mensajes push haciendo una solicitud de tipo GET al recurso de suscripción de mensajes push. El servicio push no responde a esta solicitud, sino que usa el estándar HTTP/2 Server Push (comentado en el capítulo 3) para enviar el contenido de los mensajes push en cuanto son recibidos del servidor de aplicación.

El UA puede incluir un campo de prioridad en la cabecera de su solicitud; en este caso, el servicio push solo debe entregar aquellos mensajes con una prioridad mayor o igual definido en Tabla 5 – Prioridades de mensajes push.

Cada mensaje push es enviado como respuesta a una solicitud GET sintetizada, en un PUSH_PROMISE. Esta solicitud GET se realiza al recurso de mensajes push, previamente creado por el servicio push cuando el servidor de aplicación solicitó el envío del mensaje.

La cabecera de respuesta debe proporcionar un URI para el recurso push correspondiente a la suscripción de mensajes push en un enlace del tipo "urn:ietf:params:push". El cuerpo de la respuesta es el mismo cuerpo que el de la última solicitud enviada por el servidor de aplicación al recurso push.

En el siguiente ejemplo vemos la solicitud sobre HTTP/2.

```
HEADERS [stream 7] +END_STREAM +END_HEADERS
 :method = GET
 :path = /subscription/LBhhw0OohO-Wl4Oi971UG
 :authority = push.example.net
```

La respuesta es la siguiente:

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
 :method = GET
 :path = /message/qDIYHNcfAIPP_5ITvURr-d6BGt
 :authority = push.example.net

HEADERS [stream 4] +END_HEADERS
 :status = 200
 date = Thu, 11 Dec 2014 23:56:56 GMT
 last-modified = Thu, 11 Dec 2014 23:56:55 GMT
 cache-control = private
 :link = </push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
 rel="urn:ietf:params:push"
 content-type = text/plain;charset=utf8
 content-length = 36

DATA [stream 4] +END_STREAM
 iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB

HEADERS [stream 7] +END_STREAM +END_HEADERS
 :status = 200
```

Donde se ve el entrelazado de las respuestas. Para ello, se observa que primero se recibe la cabecera correspondiente al stream 7, indicando en el mismo mensaje que el contenido continúa en el stream 4, así se garantiza el orden de mensajes al recibirlos.

Merece la pena recordar que los identificadores de :*path = XXXX*, siguen el mismo formato de antes, siendo generados de forma aleatoria por el servicio push; para hacer referencia en este caso a la suscripción y al mensaje, respectivamente. También es aplicacble al identificador de *:link =YYYY*, donde se hace referencia al servicio push.

4.1.7.1 Recepción de mensajes push de un set de suscripciones

Realmente existen pocas diferencias entre recibir mensajes push de una suscripción y de un set de suscripciones.

El UA solicita el envío de nuevos mensajes push enviando una petición GET al recurso de set de suscripciones. Al igual que antes, el servicio push no responde a esta solicitud sino que usa el estándar HTTP/2 Server Push para enviar el contenido de los mensajes push en cuanto son recibidos del servidor de aplicación.

El UA también puede incluir el campo prioridad en su cabecera para que el servicio push solo le envíe los mensajes con igual o mayor prioridad.

Cada mensaje push se envía como respuesta de la petición GET anterior en un PUSH_PROMISE, al igual que en caso anterior. También se añade un URI en un enlace de tipo "urn:ietf:params:push". Esto permite al UA diferenciar el origen del mensaje.

Un ejemplo de solicitud es el siguiente (usando HTTP/2):

```
HEADERS [stream 7] +END_STREAM +END_HEADERS
 :method = GET
 :path = /subscription-set/4UXwi2Rd7jGS7gp5cuutF8ZldnEuvbOy
 :authority = push.example.net
```

Y su respuesta:

```
PUSH_PROMISE [stream 7; promised stream 4] +END_HEADERS
 :method = GET
 :path = /message/qDIYHNcfAIPP_5ITvURr-d6BGt
 :authority = push.example.net
 :link = </push/JzLQ3raZJfFBR0aqvOMsLrt54w4rJUsV>;
 rel="urn:ietf:params:push"

HEADERS [stream 4] +END_HEADERS
 :status = 200
 date = Thu, 11 Dec 2014 23:56:56 GMT
 last-modified = Thu, 11 Dec 2014 23:56:55 GMT
 cache-control = private
 content-type = text/plain;charset=utf8
 content-length = 36

DATA [stream 4] +END_STREAM
 iChYuI3jMzt3ir20P8r_jgRR-dSuN182x7iB

HEADERS [stream 7] +END_STREAM +END_HEADERS
 :status = 200
```

4.1.7.2 Asentimientos de mensajes push

Para asegurar que un mensaje push es correctamente enviado al UA, este mismo debe asentir la recepción del mensaje eliminando el recurso de mensaje push correspondiente.

Para ello realiza un mensaje del tipo DELETE:

```
DELETE /message/qDIYHNcfAIPP_5ITvURr-d6BGt HTTP/1.1
Host: push.example.net
```

Si el servicio push recibe el asentimiento y la aplicación ha solicitado el recibo de envío, el servicio push debe enviar una respuesta afirmativa al servidor de aplicación monitorizando el recurso de suscripción de recibo.

Si el recurso push no recibe este asentimiento durante un tiempo fijado, se considera al mensaje como no enviado y el servicio push debe reenviarlo mientras no haya expirado su TTL.

4.1.7.3 Recibos de mensajes push

El servidor de aplicación solicita el envío de recibos del servicio push haciendo una petición GET al recurso de suscripción de recibo. El servicio push no responde a esta petición sino que manda usando HTTP/2 los recibos de los mensajes una vez han sido asentidos por el UA.

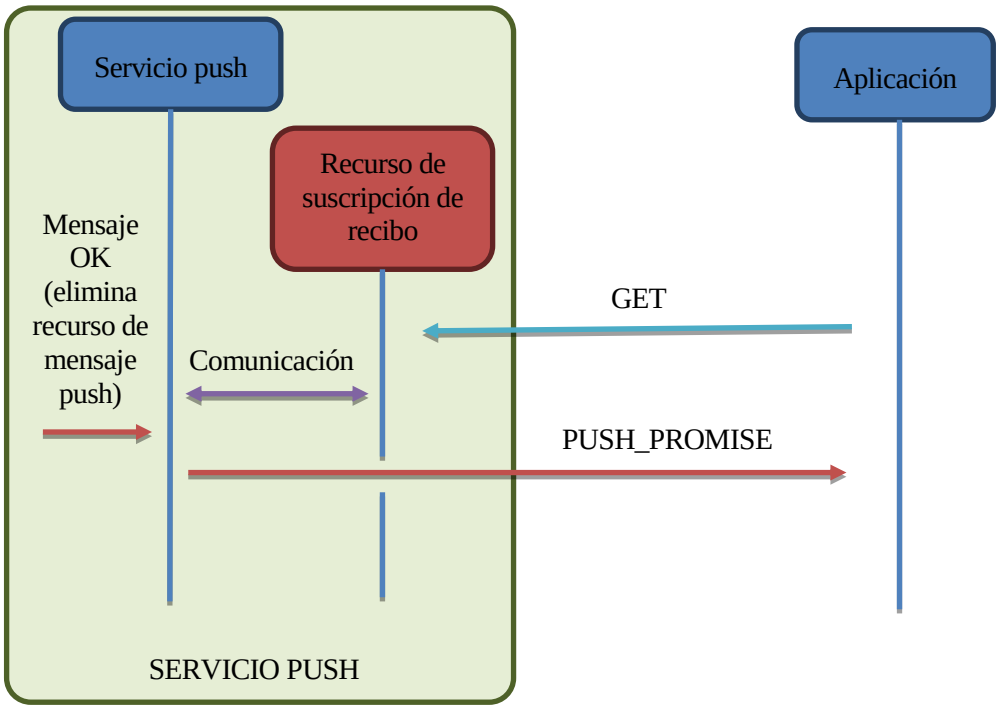


Figura 27 – Estructura con recurso de suscripción de recibo

Cada recibo se manda como respuesta al GET en una solicitud PUSH_PROMISE con el código de estado 204 (No Content) sin datos, indicando que el mensaje ha sido enviado al UA y asentido por el mismo.

Un ejemplo de solicitud es el siguiente (usando HTTP/2):

```
HEADERS [stream 13] +END_STREAM +END_HEADERS
 :method = GET
 :path = /receipt-subscription/3ZtI4YVNBnUUZhuoChl6omUvG4ZM
 :authority = push.example.net
```

La respuesta:

```
PUSH_PROMISE [stream 13; promised stream 82] +END_HEADERS
 :method = GET
 :path = /message/qDIYHNcfAIPP_5ITvURr-d6BGtYnTRnk
 :authority = push.example.net

HEADERS [stream 82] +END_STREAM +END_HEADERS
 :status = 204
 date = Thu, 11 Dec 2014 23:56:56 GMT
```

Cabe destacar que el servidor de aplicación se suscribe una única vez para la recepción de estos recibos. Será el recurso de suscripción de recibo quien se encarga de gestionar todo el proceso de recepción de recibos y comunicárselo al servicio push, que posteriormente mandará dichos recibos al servidor de aplicación.

4.2 Web Push API

4.2.1 Introducción

En esta especificación de la W3C (World Wide Web Consortium) se propone un procedimiento para enviar un mensaje push a una página web, a través de un servicio push. Así, el servidor de aplicación puede enviar mensajes push en cualquier momento, incluso cuando la aplicación web o el agente de usuario (UA) están inactivos. El servicio push asegura una entrega fiable y eficiente al agente de usuario. [28]

Los mensajes push son enviados a un Service Worker que se ejecuta en la web, y puede usar la información contenida en el mensaje para actualizar su estado local o mostrar una notificación al UA.

Esta especificación está diseñada para usar el protocolo WEBPUSH de la IETF antes comentado, donde se describe cómo el servidor de aplicación y el agente de usuario interactúan con el servicio push.

4.2.2 Definiciones

Web Worker: elemento en JavaScript de una página HTML que se ejecuta en segundo plano, independiente del resto de componentes JavaScript que contenga la página. Está definido por el W3C y el WHATWG.

WebRTC: API elaborada por el W3C que permite a las aplicaciones web realizar llamadas de voz, chat de video y uso compartido de archivos P2P sin necesidad de plugins adicionales.

Service Worker: servicio que se ejecuta en segundo plano que maneja peticiones. Hoy día se utiliza para recepción de mensajes push, trabajar con páginas offline, etc.

4.2.3 Contextos y casos de uso

Como se define en el documento, los servicios push solo soportan el envío de mensajes del servidor de aplicación en los siguientes contextos y casos de uso:

1. El usuario está activo usando la aplicación web. Es el caso de uso normal y en el que puede beneficiarse más de los mensajes push.
2. El usuario está inactivo, pero la aplicación web está activa o ejecutándose como un Web Worker. Es el caso de redes sociales, mensajería, lectores de documentos online, etc., donde el usuario puede mantenerse al tanto sin estar involucrado usando la web, por lo que se beneficia, también, de los mensajes push.
3. La aplicación web no está activa en un navegador. Es el caso en el que el usuario ha cerrado la aplicación pero sigue permitiendo que esta se reinicie cuando se reciba un mensaje push. Un ejemplo sería WebRTC, cuando una llamada entrante en tu dispositivo móvil puede invocar a la aplicación web correspondiente para recibir la llamada.
4. Existen múltiples aplicaciones web ejecutándose, pero los mensajes push solo llegan a aquella que los ha solicitado. Sucede cuando se tienen abiertas y activas varias páginas web, usando servicios push, en el navegador.
5. Cuando hay diferentes instancias de la misma aplicación web en el mismo navegador y el mensaje push va dirigido solo a una de ellas. Un ejemplo sería si se tienen abiertas dos ventanas del correo electrónico con diferentes cuentas.
6. Cuando hay varias instancias de la misma aplicación web en diferentes navegadores y el mensaje push va a todas. Esta situación se daría, por ejemplo, si se tiene abierto el correo con la misma cuenta en diferentes navegadores.
7. Cuando hay varias instancias de la misma aplicación web en diferentes navegadores y los mensajes push son enviados a todas las instancias de la aplicación web. Es el caso de un mensaje de difusión a todos los usuarios de una web.
8. El agente de usuario no tiene conexión a la red temporalmente. En este caso depende del servicio push el envío de estos mensajes, si la aplicación soporta cambios si está offline.

4.2.4 Diagrama de secuencia

En el siguiente diagrama se muestra un resumen de la estructura que propone esta API, incluyendo el paso de mensajes:

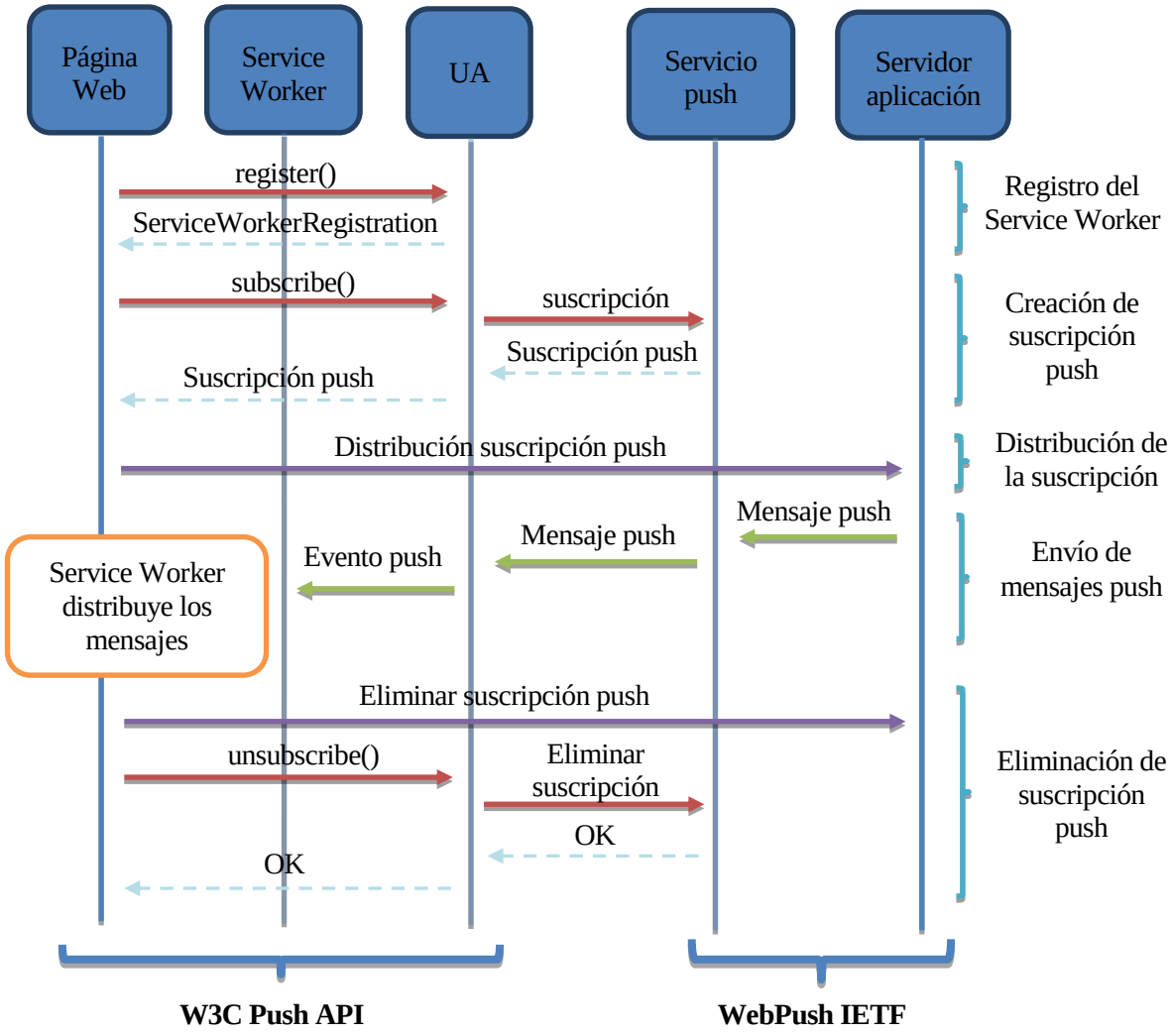


Figura 28 – Diagrama de secuencia WebPush API

Donde se observan 5 partes fundamentales, que se explicarán brevemente en los siguientes apartados.

4.2.4.1 Registro del Service Worker

En esta primera fase se realiza el registro del Service Worker para que reciba los eventos push correspondientes a mensajes que envía la aplicación.

Para ello la página web envía un mensaje de registro al UA, indicando los parámetros necesarios para esta acción.

4.2.4.2 Creación de la suscripción push

Como se ha comentado en apartados anteriores, para poder recibir mensajes push es necesario crear una suscripción a un servicio push. Para ello la página web manda un mensaje de suscripción al UA, para que posteriormente se suscriba al recurso push correspondiente.

4.2.4.3 Distribución de la suscripción push

Una vez creada la suscripción, la página web debe distribuir la misma a la aplicación. Así podrá enviar los mensajes push gracias a esta.

4.2.4.4 Envío de mensajes push

Esta es la fase en la que el servidor envía los diferentes mensajes push al servicio push. Posteriormente, este los distribuye al UA para que a través de un evento push sea enviado al Service Worker.

Destacamos que la aplicación puede procesar el mensaje push como desee, usando las características disponibles del Service Worker.

4.2.4.5 Eliminación de suscripción push

En esta última fase, se cierra la suscripción con la aplicación. Esto puede realizarse cuando ya no se quiera seguir recibiendo mensajes push correspondientes a la suscripción.

Para ello la página web comunica el cierre de conexión a la aplicación, a la vez que elimina la suscripción creada con el UA y el servicio push.

En el caso de que la página se cierre inesperadamente o se produzca algún error en la red, estas suscripciones push tienen un tiempo de actividad, por lo que si este tiempo expira, se produce un evento indicando que la suscripción se ha invalidado. Este tiempo lo asigna el servicio push.

5 APLICACIÓN PRÁCTICA

En una hora de juego se puede descubrir más acerca
de una persona que en un año de conversación.

Platón

En esta sección se explicará como se ha implementado una aplicación web interactiva que haga uso de todos los conceptos teóricos explicados anteriormente.

Concretamente se usará las tecnologías y Frameworks Java Server Faces (JSF) y Primefaces para el desarrollo de la interfaz de usuario de la aplicación, además de algunos Frameworks para manejar WebSockets, como son atmosphere y Orbiter. En los siguientes subapartados se explicará con mayor detalle todas estas cuestiones.

La aplicación consistirá en un pictionary interactivo, en el que un usuario pintará un dibujo y el resto tendrá que adivinar qué es. Toda la estructura de aplicación se organizará mediante el paso de mensajes Websockets.

5.1 Java Server Faces (JSF)

Java Server Faces es un Framework basado en el patrón de arquitectura Modelo-Vista-Controlador (MVC), para el desarrollo de aplicaciones web dinámicas.

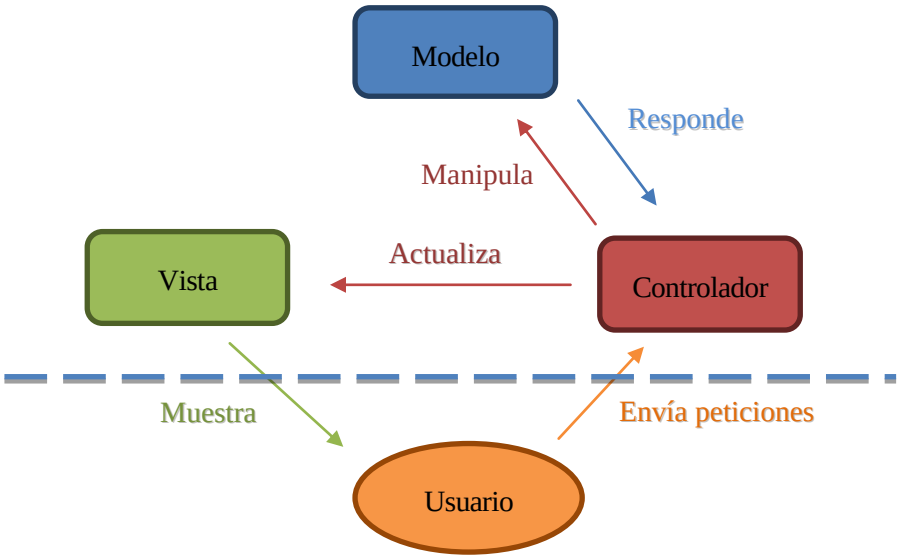


Figura 29 – Estructura Modelo-Vista-Controlador

JSF simplifica mucho el diseño de la interfaz de usuario ya que se basa en el uso y reutilización de componentes dentro de la página.

El estándar JSF define un conjunto de APIs (Application Programming Interface) para representar dichos componentes de interfaz de usuario y administrar su estado, manejar eventos, validar entradas, definir un esquema de navegación de las páginas y dar soporte para internacionalización y accesibilidad.

Otra de las principales características de JSF es que incluye dos librerías de etiquetas para la representación de sus elementos. Por otro lado, incluye un modelo de eventos en el lado del servidor y el uso de beans administrados (clases en java para controlar la aplicación).

En torno a la programación usando JSF, se definen los siguientes objetivos principales:

- Definir un conjunto simple de clases base de Java para componentes de la interfaz de usuario, estado de los componentes y eventos de entrada. Estas clases tratarán los aspectos del ciclo de vida de la interfaz de usuario, controlando el estado de un componente durante el ciclo de vida de la página.
- Proporcionar un conjunto de componentes para la interfaz de usuario, incluyendo los elementos estándares de HTML para representar un formulario. Estos componentes se obtendrán de un conjunto básico de clases base que se pueden utilizar para definir componentes nuevos.
- Proporcionar un modelo de JavaBeans para enviar eventos desde los controles de la interfaz de usuario del lado del cliente a la aplicación del servidor.
- Definir APIs para la validación de entrada, incluyendo soporte para la validación en el lado del cliente.
- Especificar un modelo para la internacionalización y localización de la interfaz de usuario.
- Automatizar la generación de salidas apropiadas para el objetivo del cliente, teniendo en cuenta todos los datos de configuración disponibles del cliente, como versión del navegador.

5.1.1 Versiones JSF

La primera versión de JSF salió el año 2004, hasta entonces se han publicado diferentes nuevas versiones de la misma, entre las que se destacan:

Tabla 6 – Versiones JSF

Versión	Año de lanzamiento	Descripción
JSF 1.0	11-03-2004	Lanzamiento inicial.
JSF 1.1	27-05-2004	Solucionaba errores. Sin cambios en las especificaciones.
JSF 1.2	11-05-2006	Lanzamiento con mejoras y correcciones de errores. Adopción inicial con Java EE.
JSF 2.0	12-08-2009	Lanzamiento con mejoras de funcionalidad, rendimiento y facilidad de uso.
JSF 2.1	22-10-2010	Lanzamiento de mantenimiento, con cambios mínimos.
JSF 2.2	16-04-2013	Versión actual, introduce soporte a HTML5, Faces Flow, Stateless views y Resource library contracts

5.1.2 Aspectos importantes de JSF

En este apartado se esplicarán algunos puntos importantes de JSF para entender la aplicación, entre los que se encuentran Managed Beans y las extensiones.

5.1.2.1 Managed Beans

Los Managed Beans son clases en Java con sus propiedades y métodos getter-setter que pueden ser manejados desde una página JSF. Por tanto, gracias a los Managed Beans, tenemos definido el modelo de la aplicación.

Cada usuario de la página tiene una instancia diferente de dicho Bean, para así conectar los campos de la página JSF con los atributos del Managed Bean.

En las primeras versiones de JSF era necesario declararlas en ficheros XML, sin embargo, a partir de la especificación 2.0 no es necesario; con solo declarar una anotación del tipo @ManagedBean sobre la clase Java es suficiente.

Otro de los aspectos más imporatntes de los Beans es su ámbito, es decir el tiempo de vida que tienen en el ciclo de vida de la aplicación. Existen los siguientes tipos:

Tabla 7 – Ámbitos de los Managed Beans

Tipo	Descripción
@RequestScoped	Para el ámbito de petición. Al terminar la petición se elimina la instancia del Bean.
@SessionScoped	Para el ámbito de sesión. El Bean estará instanciado mientras la sesión exista.
@ViewScoped	Para el ámbito entre Request y Session. El Bean existirá mientras la petición se envíe a la misma vista.
@ApplicationScoped	Para el ámbito de la aplicación Web. Vivirá mientras la web se ejecute.

Un ejemplo de uso en la aplicación es el Bean que se usa para almacenar los usuarios o las puntuaciones del juego, que tienen ámbito de aplicación, como se ve en la siguiente imagen:

```
@ManagedBean
@ApplicationScoped
public class PictionaryUsers implements Serializable {

    private List<String> users;

    @PostConstruct
    public void init() {
        users = new ArrayList<String>();
    }
    ...
```

5.1.2.2 Extensiones

Otro de los aspectos más importantes de JSF es que permite el uso de extensiones. Aquí se observa la potencia de este Framework, ya que se complementa y extiende gracias a ellas.

Entre las más extendidas están las siguientes:

Tabla 8 – Extensiones JSF

Extensión	Características
RichFaces	Agrega componentes visuales y soporte para AJAX.
ICEfaces	Contiene diversos componentes para interfaces de usuarios más enriquecidas, tales como editores de texto enriquecidos, reproductores multimedia.
jQuery4jsf	Contiene diversos componentes sobre la base de jQuery.
PrimeFaces	Es una librería muy pequeña en tamaño, conteniendo muchos componentes y todo sin dependencias y configuraciones.
OpenFaces	Librería open source que contiene diferentes componentes JSF, un Framework Ajax y un Framework de validación por parte del cliente.

Para la realización de la aplicación se usará la extensión **PrimeFaces**, que proporciona una serie de componentes y temas muy fáciles de implementar. Además de una página web con demos y las etiquetas correspondiente a cada componente. [29]

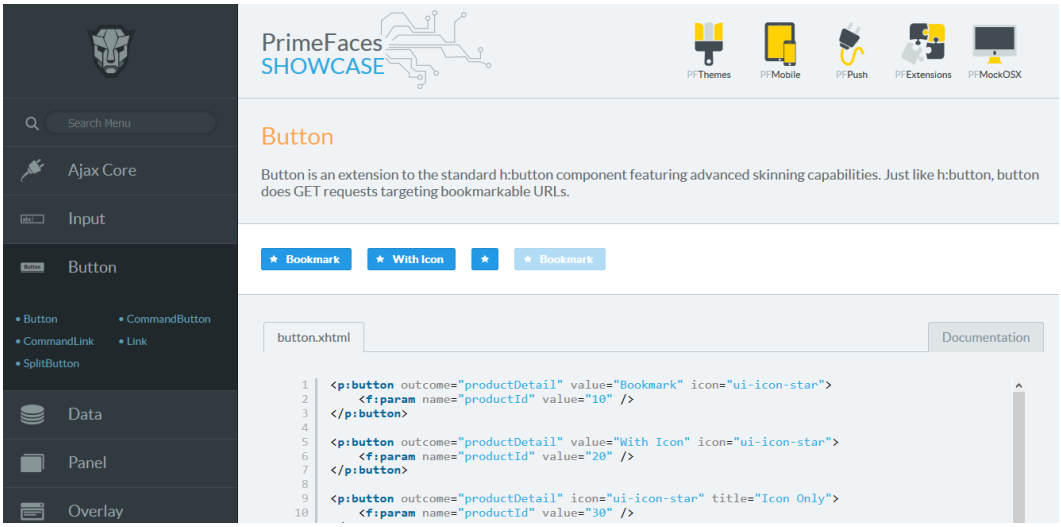


Figura 30 – PrimeFaces Demo

Para el uso de PrimeFaces en el proyecto, lo único que hay que hacer es descargar el *.jar* desde la página oficial al mismo. Otra opción es incluirlo en una dependencia en un proyecto Maven.

Maven es una herramienta de software para la gestión y construcción de proyectos Java, donde se define un modelo de configuración de construcción muy simple, basado en un formato XML. Para ello, se incluye en un fichero de configuración (*pom.xml*) todas las dependencias y configuraciones necesarias del proyecto.

Para el proyecto se ha seguido el modelo basado en Maven, donde se añade la siguiente dependencia:

```
<dependency>
   <groupId>org.primefaces</groupId>
   <artifactId>primefaces</artifactId>
   <version>5.3</version>
</dependency>
```

Donde se indica la versión que se va a utilizar, en esta aplicación se usará la versión 5.3.

Por otro lado, PrimeFaces permite añadir el uso de temas y extensiones, que posibilita que se disponga de más componentes y a su vez darle diferentes aspectos. Para ello, simplemente hay que añadir en el fichero de dependencias Maven lo siguiente:

```
<dependency>
    <groupId>org.primefaces.extensions</groupId>
    <artifactId>primefaces-extensions</artifactId>
    <version>4.0.0</version>
</dependency>

<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>all-themes</artifactId>
    <version>1.0.10</version>
</dependency>
```

Por último, hay que declarar este tipo de etiquetas en el documento XHTML de la siguiente forma:

```
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:c="http://java.sun.com/jsp/jstl/core"
    xmlns:p="http://primefaces.org/ui"
    xmlns:pe="http://primefaces.org/ui/extensions">
```

Donde las dos últimas etiquetas se corresponden con los componentes de PrimeFaces y sus extensiones.

5.2 Atmosphere Framework

Atmosphere es un Framework que contiene tanto el cliente y servidor para la construcción de aplicaciones web asíncronas. La mayoría de Frameworks populares para el diseño web, tal como PrimeFaces, soporta Atmosphere. Por otro lado, es soportado por la mayoría de navegadores actuales. [30]

Atmosphere proporciona transparencia para la implementación de WebSockets, Server Sent Events (SSE), Long-Polling, HTTP Streaming y JSONP. Simplemente basta con indicar el método deseado y serguir la estructura de implementación que se explicará posteriormente.

La estructura de Atmosphere es la siguiente:

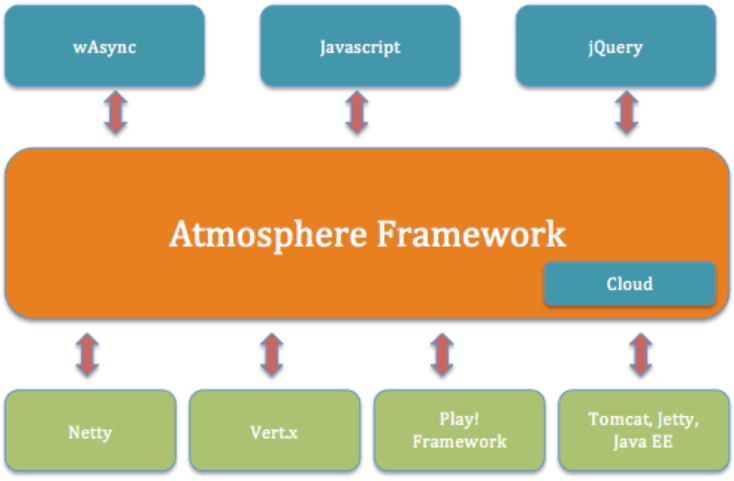


Figura 31 – Estructura de Atmosphere [30]

Atmosphere funciona sobre la mayoría de servidores basados en Servlets , tales como Tomcat, JBoss Jetty, Resin, GlassFish, Undertow, WebSphere, WebLogic, etc. Además tiene soporte para una variedad de extensiones, como STOMP, Redis, Hazelcast, JMS, Jgroups entre otras.

Por otro lado, cabe destacar que Atmosphere también incluye el soporte de otros Framework en tiempo real como Socket.io, SockJS o Cometd. Por lo que su uso no se hace exclusivo, sino que se puede adaptar al Framework deseado. En [31] existe una lista de todas las extensiones y Frameworks soportados por Atmosphere.

Para el desarrollo de la aplicación, Atmosphere se ha integrado con PrimeFaces, proporcionando todos los elementos necesarios para desarrollar una aplicación que siga el modelo push.

Entre los principales objetivos de esta tecnología se encuentra el permitir a los desarrolladores crear una aplicación y de manera transparente. Atmosphere se encarga de elegir cual es el mejor canal de comunicación entre el cliente y el servidor. Con lo que es altamente soportado por todos los navegadores (nuevos y no tan nuevos).

Por ejemplo, con Atmosphere un desarrollador puede escribir una aplicación que use el protocolo WebSockets si el navegador lo permite, y usar otro protocolo como Long-Polling en caso contrario, todo de forma transparente. Un ejemplo claro sería el funcionamiento de la aplicación con Long-Polling en Internet Explorer 6,7,8 y 9; mientras que en Internet Explorer 10 ya sí se usaría WebSockets (soportado en esta versión).

5.2.1 Incorporación de Atmosphere al proyecto

En primer lugar, para la incorporación de Atmosphere, hay que añadir al fichero *pom.xml* la siguiente dependencia.

```
<dependency>
    <groupId>org.atmosphere</groupId>
    <artifactId>atmosphere-runtime</artifactId>
    <version>2.4.2</version>
</dependency>
```

En este caso se ha incorporado la versión 2.4.2, que requiere una versión de JDK igual o superior a la 1.7.

5.2.2 Atmosphere y PrimeFaces

PrimeFaces propone un modelo de diseño de aplicaciones push construido sobre Atmosphere, siendo altamente escalable y válido para aplicaciones web interactivas y dinámicas.

Lo primero es definir el Servlet que servirá de pasarela para los clientes. Para ello se define lo siguiente en el fichero web.xml del proyecto:

```
<servlet>
    <servlet-name>Push Servlet</servlet-name>
    <servlet-class>org.primefaces.push.PushServlet</servlet-class>
    <async-supported>true</async-supported>
</servlet>

<servlet-mapping>
    <servlet-name>Push Servlet</servlet-name>
    <url-pattern>/primepush/*</url-pattern>
</servlet-mapping>
```

Una vez configurados las dependencias y el Servlet se procede a analizar el modelo push de esta tecnología.

Todo el diseño se basa en el uso de anotaciones, que serán incluidas en las clases Java para definir el funcionamiento de la aplicación. En el siguiente subapartado se explica con mayor detalle las más importantes.

5.2.2.1 Anotaciones

Como ya se ha comentado antes, las anotaciones juegan un papel fundamental en la programación que sigue el modelo MVC y JSF, ya que gracias a ellas se pueden definir comportamientos, componentes, recursos, etc. A continuación se listan y explican las más importantes.

@PushEndPoint

El diseño de una aplicación siguiendo el esquema que propone PrimeFaces Push es centralizado sobre la clase principal anotada con @PushEndPoint. Gracias a esta anotación, se simplifica el proceso de construcción del modelo push de Atmosphere sin la necesidad de interaccionar con su API de forma más sofisticada. Así se reduce notablemente la cantidad de código requerido para la construcción de una aplicación en tiempo real.

Esta anotación tiene un parámetro principal llamado *path*, donde se define la ruta del recurso. La forma de declararla es la siguiente:

```
@PushEndPoint("/pictionaryroom")
```

Gracias a esta ruta, se consigue que todas las solicitudes se envíen a dicha clase anotada para que se procesen y así se realicen las acciones pertinentes que la aplicación requiera.

@Singleton

Gracias a esta anotación se consigue que solo se cree una única instancia de la clase anotada con la anotación anterior PushEndPoint. Así se evitan conflictos de que haya varias instancias de dicha clase.

@OnOpen

Esta anotación se incluye a un método para que se ejecute cuando la conexión está preparada para usarse, por ejemplo para escribir o enviar mensajes. Existen tres formas de declararlas, que son las siguientes:

```
@OnOpen
public void onOpen();

@OnOpen
public void onOpen(RemoteEndpoint r);

@OnOpen
public void onOpen(RemoteEndpoint r, EventBus e);
```

Donde RemoteEndpoint representa la conexión física y puede ser usada para enviar o devolver datos al navegador. Y EventBus se usa para enviar mensajes a uno o varios RemoteEndpoints.

Sin embargo, en la versión 5.3 de Atmosphere existe un bug donde el EventBus se resuelve a null, por lo que hay que recurrir a su obtención manual y no usando la tercera forma. Para ello hay que realizar lo siguiente:

```
EventBus eventBus = EventBusFactory.getDefault().eventBus();
```

Así se obtiene el bus por defecto para enviar los mensajes deseados.

@OnMessage

Esta anotación se coloca sobre un método que será invocado cuando el mensaje esté preparado para el envío. Por ejemplo, cuando se usa el EventBus para publicar datos o cuando el navegador realiza el envío POST de datos.

Existen dos atributos que se pueden añadir a la anotación; son los siguientes:

- Encoders: donde se añade una clase que codificará el mensaje.
- Decoders: usado para que una clase decodifique el mensaje en un objeto de la aplicación.

El ejemplo de uso en la aplicación es el siguiente:

```
@OnMessage(decoders = {MessageDecoder.class},
           encoders = {MessageEncoder.class})
     public Message onMessage(Message message) {
         return message;
     }
```

Donde *MessageEncoder* será el codificador del mensaje en formato JSON, mientras que *MessageDecoder* se usará para convertir un determinado tipo de mensaje a un objeto de tipo *Message*, tal y como se observa en las siguientes líneas de código:

MessageEncoder.java

```
public final class MessageEncoder implements Encoder<Message, String>
{

    public String encode(Message message) {
        return new JSONObject(message).toString();
    }
  }
```

Donde se hace uso de la librería *org.primefaces.json.JSONObject* para codificar un mensaje a formato JSON de manera muy sencilla.

MessageDecoder.java

```
public class MessageDecoder implements Decoder<String,Message> {

  public Message decode(String s) {
    String[] userAndMessage = s.split(":");

    if (userAndMessage.length == 2) {
      return new Message().setUser(userAndMessage[0])
                          .setText(userAndMessage[1]);
    } else if (userAndMessage.length == 3) {
      return new Message().setUser(userAndMessage[0])
                          .setText(userAndMessage[1])
                          .setColor(userAndMessage[2]);
    } else {
      return new Message(s);
    }
  }
}
```

Donde se decodifica un mensaje en función de los elementos que contenga la cadena. Si la cadena contiene dos elementos separados por ":" entonces se creará un objeto Message con usuario y texto; si la cadena contiene tres elementos separados por ":" entonces se creará un objeto Message con usuario, texto y color. En caso contrario se envía el mensaje tal cual, solo inicializando el campo texto.

Así se consigue crear los diferentes objetos Message en función de la cadena que enviemos, todo de manera inmediata y sin mucha dificultad.

@OnClose

Esta anotación es muy útil para que el método anotado se invoque cuando la conexión se cierra, ya sea intencionadamente o por un cierre inesperado de la página o problemas en la red.

Al igual que con OnOpen, existen tres formas de declararla:

```
@OnClose
public void onClose();

@OnClose
public void onClose(RemoteEndpoint r);

@OnClose
public void onClose(RemoteEndpoint r, EventBus e);
```

Así se consigue de manera transparente la gestión de fallos de la aplicación, ya que siempre que la conexión se cierre se invocará al método para asegurar el correcto cierre de la aplicación; ya sea para borrar al usuario del sistema o para cerrar todas las conexiones de manera segura.

@PathParam

Por último, esta anotación es útil cuando se desea crear una ruta que no sea estática, ya sea para tener varios PushEndpoint o para la gestión de muchos usuarios dentro de un canal, todo de manera automática.

En el ejemplo de la aplicación se usa de la siguiente manera:

```
@PushEndpoint("/pictionaryroom/{user}")
@Singleton
public class PictionaryResource {

    @PathParam("user")
    private String username;
...
```

Así se consigue tener una sala común llamada */pictionaryroom/* y, por cada usuario que se conecte se dispondrá, además, de una ruta propia por usuario, para el caso en que se desee enviar mensajes a un determinado jugador. Por ejemplo, si un usuario Alejandro se conecta al sistema, dispondrá de la ruta general */pictionaryroom/* para recibir mensajes globales, y la ruta */pictionaryroom/Alejandro/* para mensajes exclusivos.

5.2.2.2 Atmosphere API

Para concluir con la explicación de Atmosphere, se detallarán dos aspectos importantes a tener en cuenta de su API, como son las clases RemoteEndpoint y EventBus.

Ambas son usadas en el proyecto con frecuencia, ya que gracias a ellas conseguimos información de las conexiones, además de poder realizar el envío de información por el bus.

RemoteEndpoint

Esta clase representa la conexión remota, por ejemplo el navegador. Gracias a ella se consigue información de la conexión, como las cabeceras, los URI, el body, path, etc. que pueden usarse para modificar las solicitudes, o simplemente para ver que todo es correcto.

En la aplicación se utiliza para imprimir en el log información de las aperturas de conexiones, cierres y envíos de mensajes. Para ello se incluye la siguiente línea:

```
public void onOpen(RemoteEndpoint r) {
  logger.info("OnOpen {}", r);
...
```

Donde se obtiene una salida como la siguiente; si un usuario llamado Alejandro entra a la aplicación y se abre su conexión WebSocket:

```
INFO  e.u.d.e.p.PictionaryResource - OnOpen RemoteEndpointImpl
{  request=AtmosphereRequest {
      method=GET
      contextPath=/Application
      servletPath=/primepush
      pathInfo=/pictionaryroom/Alejandro
      requestURI=/Application/primepush/pictionaryroom/Alejandro
      requestURL=http://localhost:8080/Application/primepush/
                                          pictionaryroom/Alejandro
      AtmosphereResource
      UUID=d9a41b7b-fd5e-492e-9635-5ced2723f824
      destroyable=false
      },
   uri='/Application/primepush/pictionaryroom/Alejandro',
   status=Status{status=OPEN}
}
```

Donde se observan los parámetros más importantes de la solicitud de conexión, como el método usado (GET), los diferentes *path*, los URI y URL correspondientes a la conexión, el estado de la conexión (OPEN), entre otras cosas.

EventBus

La otra clase importante para entender la aplicación es EventBus, donde se define el bus de comunicaciones entre los diferentes PushEndpoint. Seguirá un modelo de cola único, donde el publicador envía los mensajes a los usuarios subscritos conforme le van llegando.

Por tanto este será el canal de comunicación entre los diferentes usuarios, para enviar los diferentes mensajes requeridos por la aplicación.

En primer lugar, para poder usar dicho bus es necesario obtenerlo de la siguiente forma:

```
EventBus eventBus = EventBusFactory.getDefault().eventBus();
```

Una vez que se tiene el bus, ya se puede utilizar para publicar información a los demás usuarios. Para la aplicación se usará el método *publish*. Existen tres implementaciones del método publish para realizar dicha acción:

```
publish(Object o)

publish(String path, Object o)

publish(String path, Object o, Reply reply)
```

La primera implementación envía el objeto a todos los RemoteEndpoints conectados. La segunda se usará para enviar el objeto a los RemoteEndpoints conectados al path indicado. Por último, la tercera implentación realiza lo mismo que la segunda, pero además añade una instancia de respuesta, que se invocará cuando el mensaje ha llegado al destino del path indicado, gracias al callback *completed(String path)*.

Por tanto, para la aplicación se usará la primera implentación cuando se desee enviar un mensaje a todos los usuarios de la partida, mientras que se usará la segunda para mensajes exclusivos para algún usuario en concreto. En apartados posteriores se definirán y explicarán los diferentes mensajes que usa la aplicación.

Un ejemplo de la aplicación donde se envía un mensaje a todos los usuarios es el siguiente:

```
eventBus.publish(CHANNEL + "*", username + ": " + globalMessage + ": "
                                                        + color);
```

5.2.2.3 Socket

El último aspecto relevante para comprender la arquitectura PrimeFaces+Atmosphere es el elemento socket. Este se encargará de conectar el navegador con el servidor, para así manejar la recepción y envío de mensajes de la aplicación.

Para ello hay que declararlo dentro de la página de la siguiente forma:

```
<p:socket onMessage="handleMessage"
          channel="/pictionaryroom"
          autoConnect="false"
          widgetVar='subscriber2' />
```

Donde handleMessage será una función en JavaScript para decidir lo que hacer con el mensaje recibido, channel será el canal al que se conectará el webSocket, autoConnect estará deshabilitado para que no se conecte al cargar la página y widgetVar será el nombre que tendrá el socket para referirnos al componente de la parte cliente, en este caso se llamará subscriber2.

En la siguiente lista se tienen todos los atributos que se pueden incorporar a la etiqueta socket:

Tabla 9 - Atributos etiqueta socket PrimeFaces

Nombre	Descripción
id	Identificador único del elemento.
rendered	Esta opción indica si el componente se renderiza, si se establece a *false* no se renderizará.
binding	Una expresión EL (Expression Language) referida a la parte servidora de la instancia de UIComponent del Bean.
widgetVar	Nombre del Widget de la parte del cliente del componente.
channel	Nombre del canal de la conexión.
transport	Protocolo deseado para la comunicación. Los valores posibles son websocket (por defecto), sse, streaming, long-polling, jsonp.
fallbackTransport	Protocolo a usar en caso de que el indicado en el atributo anterior no sea soportado por el navegador. Los valores posibles son websocket, sse, streaming, long-polling (por defecto), jsonp.
onMessage	Función Javascript a ejecutar cuando se reciben datos por el socket.
onError	Función JavaScript a ejecutar cuando se produce un error en la comunicación del socket.
autoConnect	Se define para indicar si se debe conectar automáticamente al socket al cargar la página.

Como se ha comentado en puntos anteriores, una de las principales características de este modelo es que de forma transparente se decide el protocolo a usar para el transporte y así se percibe la potencia de Atmosphere.

5.3 Orbiter Framework

El otro Framework a utilizar para el manejo de WebSockets en la aplicación es Orbiter. Este Framework se usará exclusivamente para manejar la pizarra del pictionary y así quitar carga al servidor que organiza el juego (turnos, puntuaciones, reinicios, etc).

Orbiter es un Framework escrito en JavaScript para el diseño de aplicaciones en tiempo real que hace uso del protocolo WebSocket y en su defecto HTTP Long Polling. Está preparado para reducir el consumo de ancho de banda y así incrementar la velocidad de comunicación.

5.3.1 Incorporación de Orbiter al proyecto

La incorporación de este Framework al proyecto es muy sencilla, simplemente hay que añadir una librería, que consiste en un único fichero JavaScript, a la página. Para ello se incluye la siguiente línea dentro del *head* de la página:

```
<script type="text/javascript"
 src="http://cdn.unioncloud.io/OrbiterMicro_latest_min.js"></script>
```

En el proyecto esta librería se ha descargado e incorporado al mismo, para que no haya que descargarla del exterior.

Una vez incluida ya se puede pasar a la implementación que propone Orbiter para el manejo de WebSockets.

5.3.2 Inicialización del cliente

Su funcionamiento es muy sencillo, completamente basado en JavaScript. Para su uso se crea un fichero *.js* exclusivo. Posteriormente, se procede a la inicialización de la conexión siguiendo los pasos que se especifican:

Se crea la instancia Orbiter, que se usará para comunicarse con el servidor:

```
orbiter = new net.user1.orbiter.Orbiter();
```

Posteriormente, se registran los eventos de conexión de la siguiente forma:

```
orbiter.addEventListener(net.user1.orbiter.OrbiterEvent.READY,
                         readyListener, this);
orbiter.addEventListener(net.user1.orbiter.OrbiterEvent.CLOSE,
                         closeListener, this);
```

Así se disponen de las funciones para cuando la conexión se abra y se cierre, respectivamente.

Posteriormente, hay que obtener una referencia del Manager para el envío y recepción de mensajes del servidor, que se realiza de la siguiente forma:

```
msgManager = orbiter.getMessageManager();
```

Por último, hay que establecer la conexión con el servidor. En este caso de aplicación se ha usado el servidor de Union, disponible en [32]. Para establecer dicha conexión se realiza lo siguiente:

```
orbiter.connect("localhost", 9110);
```

Donde se ha indicado la dirección del servidor y el puerto de escucha. En este caso el servidor está alojado en la máquina local y escuchando en el puerto por defecto (9110).

Una vez realizados estos pasos, está todo listo para manejar el envío de mensajes entre navegadores y así controlar la pizarra de la aplicación.

5.3.3 Envío de mensajes con Orbiter

Para el envío de mensajes usando Orbiter se usará el protocolo Union Procedure Call (UPC). UPC es el protocolo central en las comunicaciones de las plataformas Union. En él se describen el formato del mensaje y una lista de mensajes soportados por el servidor Union. [33]

De forma estructural, un mensaje UPC sigue el formato XML que incluye el identificador (ID) del mensaje y una lista de argumentos. Por lo que un mensaje UPC tiene la siguiente forma:

```
<u>
  <m>messageID</m>
  <l>
    <a>value1</a>
    <a>value2</a>
    <a>...</a>
    <a>valuen</a>
  </l>
</u>
```

La lista completa de mensajes predefinidos puede encontrarse en [33]. Para la aplicación se usan los siguientes tipos de mensaje UPC:

Tabla 10 - Mensajes UPC Cliente-Servidor

Mensaje	Descripción
CREATE_ROOM	Este mensaje se usa para crear el canal donde se unirán los usuarios para recibir mensajes.
JOIN_ROOM	Se utiliza para unirse a un canal.
SEND_MESSAGE_TO_ROOMS	Se usa para enviar un mensaje al canal, que recibirán todos los usuarios del mismo.
SET_CLIENT_ATTR	Usado para establecer el valor del atributo de un usuario. En el caso de la aplicaión se emplea para definir el color y el grosor del lápiz de cada usuario.

Tabla 11 - Mensajes UPC Servidor-Cliente

Mensaje	Descripción
JOINED_ROOM	Informa al cliente si se ha unido al canal.
ROOM_OCCUPANTCOUNT_UPDATE	Devuelve el número de usuarios conectados al canal.
ROOM_SNAPSHOT	Devuelve el contenido de usuarios y sus atributos.
CLIENT_ATTR_UPDATE	Informa del cambio de un atributo de un usuario.
CLIENT_REMOVED_FROM_ROOM	Informa de la salida de un cliente del canal.

También es posible definir un formato de mensaje propio; en la aplicación se definen dos tipos de mensajes para controlar el movimiento y ruta de coordenadas de la pizarra. Para ello se realiza lo siguiente:

```
var Messages = {MOVE:"MOVE",
                PATH:"PATH"};
```

Una vez definidos los tipos de mensajes usados en la aplicación, se mostrarán algunos ejemplos de envío a nivel práctico. En el siguiente ejemplo se tiene la creación del canal "pizarra" y la posterior unión del usuario a dicho canal, tan solo con dos líneas de código.

```
var UPC = net.user1.orbiter.UPC;
var roomID = "pizarra";

msgManager.sendUPC(UPC.CREATE_ROOM, roomID);
msgManager.sendUPC(UPC.JOIN_ROOM, roomID);
```

Donde el parámetro pasado para crear el canal es el nombre del mismo. También podría haberse asignado una contraseña, para restringir el acceso a usuarios no deseados. En este caso no se le asigna ninguna.

Por último, para completar el esquema de paso de mensajes es necesario definir los diferentes Listeners y asociarlos a funciones JavaScript. Algunos ejemplos en la aplicación son los siguientes:

```
// registrar mensajes UPC del servidor
msgManager.addMessageListener(UPC.JOINED_ROOM,
                              joinedRoomListener,
                              this);
msgManager.addMessageListener(UPC.ROOM_OCCUPANTCOUNT_UPDATE,
                              roomOccupantCountUpdateListener,
                              this);
msgManager.addMessageListener(UPC.ROOM_SNAPSHOT,
                              roomSnapshotListener,
                              this);
msgManager.addMessageListener(UPC.CLIENT_ATTR_UPDATE,
                              clientAttributeUpdateListener,
                              this);
msgManager.addMessageListener(UPC.CLIENT_REMOVED_FROM_ROOM,
                              clientRemovedFromRoomListener,
                              this);

// registrar mensajes para la pizarra
msgManager.addMessageListener(Messages.MOVE,
                              moveMessageListener,
                              this,
                              [roomID]);
msgManager.addMessageListener(Messages.PATH,
                              pathMessageListener,
                              this,
                              [roomID]);
```

Así se tienen definidas todas las funciones que se ejecutarán cuando se reciba un mensaje de un determinado tipo, estas se localizan en el segundo parámetro de *addMessageListener*. Como se observa, en los dos últimos casos están los dos mensajes que se han creado para mover el cursor y establecer las rutas de líneas en la pizarra.

5.3.4 Ejemplo pizarra compartida

Haciendo uso de este Framework se va a realizar un pequeño ejemplo que consiste en una pizarra compartida por muchos usuarios.

Cada uno de ellos podrá pintar y borrar en la misma mientras los demás usuarios conectados pueden ver los cambios en tiempo real. Cada usuario podrá elegir el color del lápiz y su grosor, así como seleccionar la goma para borrar elementos.

Para ello, cada jugador dispondrá de variables donde se almacene el estado de los demás jugadores, es decir el color que está usando, su grosor del lápiz, las coordenadas por las que ha pasado su lápiz si ha dibujado algo, etc. Así se consigue un modelo en tiempo real.

Así, cada vez que un usuario realice alguna acción sobre la pizarra, las enviará al servidor para que sean enviadas a los demás jugadores.

La pizarra tendrá el siguiente aspecto:

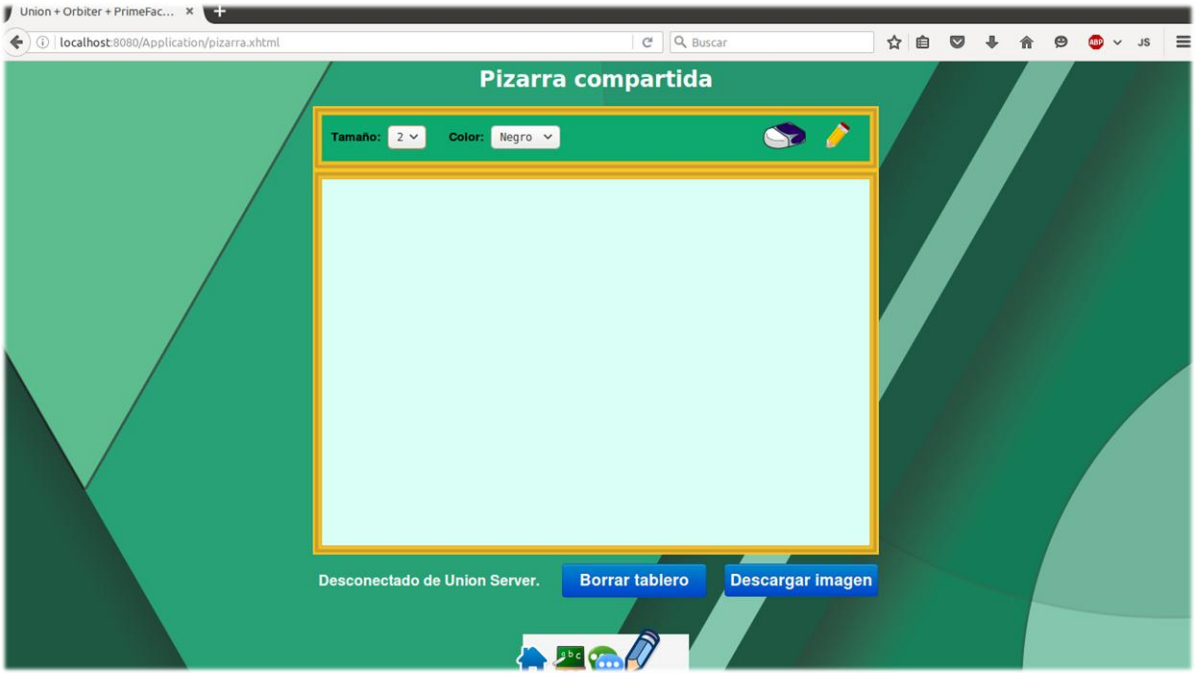


Figura 32 – Pizarra compartida

Donde el usuario dispone además de dos botones; el primero para borrar el tablero local y el segundo para descargar el dibujo en una imagen en formato PNG.

Para borrar la pizarra se usa la función en JavaScript *erase()*, que limpia la ventana canvas de HTML:

```
function erase() {
  context.clearRect(0, 0, canvas.width, canvas.height);
}
```

Para descargar la imagen se usa la función JavaScript *downloadCanvas()*, que usando un enlace se consigue mapear la pizarra a PNG:

```
function downloadCanvas(link, canvasId, filename) {
    link.href = document.getElementById(canvasId).toDataURL();
    link.download = filename;
}
```

También se dispone de información del número de usuarios conectados a la pizarra. Se puede observar en la línea de debajo de la pizarra:

Figura 33 – Información de usuarios conectados

En las siguientes imágenes se observan las distintas funcionalidades de la pizarra -elaboración de un dibujo con diferentes colores, uso de la goma y acciones de descargar la imagen y borrar el tablero-:

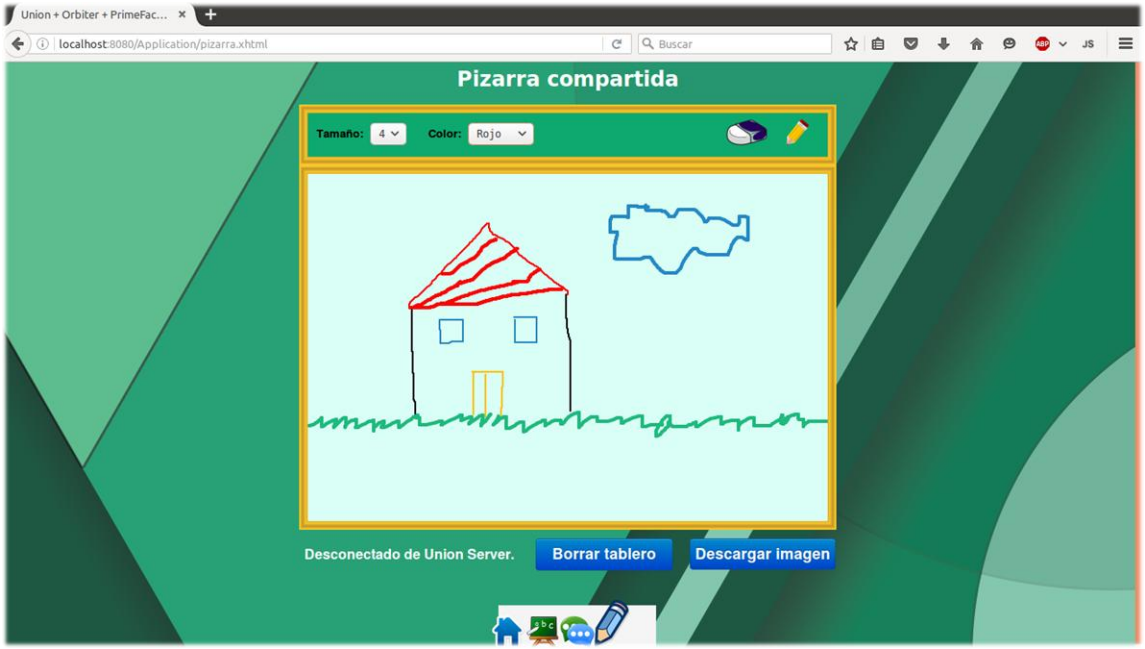


Figura 34 – Pizarra pintada

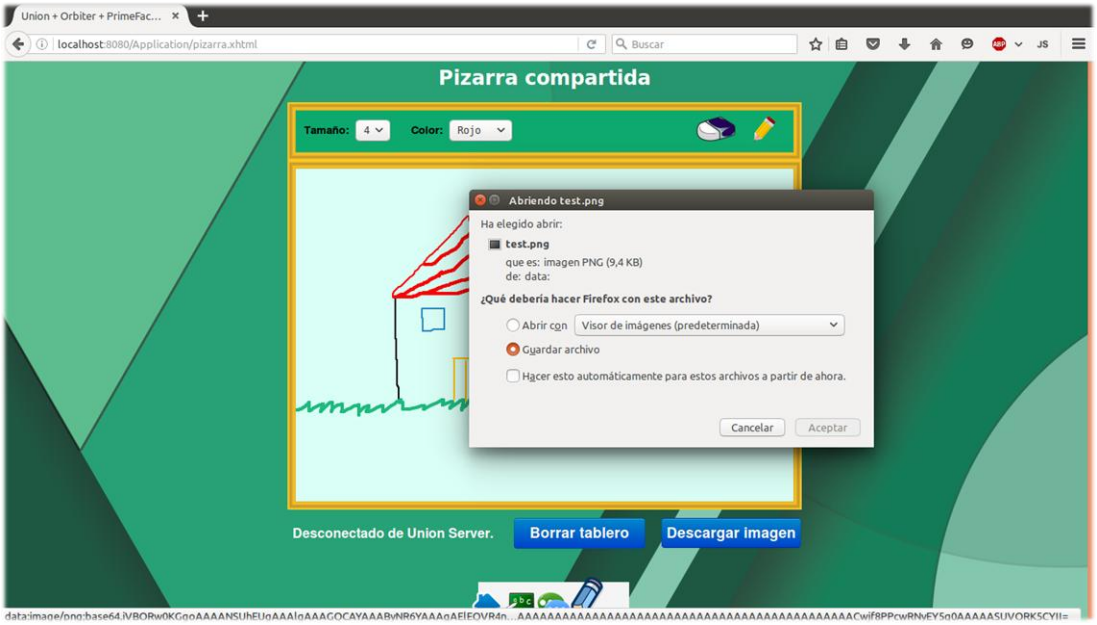


Figura 35 – Guardar imagen

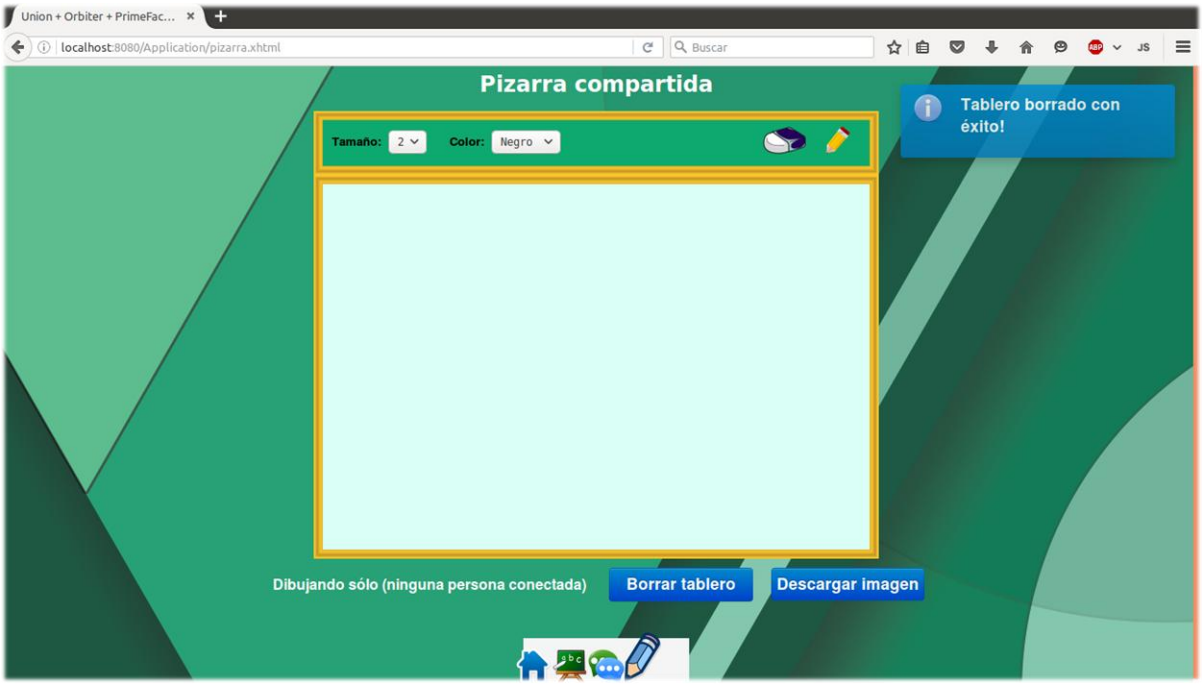


Figura 36 – Borrar tablero

Por último, se muestra el uso de la pizarra con dos usuarios conectados, viéndose cómo el dibujo es el mismo en tiempo real para los dos usuarios. En él, la palabra la ha escrito un usuario, mientras que la exclamación la ha pintado el otro.

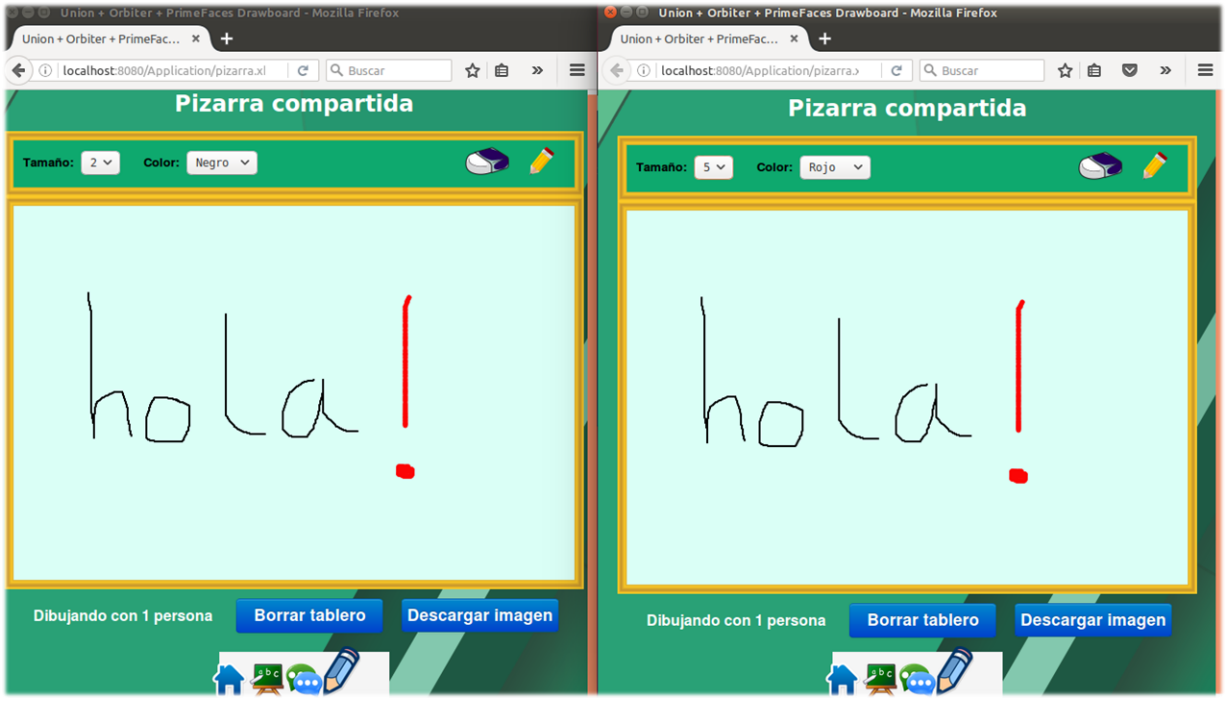


Figura 37 – Pizarra compartida por dos usuarios

5.4 Pictionary

En este apartado se explicará todo lo referente a la aplicación realizada, ya sea su estructura, tipos de mensajes utilizados, funcionamiento, etc.

Como ya se ha comentado antes, la aplicación consiste en un Pictionary, en el que un usuario se encarga de pintar un dibujo seleccionado aleatoriamente para que el resto de jugadores que componen la partida traten de adivinarlo.

La pizarra se controlará con el Framework Orbiter y el juego (turnos, mensajes, etc) con el Framework Atmosphere.

5.4.1 Estructura de la aplicación

La estructura de la aplicación sigue el modelo de un proyecto Maven, con su correspondiente fichero de configuración *pom.xml*. Siguiendo este esquema, se define de la siguiente forma:

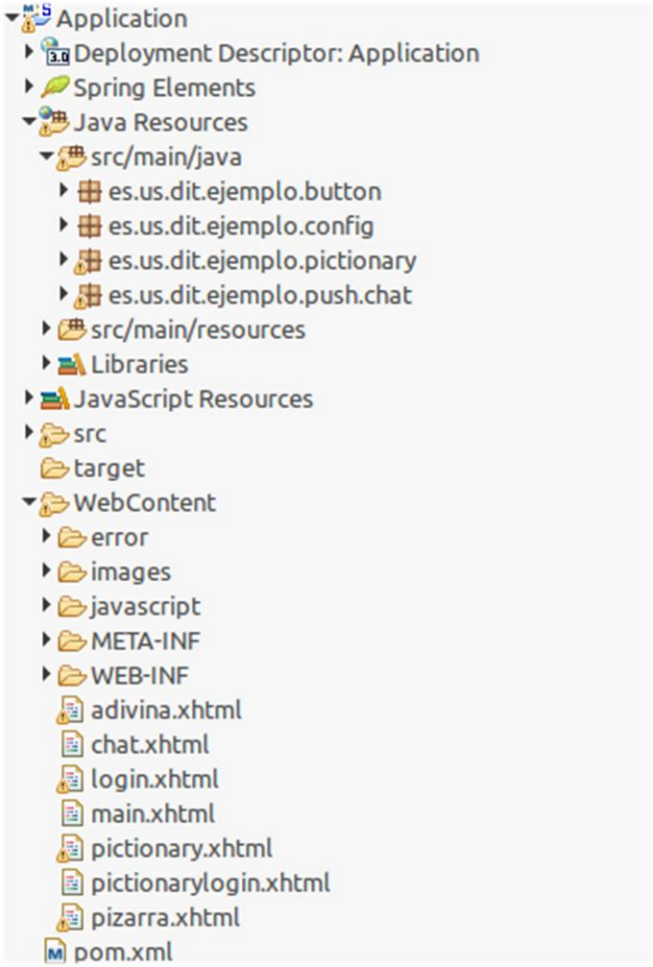


Figura 38 – Estructura del proyecto

Donde existen tres directorios principales:

- **src**: que será la contenedora de los ficheros *.java*.
- **target**: donde aparecerán los ficheros *.class*, una vez compilado el proyecto.
- **WebContent**: donde se localizarán todos los elementos necesarios para las páginas web, ya sean plantillas HTML, ficheros JavaScript, ficheros XML de configuración, imágenes, etc.

En los siguientes subapartados se detallarán algunos aspectos importantes de la estructura utilizada para la aplicación.

5.4.1.1 Ficheros Java

Para el desarrollo del Pictionary se han empleado los siguientes ficheros Java:

Figura 39 – Estructura paquetes Java

La descripción de cada uno se detalla a continuación:

- *Message.java*: Contendrá el formato de mensaje que usará la aplicación. En el apartado 5.4.3 se explica con mayor detalle todos y cada uno de estos aspectos.
- M*essaDecoder*.java y *MessageEncoder.java:* Se tratan de los codificadores y decodificadores de los mensajes. En el apartado 5.2.2.1, sección @OnMessage se explican todos los conceptos relevantes.
- *PictionaryResource.java*: En él se define el recurso PushEndpoint de cada jugador.
- *PictionaryUsers.java* y *PictionaryScores.java*: se tratan de los ManagedBeans de la aplicación, donde se almacenarán los usuarios y las puntuaciones del juego.
- *PictionaryView.java*: es la clase que se encarga de interactuar con el navegador, la que contiene los métodos que son llamados desde la página.

5.4.1.2 Ficheros de configuración

Otro aspecto destacable de la estructura del proyecto son los ficheros de configuración. En la carpeta WEB-INF del proyecto se encuentran:

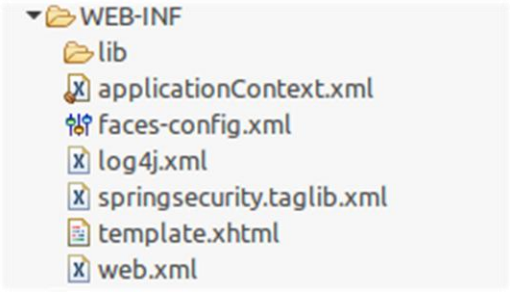


Figura 40 – Ficheros de configuración del proyecto

Donde se destacan los ficheros *applicationContext.xml, web.xml* y *faces-config.xml*, que contienen elementos de configuración del proyecto JSF, páginas de bienvenida y configuración del proyecto en general.

Por otro lado, el fichero *log4j.xml* se usa para la configuración del logger de la aplicación, muy útil para la depuración de la misma.

Por último, *springsecurity.taglib.xml* es el fichero para la configuración de seguridad de la aplicación, siguiendo un modelo que propone Spring.

5.4.1.3 Ficheros XHTML

Para el diseño de las páginas web del pictionary se usarán plantillas XHTML. Existen tres plantillas principales que forman la aplicación:

- *pictionarylogin.xhtml*: donde se realiza la ingresión a la aplicación.
- *pictionary.xhtml*: es la página donde se proporciona la interfaz para pintar. Solo habrá un único jugador pintando en la partida.
- *adivina.xhtml*: es la página donde se proporciona la interfaz para adivinar el dibujo. Pueden haber muchos jugadores adivinando en la partida.

En las siguientes imágenes se muestran cada una de ellas:

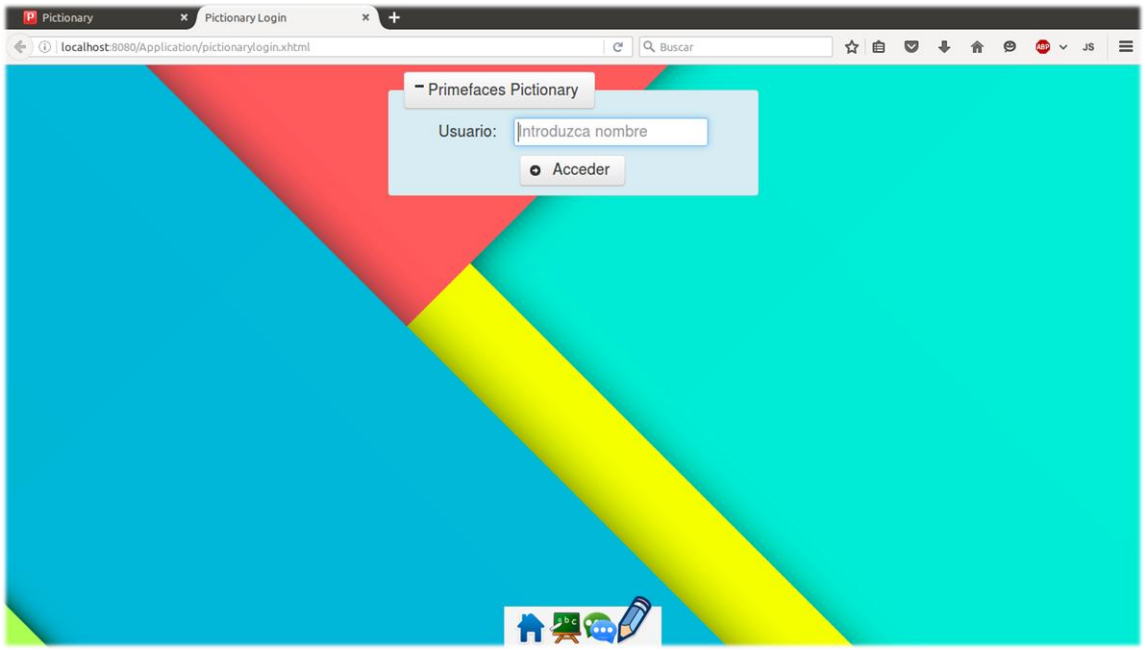


Figura 41 – Pictionary Login (pictionarylogin.xhtml)

Figura 42 – Pictionary pinta (pictionary.xhtml)

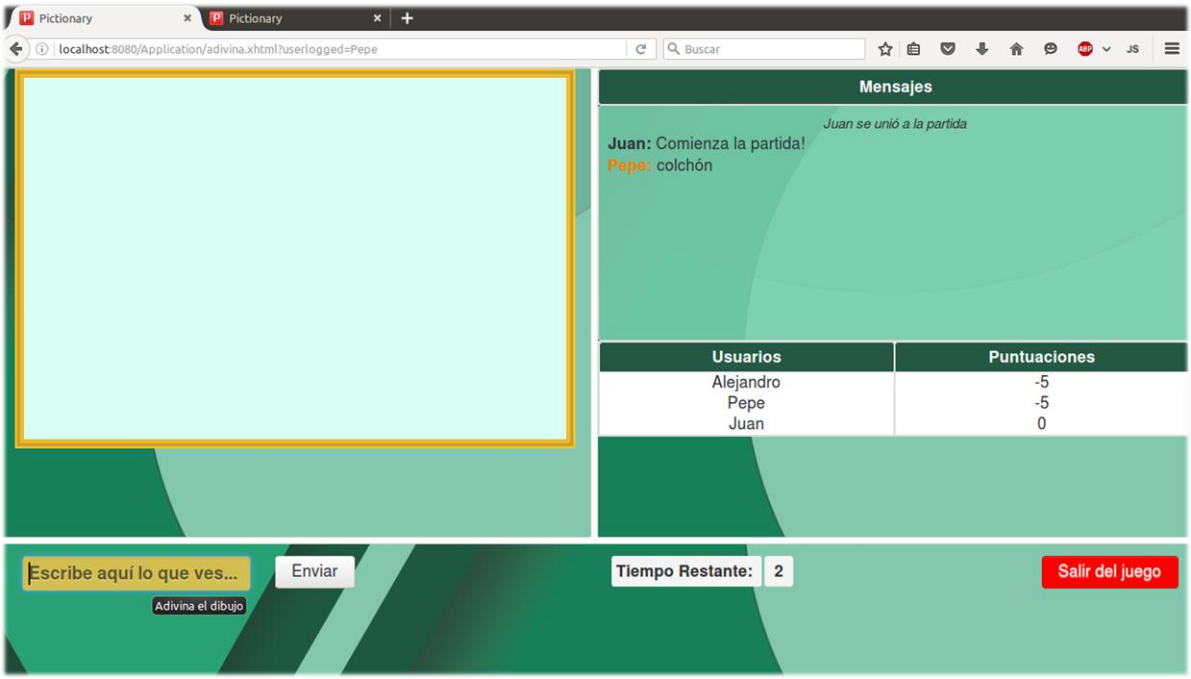


Figura 43 – Pictionary adivina (adivina.xhtml)

5.4.2 Funcionamiento de la aplicación

El funcionamiento básico de la aplicación es el siguiente:

1. El usuario ingresa en la página *pictionarylogin.xhtml*.
2. Si no hay nadie en la partida, el usuario pasa a la interfaz de pintar (*pictionary.xhtml*). Si ya hay alguien en la partida, el usuario pasa a adivinar (*adivina.xhtml*).
3. Una vez que haya al menos dos jugadores, el usuario que pinta puede iniciar la partida. Permitiendo a los demás usuarios adivinar su dibujo.
4. Si un usuario adivina el dibujo, termina el turno, y se indica a todos los jugadores que ha ganado. Éste usuario pasará a pintar en el siguiente turno, mientras que el usuario que pinta pasará a adivinar.
5. Si en un turno no se adivina el dibujo y expira el temporizador, el usuario que pinta pasará a adivinar y otro usuario (el siguiente de la lista) será el que pinte en el próximo turno.
6. Si un usuario ingresa en una partida empezada, se queda a la espera de que comience la siguiente.
7. Si el usuario que pinta se desconecta de la aplicación, se notifica a todos los usuarios, permitiendo que otro usuario (el siguiente de la lista) pase a pintar.
8. Si un usuario que adivina se desconecta de la aplicación, simplemente se elimina de la lista y no se interrumpe la partida actual.
9. PUNTUACIONES:
    - Cada usuario ingresa en la aplicación con 0 puntos.
    - Si un usuario adivina la palabra dentro del tiempo, suma 10 puntos. También suma 5 puntos el usuario que pinta.
    - Si nadie adivina la palabra dentro del tiempo, el usuario que pinta pierde 5 puntos.

En el apartado 5.4.6 se muestra de forma gráfica cómo sería una partida, con todos los mensajes y diálogos de juego.

5.4.3 Mensajes de la aplicación

5.4.3.1 Message.java

Como se ha comentado anteriormente, la clase *Message.java* contiene los atributos para los diferentes mensajes que usará la aplicación. Su estructura es la siguiente:

```
public class Message {

    private String text;
    private String user;
    private boolean updateList;
    private String color;

    // only for initial messages
    private String keyword;
    private boolean startMessage;

    // only for end messages
    private boolean endMessage;

    // disconnect message
    private boolean disconnect;

    // restart message
    private boolean restart;

    // change painter, no winners
    private boolean changePainter;

    // wait for next game, no winners (general)
    private boolean waitNextGameNoWinners;
...
```

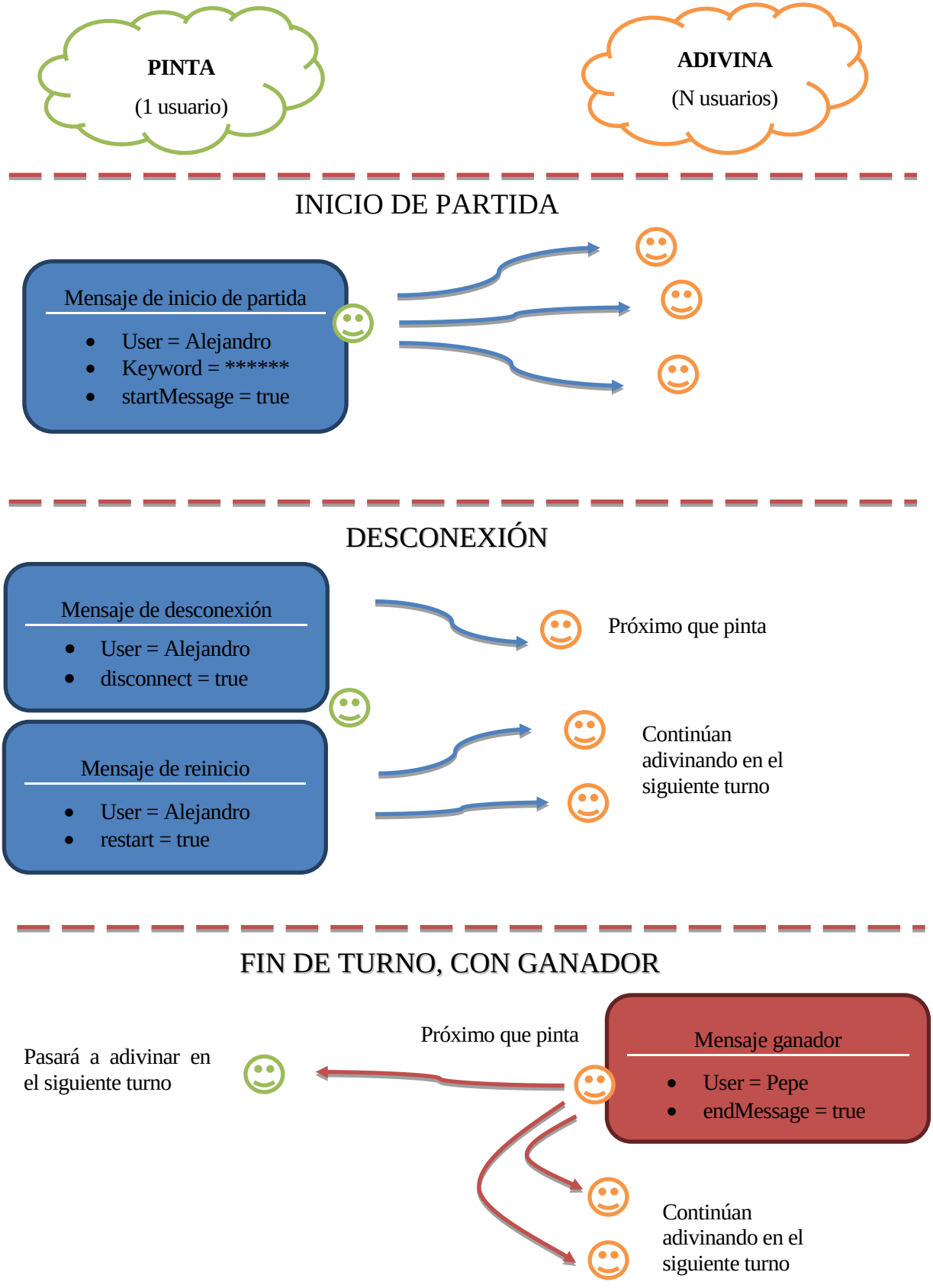
Donde cada atributo se corresponde con lo siguiente:

- **text:** será el texto que envía cada usuario, en el caso del pictionary será la palabra indicando lo que es el dibujo.
- **user:** es el usuario que manda el mensaje.
- **updateList:** si vale true, significa que hay que actualizar la lista de usuarios y puntuaciones.
- **color:** es el color correspondiente a cada usuario; se usará en el cuadro de mensajes para colorear el nombre de usuario.
- **keyword:** contiene la palabra secreta, es decir el contenido del dibujo. Sólo se envía en el mensaje inicial; desde el usuario que pinta al resto.
- **startMessage:** indica si es el mensaje de comienzo de partida y, además, contendrá la palabra secreta.
- **endMessage:** indica el final de una partida y comunica que un usuario ha ganado.
- **disconnect:** indica la desconexión del usuario que pinta. Comienza, por tanto, un proceso de reinicio de la partida.
- **restart:** indica el reinicio de una partida tras una desconexión.
- **changePainter:** indica el final de una partida; el que lo recibe será el próximo usuario que pinte.
- **waitNextGameNoWinners:** indica el final de una partida; comunica que nadie ha ganado y pone en espera de comienzo de nueva partida al jugador que lo recibe.

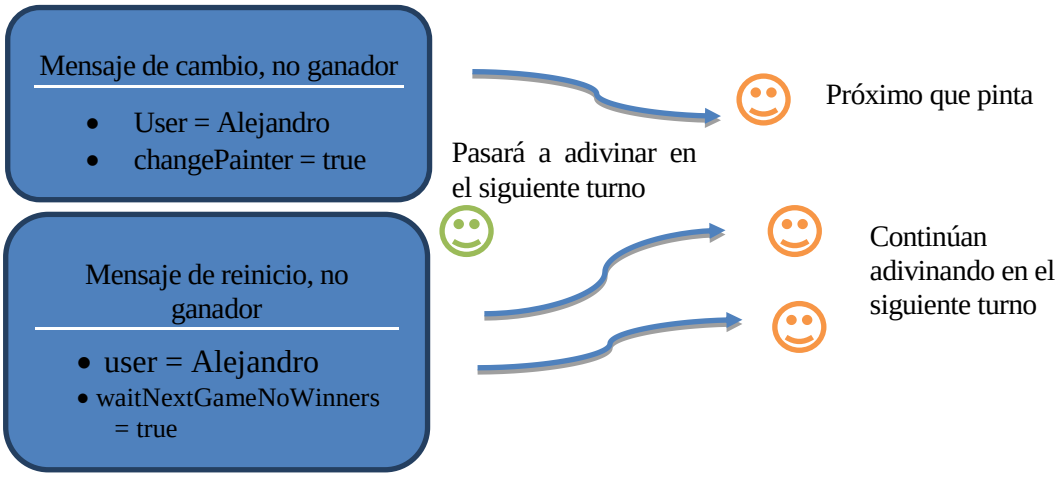
Así, simplemente, basta crear el mensaje con los atributos deseados y enviarlo a uno o varios usuarios. Se permite la inclusión de varios parámetros en el mismo mensaje. Un ejemplo sería un único mensaje para indicar fin de partida (*endMessage*) y actualizar las puntuaciones (*updateList*).

5.4.3.2 Resumen de mensajes

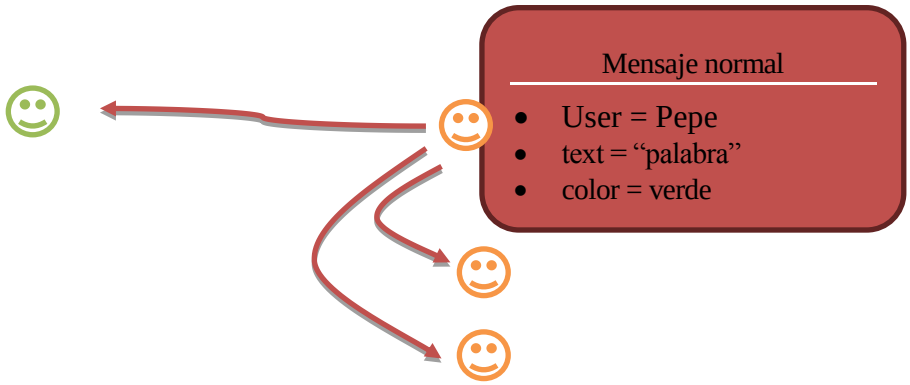
En el siguiente esquema se muestran los mensajes para organizar el juego de la aplicación:



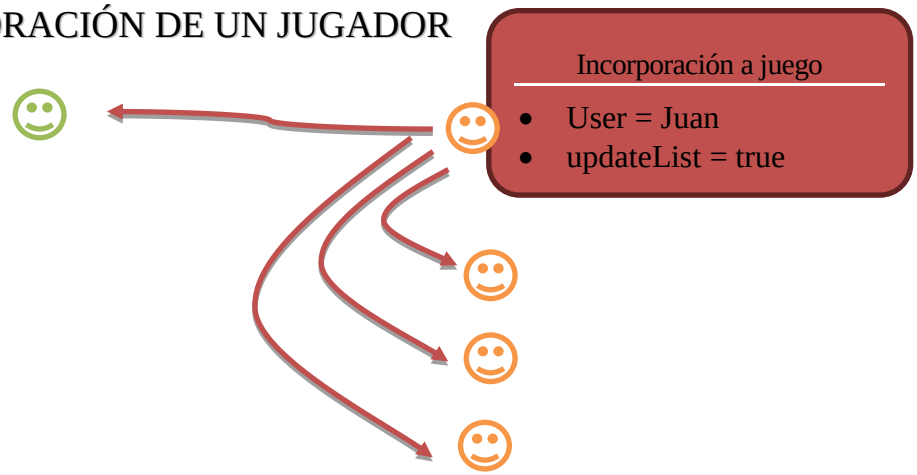
FIN DE TURNO, SIN GANADOR



ENVÍO DE MENSAJE NORMAL



INCORPORACIÓN DE UN JUGADOR



5.4.3.3 Procesamiento con JavaScript

Cabe destacar que con el uso de Atmosphere, cuando se recibe un mensaje hay que apoyarse en JavaScript para procesarlo. Tal y como se comentó en apartados anteriores, en el elemento socket se define la función JavaScript que manejará el mensaje, de la siguiente manera:

```
<p:socket onMessage="handleMessage"
          channel="/pictionaryroom"
          autoConnect="false"
          widgetVar='subscriber2' />
```

Por lo que en la página se debe incluir dicha función; un ejemplo sería:

```
function handleMessage(message) {

  if (message.endMessage) {
    ...
  } else {
    // imprime el texto en el cuadro de mensaje
    ...
    // Accedo al mensaje con message.user, message.text,...
  }
  if (message.updateList) {
    updateList();
  }
}
```

Donde se posibilita el acceso a los campos del mensaje recibido simplemente indicando *message.ATRIBUTO*. Así se puede organizar el juego con el paso de mensajes.

Por otro lado, si se desea llamar a una función de los Beans desde JavaScript, hay que añadir en la página la etiqueta *remoteCommand*; un ejemplo es el siguiente:

```
<p:remoteCommand name="socket_connection"
                 process="@this"
                 actionListener="#{pictionaryView.login2}"/>
```

En este caso desde JavaScript se llama a la función *socket_connection()*, para que desde la página se invoque al método *login2()* del Bean *PictionaryView*. Como se ve, todo de forma muy fácil.

Así se termina de definir la estructura de mensajes de la aplicación.

5.4.4 Diálogos de juego

Otro aspecto a destacar de la aplicación es el uso de dialogos para bloquear la partida cuando sea necesario. Para ello se hace uso del componente *dialog* de PrimeFaces. Un ejemplo de su uso es el siguiente:

```
<!-- Initial dialog -->
<p:dialog header="Esperando inicio de nueva partida..."
          widgetVar="dialogStart" visible="true"
          resizable="false" closable="false" modal="true">

   <p:graphicImage value="images/loading.gif" />
   <p:commandButton value="Salir del juego"
                    actionListener="#{pictionaryView.disconnect}"
                    update="users scores"/>
</p:dialog>
```

Así se obtiene una salida como esta:

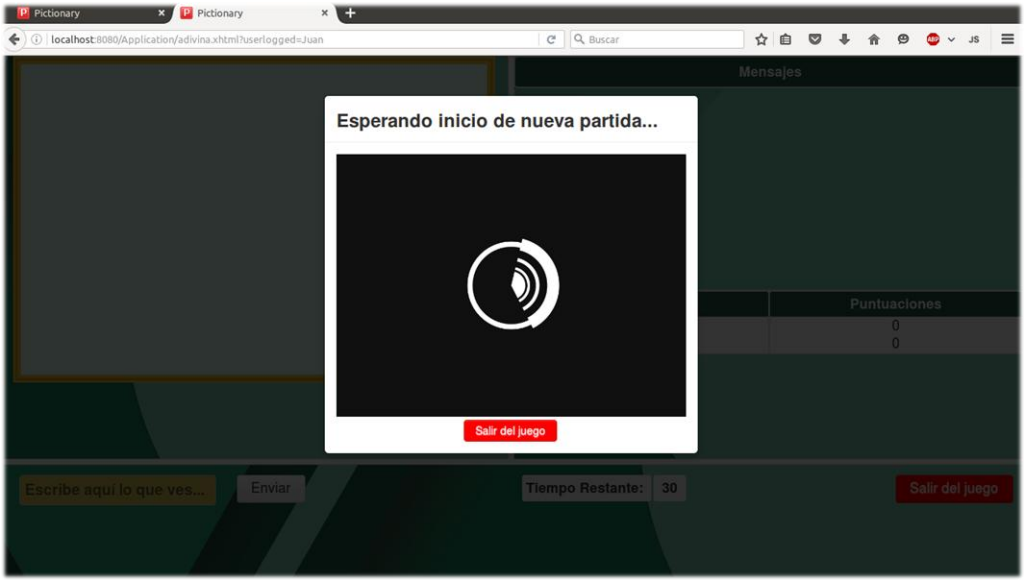


Figura 44 – Dialog de espera

Como se observa, al ser un *dialog* de tipo modal, el fondo se oscurece no permitiendo al usuario acceder al juego. Este *dialog* se ocultará cuando el jugador reciba el mensaje "comienzo de partida".

Para mostrar y ocultar *dialogs*, basta con llamar a las funciones JavaScript *show()* y *hide()*, respectivamente. Para ello accedemos al *dialog* deseado, gracias a su *widgetVar*:

```
PF('dialogStart').hide(); // ocultar dialog
PF('dialogStart').show(); // mostrar dialog
```

El resto de *dialogs* del juego funcionan de la misma manera. En las siguientes imágenes se muestran todos:

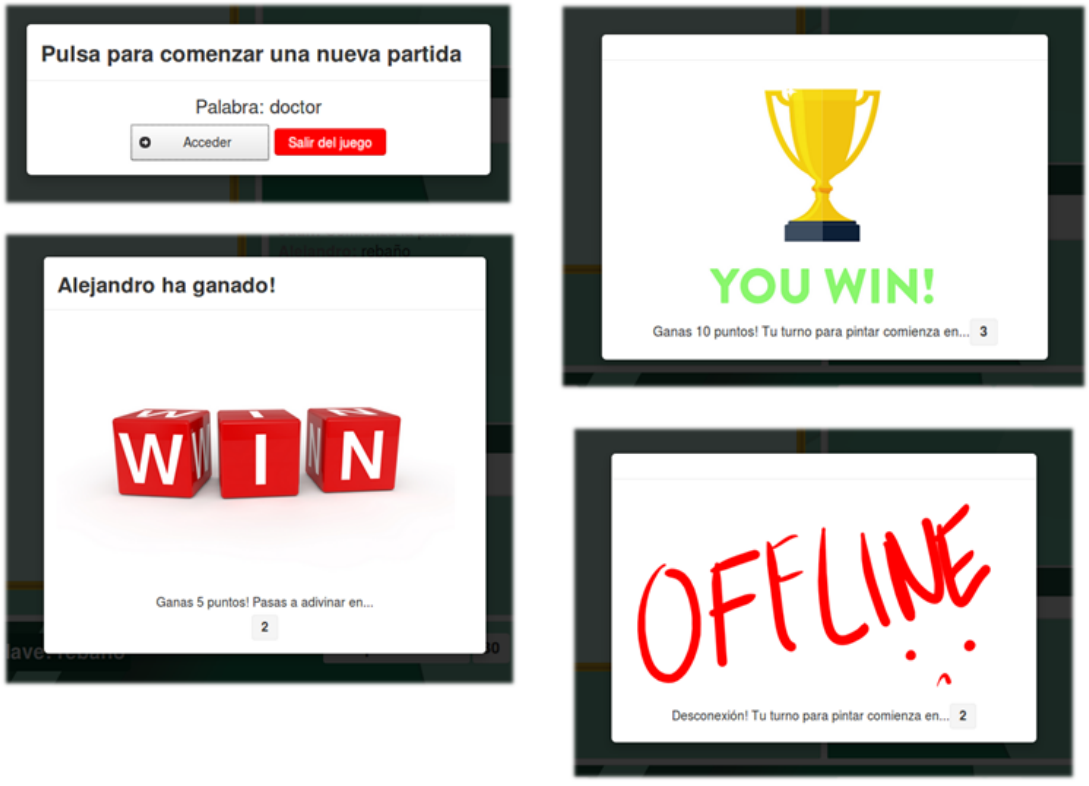


Figura 45 – Dialogs I

Figura 46 – Dialogs II

5.4.5 Temporizadores

Otro de los aspectos interesantes de la aplicación son los temporizadores, ya que gracias a ellos se consigue dar por finalizado un turno o establecer una cuenta atrás para una redirección de página, etc.

Para la aplicación se ha usado el componente *timer*, proporcionado con las extensiones de PrimeFaces [34]. Su manejo es muy sencillo, basta con declarar en la página la siguiente etiqueta:

```
<pe:timer widgetVar="timerGame"
          timeout="30"
          singleRun="true"
          autoStart="false"
          listener="#{pictionaryView.noWinners}"/>
```

Donde se le asignan los atributos necesarios. En este caso tiene la referencia para su acceso *timerGame*, una duración de 30 segundos, desactivado el inicio automático y activado en modo cuenta única (no se reinicia al terminar). Por último, cuando el temporizador vence se invoca al método especificado en el *listener*.

El aspecto del temporizador en el juego es el siguiente:

Figura 47 - Temporizador

En este caso se invoca al método *noWinners* del Bean *pictionaryView*, que iniciará un proceso de cambio de turno y cambio de jugador que pinta.

Para el manejo de temporizadores se usan las funciones JavaScript *start()*, *pause()*, *restart()* y *stop()*. Para ello, accedemos al temporizador de la siguiente forma:

```
PF('timerGame').start(); // inicia el temporizador
PF('timerGame').stop();  // para el temporizador (reiniciando cuenta)
PF('timerGame').restart(); // reinicia el temporizador
PF('timerGame').pause();  // pausa el temporizador
```

Así se pueden controlar los temporizadores al recibir mensajes de control de juego.

5.4.6 Demostración de juego

En este apartado se mostrará el funcionamiento de la aplicación de forma visual, capturando imágenes de cada fase del juego.

Para esta demostración, se inicia una partida con tres jugadores. Estos, en primer lugar, ingresan en la aplicación con su nombre:

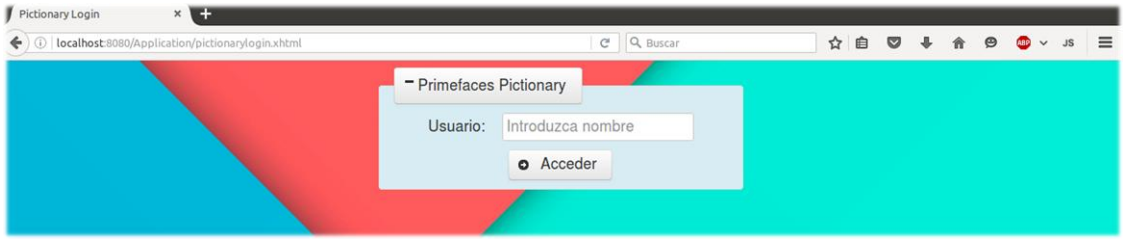


Figura 48 – Pictionary Login

Una vez realizado el ingreso, al primer jugador le aparecerá el siguiente mensaje:

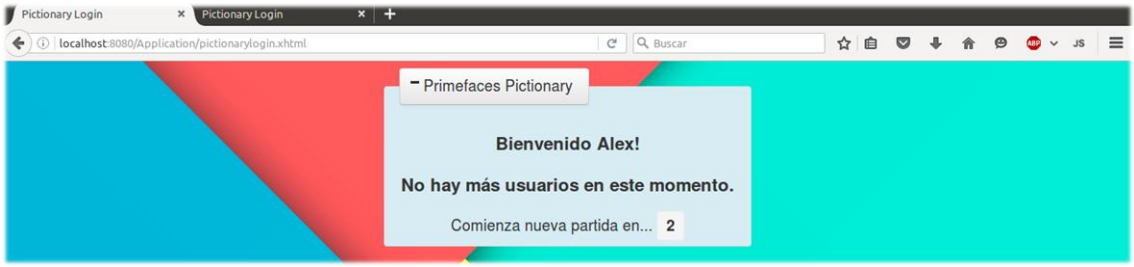


Figura 49 – Mensaje de bienvenida I

Mientras que al resto le saldrá un mensaje de los usuarios que hay conectados para posteriormente unirse a la partida:

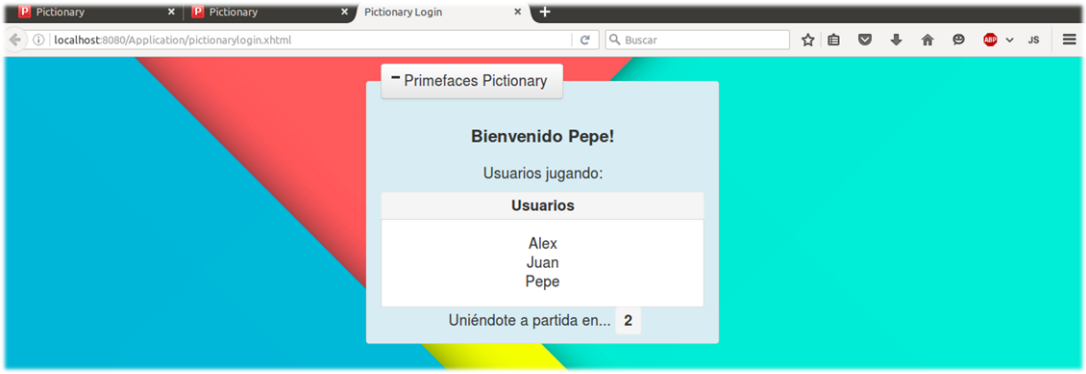


Figura 50 – Mensaje de bienvenida II

Al usuario que pinta le saldrá el siguiente *dialog* con una palabra aleatoria que tendrá que pintar. Para ello se elegirá de una lista de palabras alojadas en el servidor:

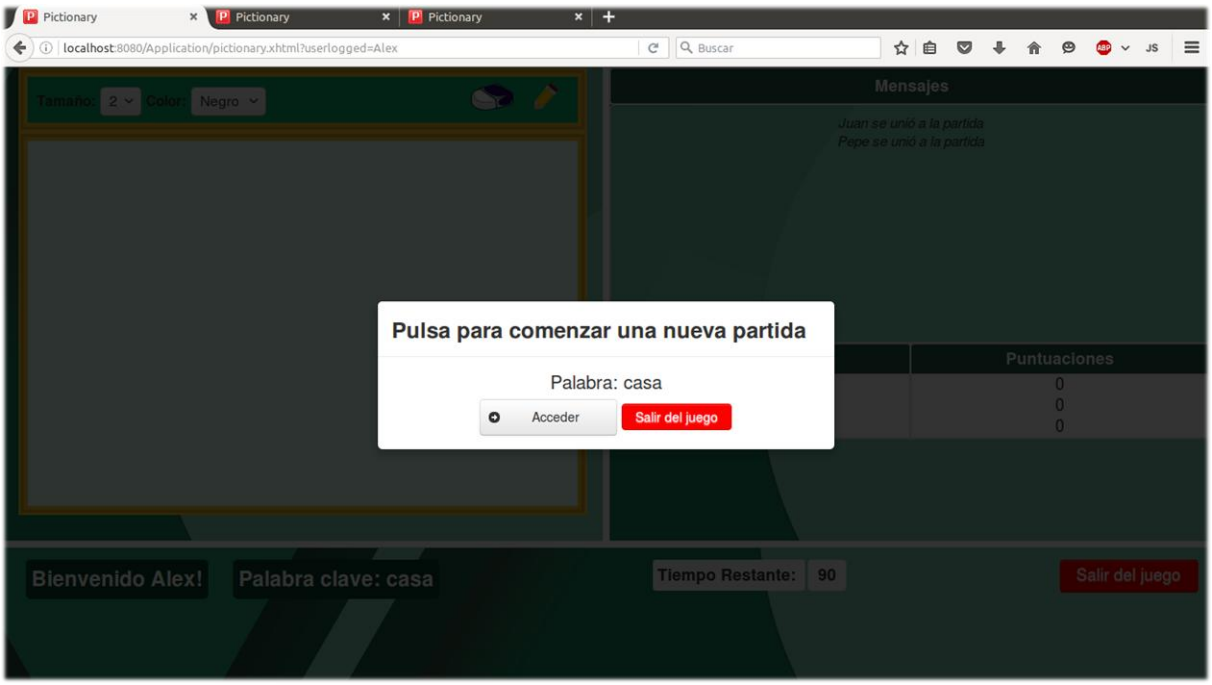


Figura 51 – Inicio de partida

El resto de jugadores tendrá que esperar el inicio de la partida:

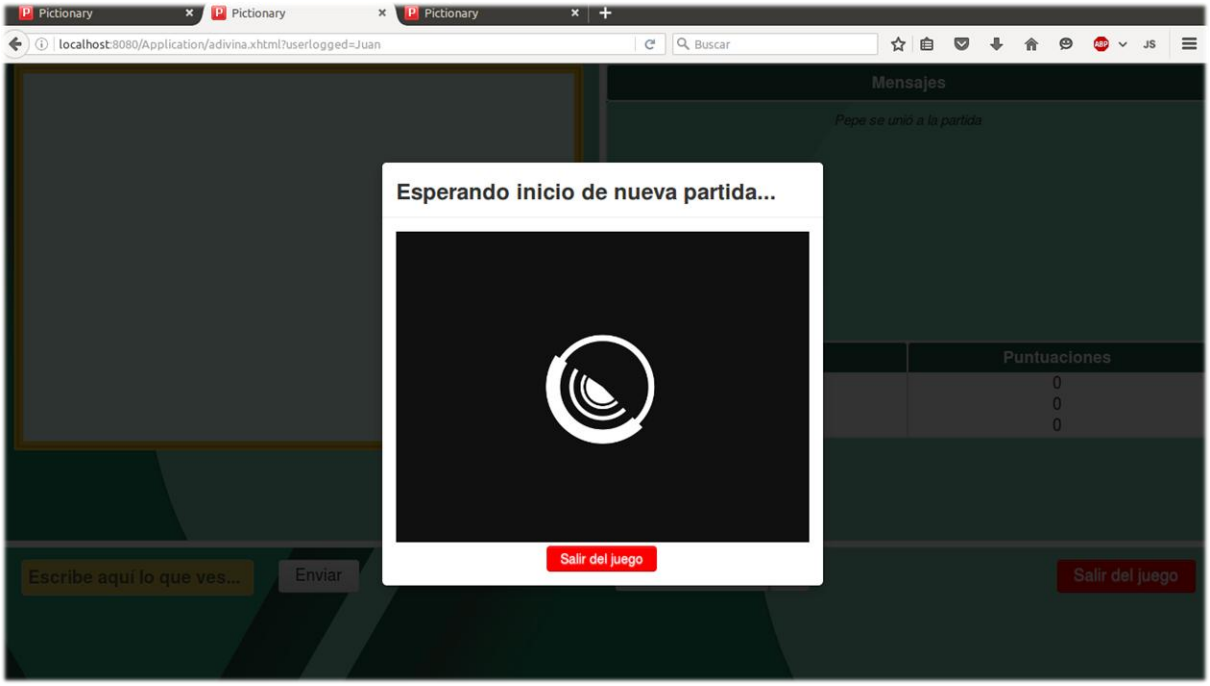


Figura 52 – Espera de inicio

Cuando el jugador que pinta comience la partida, los *dialogs* se ocultan para empezar el juego.

La interfaz para pintar se abre para el primer jugador, y la interfaz de adivinar se abre para el resto de jugadores:

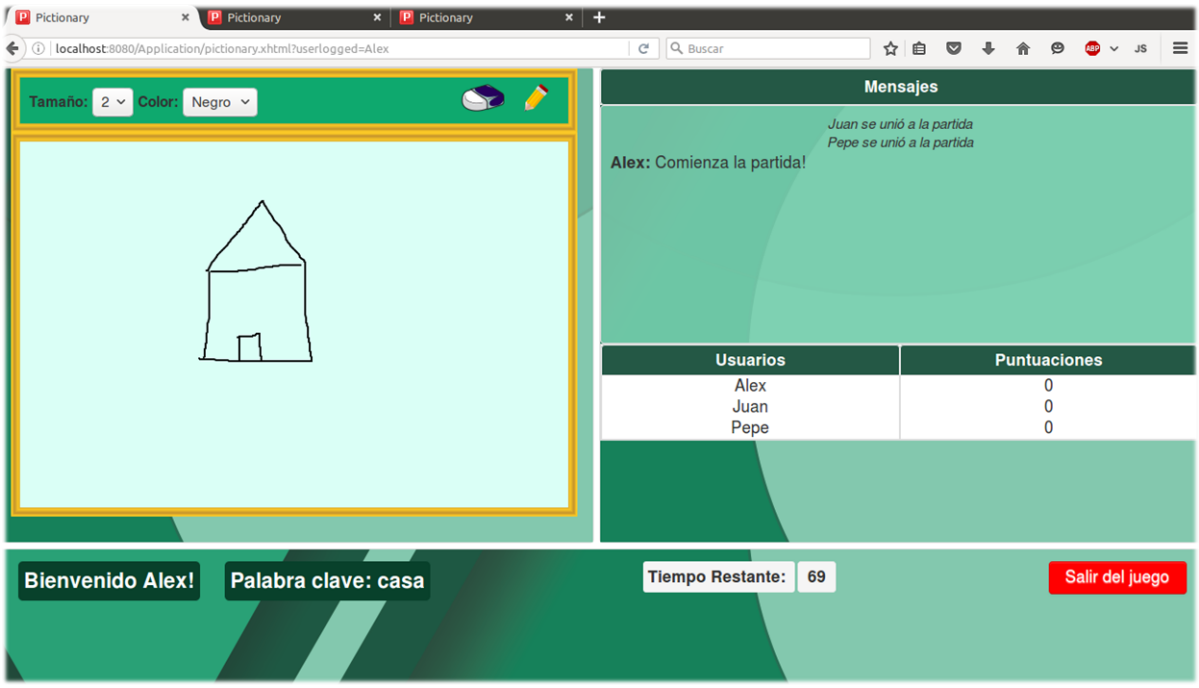


Figura 53 – Interfaz para pintar

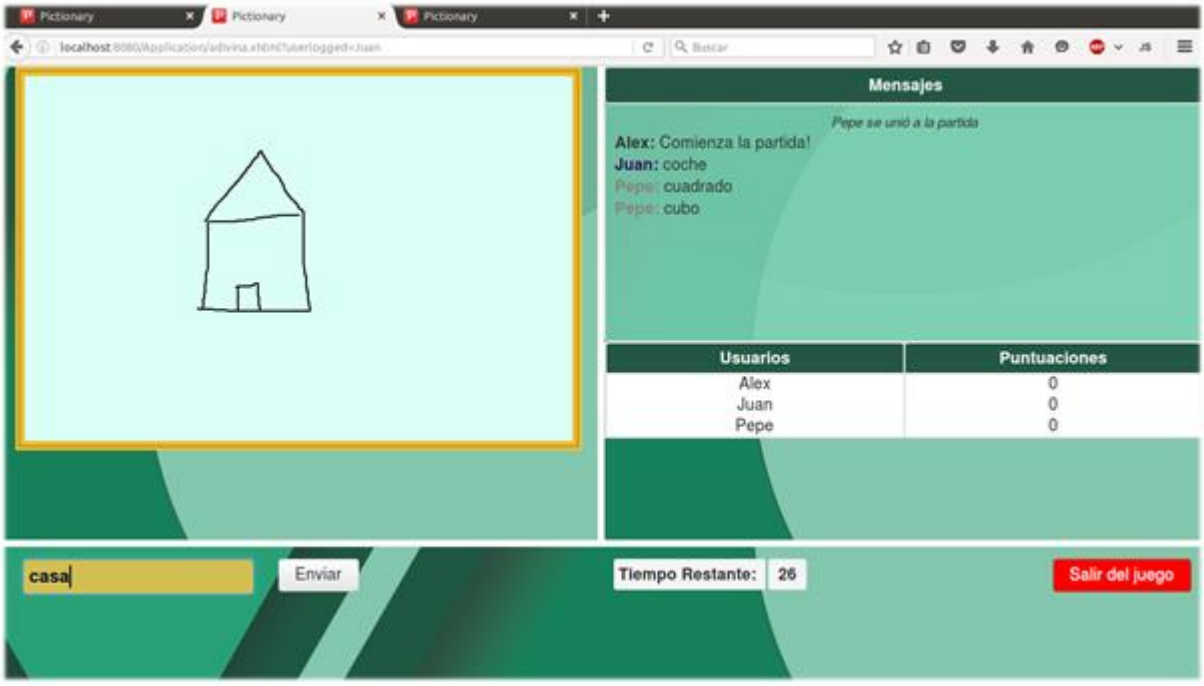


Figura 54 – Interfaz para adivinar

Se dispone de un tiempo de 90 segundos para adivinar el dibujo. Los usuarios que adivinan tendrán que introducir la palabra en el cuadro de texto. Todas las palabras irán saliendo en el cuadro de mensajes de todos los usuarios, incluyendo al que pinta.

El primer usuario que inserte la palabra correcta ganará y pasará a pintar en el siguiente turno. Le aparecerá el siguiente mensaje:

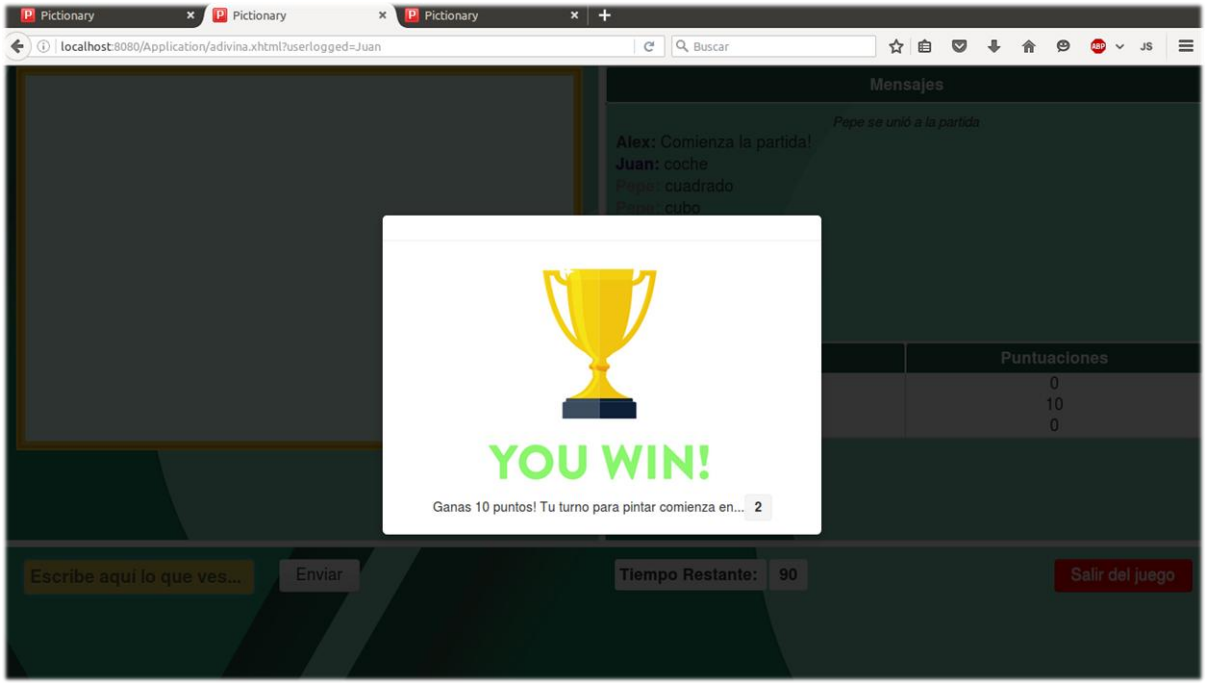


Figura 55 – Mensaje ganador

Al resto de jugadores les aparecerá un mensaje indicando quién es el ganador, para ello, cuando el jugador envíe un mensaje, el servidor comprueba si es la palabra secreta; en caso afirmativo, terminará la partida actual y se iniciará la espera del siguiente turno:

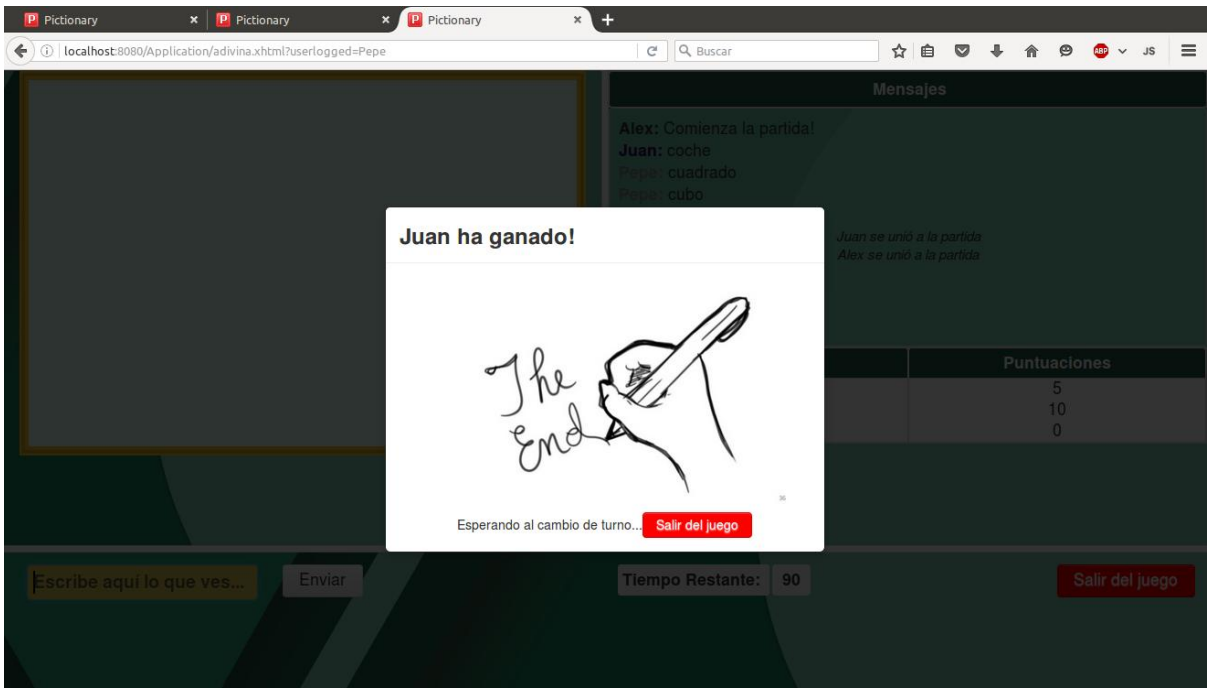


Figura 56 – Espera de siguiente turno

Por último, si el usuario que pinta se desconecta en medio de la partida, se envía un mensaje de desconexión a los jugadores y se inicia un proceso de cambio de jugador que pinta.

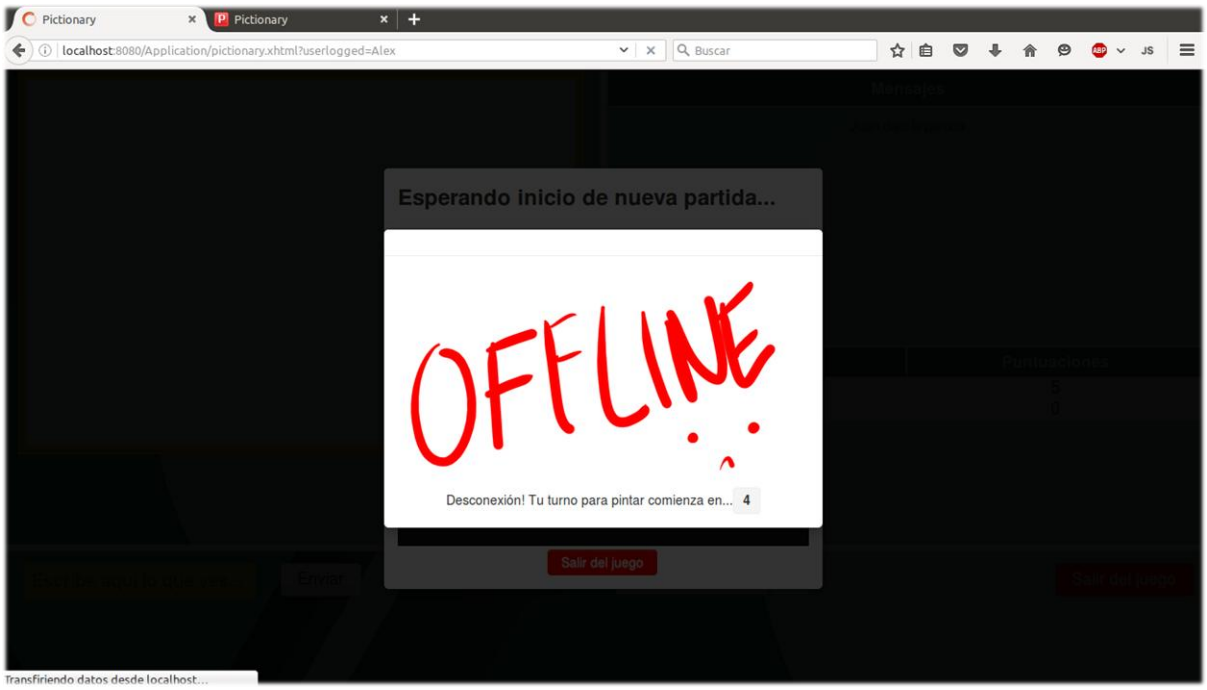


Figura 57 - Desconexión

Destacando que si un usuario sale de la partida perderá los puntos que lleve hasta el momento.

6 CONCLUSIONES Y TRABAJOS FUTUROS

El verdadero progreso es el que pone la tecnología
al alcance de todos.

Henry Ford

6.1 Conclusiones

La realización del proyecto me ha servido para entender los modelos de tecnologías más actuales para la comunicación entre cliente y servidor, desarrollo de aplicaciones web interactivas y manejo de tecnologías y Frameworks actuales. Destaco la utilización de WebSockets; ya que forman una herramienta muy potente para el desarrollo de aplicaciones web en tiempo real.

Otro de los puntos destacables de la investigación realizada es la expansión de HTTP/2, ya que proporcionará una experiencia de usuario mejorada en todos los aspectos. Sin embargo, hasta que su despliegue sea importante tendrán que pasar unos años, ya que existen muchas implementaciones de su versión anterior HTTP/1.1, por lo que aún habrá que esperar para observar su despliegue total -¡quién sabe si será HTTP/3 el que cause un verdadero impacto!-.

Además, otro de los puntos que me ha hecho aprender como funciona el desarrollo de una aplicación es el uso de Frameworks, ya que reducen el trabajo y facilitan la implementación notablemente; en el caso de Orbiter y Atmosphere facilitan la introducción de WebSockets a un proyecto.

Por otro lado, he experimentado el uso del software libre, que es el futuro, ya que cada vez se disponen de más herramientas gratuitas para realizar cualquier proyecto.

También he aprendido a manejar aplicaciones web usando Java, concretamente JSF; ya que antes del proyecto no tenía claro como hacerlo, por lo que me ha aportado mucho en este aspecto. Así pues, concluyo que usando este modelo se pueden realizar aplicaciones que soporten un gran número de usuarios.

En resumen, mediante la realización de este proyecto he asimilado la forma de realizar una investigación, para posteriormente ponerla en práctica al realizar la aplicación.

6.2 Trabajos futuros

Tras la realización del proyecto se proponen algunas mejoras de la aplicación, propuestas para futuros trabajos. Se destacan las siguientes:

- Creación de diferentes salas de juego, para que un usuario pueda elegir en que sala entrar.
- Creación de un sistema de usuarios global, donde el usuario se registre en el sistema con unas credenciales únicas. Así cada usuario puede tener sus historiales de puntuaciones y/o puntos semanales, mensuales…
- Creación de partidas donde puedan pintar más de un usuario a la vez (compartir el dibujo).
- Sistema de audio para los usuarios que adivinan, para que puedan comunicarse entre ellos.
- Posibilidad de crear equipos de juego, para que los puntos sean en grupo.
- Diferentes modos de juego, cambiando el tiempo para adivinar.
- Creación de partidas limitadas por un número de usuarios. Ejemplo: partidas de 4 personas.

Por otro lado, se propone cambiar la estructura de la aplicación y realizarla toda en una única página, así no habría que realizar redirecciones para pintar o adivinar. Además del uso de otros Frameworks para el manejo de WebSockets, como NodeJS y Socket.io.

REFERENCIAS

[1] J. Klensin, «Simple Mail Transfer Protocol,» [En línea]. Available: https://tools.ietf.org/html/rfc5321. [Último acceso: 30 08 2016].

[2] J. Cancela, «Notificaciones 'push' en dispositivos móviles,» [En línea]. Available: https://javiercancela.com/2009/07/13/notificaciones-push-en-dispositivos-moviles/. [Último acceso: 10 7 2016].

[3] PubNub, «What is HTTP Long Polling?,» [En línea]. Available: https://www.pubnub.com/blog/2014-12-01-http-long-polling/. [Último acceso: 18 7 2016].

[4] G. Code, «Polling as an alternative,» [En línea]. Available: https://code.google.com/archive/p/google-web-toolkit-incubator/wikis/ServerPushFAQ.wiki. [Último acceso: 18 7 2016].

[5] D. Sheiko, «WebSockets vs Server-Sent Events vs Long-polling,» [En línea]. Available: http://dsheiko.com/weblog/websockets-vs-sse-vs-long-polling/. [Último acceso: 18 7 2016].

[6] IBM, «Introduction to Comet,» [En línea]. Available: http://www.ibm.com/developerworks/library/wa-reverseajax1/. [Último acceso: 18 7 2016].

[7] pushlets, «pushlets,» [En línea]. Available: http://www.pushlets.com. [Último acceso: 18 7 2016].

[8] lightstreamer, «ligthstreamer,» [En línea]. Available: http://www.lightstreamer.com. [Último acceso: 18 7 2016].

[9] webreference, «Comet Programming: Using Ajax to Simulate Server Push,» [En línea]. Available: http://www.webreference.com/programming/javascript/rg28/index.html. [Último acceso: 18 7 2016].

[10] W3C, «XMLHttpRequest,» [En línea]. Available: https://www.w3.org/TR/XMLHttpRequest/. [Último acceso: 18 7 2016].

[11] W3C, «Server-Sent Events,» [En línea]. Available: https://www.w3.org/TR/eventsource/. [Último acceso: 18 7 2016].

[12] Opera, «Event Streaming to Web Browsers,» [En línea]. Available: https://dev.opera.com/blog/event-streaming-to-web-browsers/. [Último acceso: 18 7 2016].

[13] G. N. Kotte, «HTML5 Server-Sent Events,» [En línea]. Available: https://www.jayway.com/2012/05/11/html5-server-sent-events/. [Último acceso: 30 07 2015].

[14] CanIUse, «Server-sent events,» [En línea]. Available: http://caniuse.com/#feat=eventsource. [Último acceso: 7 18 2016].

[15] IETF, «The WebSocket Protocol,» [En línea]. Available: https://tools.ietf.org/html/rfc6455. [Último acceso: 18 7 2016].

[16] W3C, «The WebSocket API,» [En línea]. Available: https://www.w3.org/TR/websockets/. [Último acceso: 18 7 2016].

[17] D. Stenberg, «http2 explained,» [En línea]. Available: https://www.gitbook.com/book/bagder/http2-explained/details. [Último acceso: 18 7 2016].

[18] Stackoverflow, «stackoverflow.com,» [En línea]. Available: http://stackoverflow.com/questions/10480122/difference-between-http-pipeling-and-http-multiplexing-with-spdy. [Último acceso: 25 08 2016].

[19] V. Popeskic, «HOL Head-of-line blocking,» [En línea]. Available: https://howdoesinternetwork.com/2015/hol-head-of-line-blocking. [Último acceso: 25 8 2016].

[20] maxcdn, «What is Domain Sharding?,» [En línea]. Available: https://www.maxcdn.com/one/visual-glossary/domain-sharding-2/. [Último acceso: 18 7 2016].

[21] s. admin, «Evolución protocolo HTTP,» [En línea]. Available: https://www.google.es/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwjG5-byoYDOAhUGrxoKHdvRCDAQjB0IBg&url=https%3A%2F%2Fwww.servidoresadmin.com%2Fmejorar-velocidad-carga-http2%2F&psig=AFQjCNEQOEAbIXRcJvjtAhUwI3Ly7MeJhg&ust=1469043029. [Último acceso: 19 7 2016].

[22] Apache, «Apache Module mod_http2,» [En línea]. Available: https://httpd.apache.org/docs/2.4/mod/mod_http2.html. [Último acceso: 13 7 2016].

[23] nginx, «Open Source NGINX 1.9.5 Released with HTTP/2 Support,» [En línea]. Available: https://www.nginx.com/blog/nginx-1-9-5/. [Último acceso: 13 7 2016].

[24] IETF, «WebSocket over HTTP/2,» [En línea]. Available: https://tools.ietf.org/html/draft-hirano-httpbis-websocket-over-http2-01. [Último acceso: 25 8 2016].

[25] golang, «HTTP/2 vs HTTP/1.1,» [En línea]. Available: https://http2.golang.org/gophertiles?latency=0. [Último acceso: 15 7 2016].

[26] cloudfare, «HTTP/2 cloudfare,» [En línea]. Available: https://www.cloudflare.com/http2/. [Último acceso: 15 7 2016].

[27] IETF, «Generic Event Delivery Using HTTP Push,» [En línea]. Available: https://tools.ietf.org/html/draft-ietf-webpush-protocol-07. [Último acceso: 19 7 2016].

[28] W3C, «Web Push API,» [En línea]. Available: https://w3c.github.io/push-api/. [Último acceso: 18 7 2016].

[29] PrimeFaces, «Primefaces,» [En línea]. Available: http://www.primefaces.org/. [Último acceso: 26 8 2016].

[30] Atmosphere, «Atmosphere,» [En línea]. Available: https://github.com/Atmosphere/atmosphere. [Último acceso: 26 8 2016].

[31] Atmosphere, «Extensiones Atmosphere,» [En línea]. Available: https://github.com/Atmosphere/atmosphere/wiki/Atmosphere-PlugIns-and-Extensions. [Último acceso: 26 8 2016].

[32] Union, «Union Server,» [En línea]. Available: http://www.unionplatform.com/?page_id=5. [Último acceso: 29 8 2016].

[33] Union, «Union Procedure Call,» [En línea]. Available: http://www.unionplatform.com/specs/upc/. [Último acceso: 29 8 2016].

[34] PrimeFaces-Extensions, «Timer,» [En línea]. Available: http://www.primefaces.org/showcase-ext/views/timer.jsf. [Último acceso: 30 8 2016].

GLOSARIO

A

ACK: acknowledgement, 23
ALPN: Aplication Layer Protocol Negotiation, 21
API: Application Programming Interface, 28

C

CSS: cascading style sheets, 19

H

HTML: HyperText Markup Language, 36
HTTP: Hypertext Transfer Protocol, 4
HTTPS: Secure HTTP, 29

I

IETF: Internet Engineering Task Force, 4

J

JDK: Java Development Kit, 44
JSF: Java Server Faces, 4
JSON: JavaScript Object Notation, 46

M

MVC: Modelo-Vista-Controlador, 39

N

NPM: Next Protocol Negotiation, 21

P

P2P: Point to Point, 36

R

RFC: Request for Comments, 17
RTC: Real Time Communication, 36
RTT: Round Trip Time, 18

S

SPDY: speedy, 20
SSE: Server-Sent-Events, 9

T

TCP: Transmission Control Protocol, 17
TLS: Transport Layer Security, 21
TTL: Time to Live, 31

U

W

X

7 ANEXOS

7.1 ANEXO A

La implementación del protocolo HTTP/2 en apache y Nginx, entre los servidores más populares, se realiza siguiendo los siguientes pasos.

La siguiente configuración se ha realizado en un Ubuntu 16.04 de 64 bits.

7.1.1 Configuración en Apache

Para la configuración de HTTP/2 en el servidor apache se usará una versión igual o superior a la 2.4.17, con el módulo mod_http2 incluido.

Se puede instalar desde los repositorios oficiales con el siguiente comando:

```
apt-get install apache2
```

Si tras la instalación este módulo no viene añadido, se podrá instalar desde la página oficial de apache [22]. Una vez instalado, es necesario activar el módulo http2 (por defecto viene deshabilitado). Para ello se ejecuta el siguiente comando:

```
a2enmod http2
```

Posteriormente, en el VirtualHost sobre TLS, se debe añadir en la directiva Protocols lo siguiente:

```
<VirtualHost *:443>
 ServerName example.com

 Protocols h2 http/1.1

 # Configuración SSL restante…
</VirtualHost>
```

Así se indica que utilice HTTP/2, y en su defecto HTTP/1.1.

Por último se reinicia el servidor:

```
service apache2 restart
```

7.1.2 Configuración en Nginx

Para la configuración de HTTP/2 en el servidor nginx, se usará una versión igual o superior a la 1.9.5, con el soporte http2 oficial [23].

Se puede instalar desde los repositorios oficiales con el siguiente comando:

```
apt-get install nginx
```

Posteriormente, en la configuración de nginx se añade lo siguiente para indicar que se desea utilizar HTTP/2:

```
server {
 listen 443 ssl http2 default_server;

 ssl_certificate server.crt;
 ssl_certificate_key server.key;
 ...
}
```

Por último, se reinicia el servidor:

```
service nginx restart
```

7.2 ANEXO B

7.2.1 Presupuesto

Para la realización del proyecto se ha empleado software libre, donde se incluyen las siguientes herramientas gratuitas:

- Eclipse IDE.
- Atmosphere Framework.
- Orbiter Framework.
- Tomcat Server.
- Union Server.

Por otro lado, el único coste que supone realizar dicho trabajo consiste en las horas invertidas por el ingeniero, incluyendo su equipo (ordenador, raspberry, etc), como se muestra en la siguiente tabla:

Tabla 12 - Presupuesto

Concepto		Precio (€)
Software		0 €
Horas ingeniero	Cantidad (20€/hora)	
	400 h	8.000 €
Ordenador personal		500 €
Raspberry Pi 2		35 €
TOTAL		8.535 €

7.2.2 Diagrama de Gantt

En la siguiente imagen se muestra la planificación del proyecto, donde se observan 3 tareas principales:

- Investigación teórica.
- Aplicación práctica.
- Documentación.

Figura 58 – Planificación del proyecto