Proyecto fin de carrera

Implementación del protocolo P2PSP usando WebRTC

Alumno:
Cristóbal Medina López
Director:
Vicente González Ruiz
Titulación:
Ingeniero en Informática

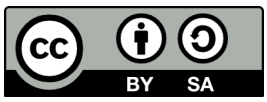
Escuela Politécnica Superior y Facultad de Ciencias Experimentales
Universidad de Almería

25 de junio de 2014

Implementación del protocolo P2PSP usando WebRTC

P2PTV directamente en el navegador, sin plugins.

Licencia



Agradecimientos

A mis padres, por su sacrificio y esfuerzo para ofrecerme la oportunidad que ellos nunca tuvieron. Por ser unos padres ejemplares. Por enseñarme que el trabajo bien hecho siempre tiene su recompensa. “Querer es poder”.

A mis hermanos, por su confianza y complicidad. Por hacer que cualquier cosa se convierta en algo divertido.

A Macarena, por estar siempre a mi lado, por su ayuda y consejo en todas mis decisiones. Porque a su lado todo es más fácil. También a su familia, por su apoyo y por hacerme sentir uno más.

A Vicente, director de este trabajo. Por su entrega, su ayuda incondicional y por darme la oportunidad de aprender a su lado. Por escuchar y valorar mis ideas. Por todo lo que me ha enseñado y por lo que aún me falta por aprender.

En general, a los que han estado a mi lado durante todos estos años, profesores, familiares y amigos, que de una forma u otra me han visto crecer en lo personal y lo académico. Con los que he compartido experiencias, hemos colaborado, me han ayudado o simplemente me han aguantado.

A todos, gracias, muchas gracias.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

Glosario de términos

API Interfaz de programación de aplicaciones. Se usa como una capa de abstracción para otro software. 19, 20, 23, 25–28, 30, 31, 33, 51, 53, 56, 63, 65, 71, 83

bit-rate Es el número de bits transmitidos por unidad de tiempo a traves de la red. También se conoce como tasa de transferencia y ancho de banda. 45, 50

branch Se refiere a una nueva rama de desarrollo. 57

buffer Espacio de la memoria reservado para el almacenamiento temporal de información. 22, 41, 42, 63–65

chunk El stream se divide en trozos, cada trozo recibe el nombre de Chunk. Normalemente hacer referencia a un número N de bytes del stream. 40, 47, 48, 50, 58, 60, 63–65, 72, 74

contenido Hace referencia a un archivo multimedia o una transmisión en directo. 16–18, 22, 39, 42, 51, 53, 56, 57, 61, 81

códec Abreviatura de codificador-decodificador. 20–23, 28, 31, 57, 64, 71

framework Patrón de diseño para el desarrollo y/o la implementación de una aplicación. 26

IETF Internet Engineering Task Force. Es una organización internacional para la normalización de la arquitectura y los protocolo de Internet. 23, 26, 30, 51

interface Conexión funcional entre varios sistemas de cualquier tipo que da lugar a una comunicación entre distintos niveles. 23, 51, 53

JSON Formato ligero para el intercambio de datos. 37, 58

overhead Sobrecarga o redundancia en la información. 27

peer Un cliente que forma parte de la red P2P y está consumiendo (y compartiendo) el contenido. 37, 40, 42, 45–50, 56, 58, 60, 62–66, 71, 74, 83, 85

plugin Una aplicación que se integra en otra para aportar una nueva función específica. 17, 18, 20, 22, 26, 43, 53, 56, 81, 87

responsive Una filosofía de diseño y desarrollo Web que mediante hojas de estilo adapta el sitio web al entorno del usuario. 56, 60, 76

socket Define un tipo especial de manejador de fichero que utiliza un proceso para pedir servicios de red al sistema operativo. 51, 57

source Es la fuente del contenido. Controla el bit-rate. 45, 57

splitter Entidad cuya función es poner en contacto a los peers entre sí, almacenar la lista de peers y enviar el stream original divido en chunks al Team. 45, 47–50, 57, 58, 60, 61, 63–65, 72, 74, 85

stream Referente al contenido. 16, 22, 40, 43–47, 52, 53, 56–58, 60–64, 66, 71, 72, 76

streaming Difusión contínua de flujo de datos multimedia que se consumen a medida que se descargan. 16–18, 22, 23, 26, 39, 41–45, 49, 52, 53, 57, 71, 81, 89

swarm Es el nombre que recibe un Team en las redes PPSP y BitTorrent. 40, 41

team En una red P2PSP es el conjunto de peers que se distribuyen el mismo contenido. 17, 46–51, 53, 56, 58, 61–65, 71, 72, 74, 76, 81

tracker Similar al Splitter. Utilizado en las redes BitTorrent. 40, 41

W3C World Wide Web Consortium. es un consorcio internacional que produce las recomendaciones para la Web. 22, 23, 26, 30, 51

Capítulo 1

Introducción

1.1. Motivación y Objetivos

Los sistemas de streaming de vídeo son servicios utilizados masivamente. De hecho, según Cisco Systems, actualmente más de la mitad de los datos que se transmiten son multimedia (audio y vídeo). Esto ha motivado que aparezcan multitud de arquitecturas de streaming que, en esencia, tratan de transportar tan rápido como sea posible una o varias señales multimedia (vídeo) a tantos usuarios como sea necesario. En una aplicación IPTV, el terminal del usuario recibe una señal de vídeo a través de su conexión a Internet de una forma muy semejante a como ocurre en la TDT (Televisión Digital Terrestre). Dicha señal transporta un contenido que, o bien está siendo generado en ese momento (un programa de TV en directo, por ejemplo), o bien ya está grabado, pero en cualquier caso todos los telespectadores ven concurrentemente la misma señal por el mismo instante de tiempo.

La solución para streaming de vídeo en directo más usada hoy en día es el modelo C/S (Cliente/Servidor) que se basa en la idea de que el stream es generado en un único punto (el servidor) y, bajo demanda (pull model) o de forma automática (push model), el stream es transmitido lo antes posible (tratando de minimizar la latencia) hasta un conjunto de clientes. Si tenemos en cuenta que YouTube, por ejemplo, que es el portal de streaming de vídeo más usado, se basa en este modelo, podría afirmarse que dicha arquitectura es la que más éxito ha tenido hasta la fecha. Sin embargo, ha de tenerse en cuenta que por la propia naturaleza del modelo, los sistemas puros C/S escalan mal si no usa replicación en la parte servidora. Por este motivo, la mayoría de los portales de streaming del tipo C/S utilizan redes de entrega de contenidos o CDNs (Content Delibery Networks) que en definitiva no son más que un conjunto de servidores con el contenido replicado. Por contrapartida, también se dispone del modelo P2P (Peer-to-Peer). En éste, la parte servidora únicamente envía una o unas pocas copias del stream y son los peers (clientes) los que usan su ancho de banda de subida excedente para replicar el stream tantas veces como sea necesario.

Por otra parte, hoy en día el contenido multimedia se consume mediante una gran variedad de dispositivos móviles (smartphone, tablet, netbook, etc). Además,

las tarifas de acceso a Internet cada vez son más reducidas, lo que permite a los usuarios descargar y compartir contenido multimedia —incluso en tiempo real— sin un sobre-coste. Sin embargo, por lo general, todas las compañías se centran en promocionar el ancho de banda de descarga olvidando una parte importante: el ancho de banda de subida. Esta capacidad está siendo desaprovechada por los actuales sistemas comerciales de streaming de vídeo, porque en estos, toda la carga de la red recae sobre el servidor, de modo que el servidor tiene que servir repetidamente el mismo contenido a todos los clientes que lo están solicitando al mismo tiempo. Esto en definitiva produce problemas de denegación de servicio y sistemas poco robustos al existir una única fuente de vídeo.

El protocolo P2PSP (Peer to Peer Straightforward Protocol) [25] es un ejemplo concreto de modelo P2P para streaming de vídeo en directo. Gracias al P2PSP, unido a otras tecnologías, podemos confeccionar clientes de streaming P2P que aprovechen ese ancho de banda que es desperdiciado, enviando una copia de su contenido al resto de clientes. Así conseguimos que todos los clientes compartan, en principio, tanto como reciben a la par que liberamos al servidor de una gran carga computacional y de transmisión, de forma que estamos ante un sistema de streaming muy escalable capaz de soportar un gran número de usuarios. Por otra parte, hacer un sistema capaz de ser ejecutado en múltiples plataformas, sobre todo en dispositivos móviles, sería un avance en el campo de la transmisión de vídeo por Internet, ya que permitiría a cualquier usuario conectarse a un team P2PSP para comenzar a recibir y compartir contenido multimedia, independientemente de la plataforma usada por el usuario.

Este trabajo presenta una nueva implementación del protocolo P2PSP (Peer-to-Peer Straightforward Protocol) [25] sobre la API WebRTC [30] y HTML5 [28]. Como resultado se obtiene un cliente Web capaz de recibir contenido multimedia en tiempo real desde un servidor de streaming. Esta solución, comparada con las anteriores que no están basadas en P2P, minimiza el consumo de ancho de banda en el lado del servidor, provee una buena calidad de servicio y evita la necesidad de descargar plugins u otro software adicional.

A lo largo de este documento se mostrará cómo implementar el módulo DBS (*Data Broadcasting Set of rules*) del P2PSP para ser ejecutado directamente en el navegador Web del usuario sin necesidad de software adicional. Para ello, se repasará la situación actual de la tecnología necesaria que está involucrada en este proceso, con el objetivo de conocer hasta dónde se podrá llevar a cabo la implementación y cómo las tecnologías han de evolucionar para conseguir disfrutar de una implementación completa del P2PSP en el navegador Web.

1.2. Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1. Introducción:** Introducción, objetivos y motivación del proyecto.

- **Capítulo 2. Tecnologías Web:** Se introducirán todas las tecnologías necesarias para llevar a cabo el proyecto así como la situación actual de dichas tecnologías.
- **Capítulo 3. Estado del arte en P2PTV:** Se introducirán conceptos sobre la tecnología P2P, los protocolos más conocidos y cómo funcionan. Además, se hablará sobre la situación actual del streaming sobre el navegador Web.
- **Capítulo 4. Protocolo P2PSP:** Se describirá el protocolo P2PSP desde una perspectiva global, prestando especial atención al módulo *Data Broadcasting Set of rules*.
- **Capítulo 5. Propuesta:** Se propondrá un sistema capaz de reproducir contenido en streaming mediante el protocolo P2PSP directamente en el navegador Web del usuario sin necesidad de plugins.
- **Capítulo 6. Implementación:** Se describirá el procedimiento seguido para la implementación del sistema propuesto y el funcionamiento de cada uno de los elementos que intervienen.
- **Capítulo 7. Evaluación y Resultados:** Se realizarán pruebas del sistema y se evaluarán los resultados obtenidos.
- **Capítulo 8. Conclusiones:** Una vez obtenidos los resultados se sacarán las conclusiones oportunas acerca de la implementación y la consecución de los objetivos.
- **Capítulo 9. Trabajo futuro:** Posibles líneas de expansión del trabajo actual.

Capítulo 2

Tecnologías Web

La evolución exponencial de las capacidades de los navegadores Web en los últimos años ha permitido que se conviertan en una herramienta indispensable capaz de ejecutar aplicaciones complejas, lejos de su uso primitivo (el de ser un mero interprete del lenguaje de marcado de hipertexto). Hoy en día, gracias a la propia evolución del lenguaje HTML junto con otras APIs en las implementaciones modernas de los navegadores, existen numerosas aplicaciones Web que permiten al usuario disfrutar de una experiencia completa en el navegador. Atrás quedaron las páginas Web que sólo podían contener texto, hipervínculos e imágenes. En la actualidad es posible visualizar contenido multimedia con reproductores dignos de ser una aplicación de escritorio hasta complejos sistemas de finanzas o herramientas de modelado.

Cada vez más usuarios prefieren el uso de las aplicaciones en el navegador en lugar de sus equivalentes de escritorio. Las ventajas de este tipo de aplicaciones, tales como: la independencia de la plataforma, el acceso remoto (no es necesario tener instalada la aplicación de forma local), la facilidad de uso o la potencia que tiene la red para dar difusión de la propia aplicación, llevan a pensar que la Web ha llegado para quedarse. No obstante, aún existen algunas limitaciones que provocan que ciertos tipos de aplicaciones no puedan ser llevadas a cabo sobre el navegador.

2.1. HTML5

Han pasado más de 20 años desde que Tim Berners-Lee liberara la primera versión de HTML (HyperText Markup Language) en 1991 [27], en la que Lee describió 22 elementos básicos que especificaban un diseño simple de HTML. En 1993 se convirtió en un estándar de referencia para la elaboración de páginas Web que ha ido evolucionando, incorporando y suprimiendo diversas características, con el fin de convertirse en un lenguaje más eficiente y ser compatible con las distintas plataformas.

HTML5 (HyperText Markup Language 5) es la quinta revisión del lenguaje básico de la Web. Como ya se ha indicado, el código HTML es interpretado por el navegador Web lo que permite que el usuario interactúe con el contenido de forma

sencilla. En esta nueva versión se establecen nuevos elementos y atributos que reflejan el uso que los usuarios le dan al navegador. En la actualidad, por ejemplo, se introducen elementos multimedia como la etiqueta `<video>` y `<audio>` para proporcionar funcionalidades multimedia directamente en el navegador mediante una interfaz estandarizada, haciendo uso de códecs para mostrar los contenidos. Con respecto a este tema, existe una lucha entre los distintos desarrolladores de navegadores Web por imponer códecs privativos frente a una alternativa con códecs libres (se hablará con más detalle acerca de esto a lo largo de este trabajo). Además se incluyen elementos de comunicación en forma de API como son los WebSockets que permiten una comunicación bidireccional entre un servidor y el navegador Web. Por otra parte, algunos elementos usados en la versión anterior del lenguaje han quedado obsoletos, ya que con la introducción de algunos nuevos elementos su existencia carece de significado.

2.1.1. La etiqueta vídeo

La etiqueta `<video>` es una nueva capacidad que incorpora el estándar HTML5 y nos permite reproducir ficheros de vídeo directamente dentro de una página Web sin necesidad de usar plugins o el uso de herramientas de terceros como Flash. Además, cabe la posibilidad de personalizar el reproductor mediante el uso de hojas de estilo CSS y JavaScript, de forma que es posible dar una apariencia única al reproductor.

El problema de la incompatibilidad entre el formato del vídeo y el navegador Web es tratado por HTML5 de la siguiente manera. El elemento vídeo intentará reproducir los objetos fuente (source) en el orden que se indican en el código, de modo que si el primer vídeo no es compatible con el navegador se pasará al siguiente y así sucesivamente hasta encontrar un vídeo que haga uso de un códec de vídeo compatible.

El Código 2.1 muestra un ejemplo del uso de la etiqueta `<video>` en HTML5.

```
1  <!DOCTYPE html>
2  <head>
3     <title>Ejemplo de la etiqueta video</title>
4     <meta http-equiv="Content-Type" ...
          content="text/html;charset=utf-8" />
5  </head>
6  <body>
7     <video controls>
8        <source src="video.mp4" type="video/mp4" />
9        <source src="video.webm" type="video/webm" />
10 </video>
11 </body>
```

Código 2.1: Ejemplo de uso de la etiqueta `<video>`.

Formatos aceptados

H.264

Uno de los códecs de vídeo más populares es MPEG-4 parte 10, conocido también como H.264. El H.264 es un códec de vídeo de alta compresión, desarrollado por el ITU-T Video Coding Experts Group (VCEG) y el ISO/IEC Moving Picture Experts Group (MPEG). Es un formato muy extendido en Internet y en reproductores de vídeo de todo tipo. En la actualidad existe controversia en su uso ya que está protegido por patentes de software, obligando a pagar a los desarrolladores de herramientas que hagan uso del mismo.

El códec H.264 se puede combinar con los códecs de audio ACC o MP3 dentro del contenedor MPEG-4 (conocido como MP4).

OGG

Ogg es un contenedor orientado a la difusión de flujo multimedia con códecs libres de audio y vídeo que no están restringidos por patentes de software. Generalmente, para el vídeo se hace uso del códec Theora y para el audio se hace uso del códec Vorbis, ambos creados por la fundación Xiph.Org.

WebM

WebM es un formato multimedia de audio y vídeo orientado al uso con HTML5, que soporta un solo códec de audio: Vorbis, y un solo códec de vídeo: VP8. VP8 fue creado por On2, una compañía que fue comprada por Google, ocurriendo que poco después del acuerdo Google liberó el códec convirtiéndolo así en un códec de código abierto.

Compatibilidad con los navegadores

Prácticamente todos los navegadores aceptan al menos un formato de audio y vídeo para HTML5. Sin embargo, no todos los códec de vídeo son compatibles en todos los navegadores. En la Tabla 2.1 se muestra una lista de los navegadores más populares junto con los códec de vídeo a los cuales ofrece soporte.

Formato	Chrome	Firefox	Internet Explorer	Opera	Safari
WebM (VP8+Vorbis)	SI	SI	NO*	SI	NO*
OGG (Theora+Vorbis)	SI	SI	NO	SI	NO*
MP4 (H.264+MP3)	SI	NO*	SI	NO	SI
MP4 (H.264+AAC)	SI	NO*	SI	NO	SI

Tabla 2.1: Codecs de vídeo soportados por los actuales navegadores Web.

Algunos navegadores no incorporan el soporte de vídeo para ciertos formatos propietarios a fin de evitar problemas de patentes, y en su lugar se basan en el

soporte del sistema operativo o el hardware. Por otro lado, otros navegadores no son compatibles con los formatos libres pero pueden llegar a serlo mediante la instalación de plugins o software de terceros. Ambos casos se han marcado con un asterisco (*) en la tabla 2.1.

2.1.2. Media Source Extensions

Media Source Extensions [29], también conocido como MSE es una especificación elaborada por el HTML Working Group del W3C con el objetivo de extender las funcionalidades del objeto HTMLMediaElement. Las MSE permiten a los desarrolladores construir media stream dinámicos en JavaScript para la etiqueta `<audio>` y `<video>`. Adicionalmente, se define un modelo de buffering que describe cómo los agentes de usuario deben actuar cuando se añaden diferentes medios.

El objeto MediaSource representa una fuente de datos multimedia para un HTMLMediaElement. Se usa el objeto SourceBuffer para añadir datos multimedia al buffer, lo que hace posible alimentar el buffer en tiempo real al mismo tiempo que el elemento vídeo está reproduciendo el contenido.

Las MSE aún están en proceso de estandarización por el W3C, siendo actualmente una *Candidate Recommendation*. Algunas de las ventajas que incorpora y/o extiende del elemento Media Source son:

- Permite construir streams multimedia mediante JavaScript independientemente de cómo el elemento multimedia sea incrustado.
- Define un modelo de buffering que facilita casos de uso como streaming adaptativo, inserción de publicidad, saltos de tiempo y edición de vídeo.
- Aprovecha la caché del navegador tanto como sea posible.
- No requiere soporte para un códec particular.

Navegador	MSE	Códec para MSE
Chrome	SI	WebM (VP8+Vorbis)
Chrome Canary	SI	WebM (VP8+Vorbis) y MPEG-DASH
Firefox	NO	Ninguno
Firefox Nightly	SI	WebM (VP8+Vorbis)
Internet Explorer	SI	MPEG-DASH
Opera	NO	Ninguno
Safari	NO	Ninguno

Tabla 2.2: Compatibilidad de las Media Source Extensions con los navegadores Web actuales.

Como se muestra en la Tabla 2.2 y aunque las MSE se definen como agnóstico en el contenido, no todos los navegadores implementan Media Source Extension y los

que lo hacen, implementan códecs de vídeo distintos lo que hace que una aplicación tenga que ser modificada para mantener la compatibilidad entre navegadores. En este momento, Chrome, Firefox Nightly (la versión para desarrolladores) e Internet Explorer soportan este nuevo elemento. La mala noticia es que lo hacen parcialmente, de modo que Chrome y Firefox soportan únicamente el formato de vídeo WebM (VP8+Vorbis) que además sea específicamente codificado para streaming añadiendo keyframes regularmente. IE por su parte soporta MP4 segmentando. La buena noticia es que se está trabajando para mejorar esto en el futuro.

2.1.3. La API de archivos

La API de archivos es una forma estándar de interactuar con archivos en HTML5. Esta API define la representación básica de los archivos, listas de archivos y errores que sucedan en el acceso a dichos archivos. La especificación también define una interface que representa *raw data* (datos sin procesar) que pueden procesarse de forma asíncrona en el hilo principal de acuerdo con los agentes de usuario.

Existen tres interface principales:

1. **File:** representa un archivo individual de datos que se obtienen normalmente del sistema de archivos.
2. **Blob:** (Binary Large Object) representa un raw data permitiendo así fragmentar un archivo en intervalos de bytes.
3. **FileList:** representa un conjunto de objetos File.

Gracias a la interfaz FileReader es posible leer los archivos de forma asíncrona al proceso principal, evitando así el uso del bloque de la interfaz de usuario. La API de archivos define un modelo de eventos para leer y acceder a un archivo o un raw data de forma asíncrona a través de atributos de control y disparadores de eventos.

2.1.4. WebSocket

WebSocket es un protocolo que provee al navegador Web de una canal de comunicación bidireccional con el servidor sobre una única conexión TCP. Su diseño se basa en las infraestructuras HTTP existentes, de modo que se ha diseñado para funcionar sobre los puertos 80 y 443 además de sobre proxies HTTP. Sin embargo, la especificación no limita el uso de HTTP en WebSocket sino que futuras implementaciones podrían llevarse a cabo en un puerto especifico usando el intercambio de mensajes propuesto para HTTP sin tener que reinventar el protocolo. El protocolo WebSocket se estandarizó en 2011 por el IETF y actualmente está siendo estandarizada su API por el W3C.

El diseño de WebSocket se ha realizado pensando en su implementación para el navegador Web por un lado, y para el servidor Web en el otro. Gracias a este

protocolo es posible mantener conexiones persistentes entre el navegador y el servidor donde ambas partes pueden comenzar el intercambio de información en cualquier momento. Hasta que WebSocket se convirtió en un estándar las aplicaciones Webs que necesitaban comunicación bidireccional entre el navegador y el servidor hacían un uso excesivo del paradigma solicitud/respuesta del HTTP haciendo solicitudes periódicas a la otra parte. Más tarde se popularizó el uso de Long Polling, una técnica que consiste en dejar constantemente una conexión HTTP abierta con el servidor. Cuando un cliente hace una petición, el servidor, en lugar de enviar una respuesta inmediatamente al cliente, mantiene abierta esta conexión el tiempo que sea necesario hasta tener algo que enviar como respuesta. Cuando el cliente recibe la información inmediatamente hace otra petición para repetir el proceso. Al igual que el paradigma de solicitud/respuesta esta última técnica hace un uso excesivo del HTTP. Esto hace que ambas técnicas no sean factibles en aplicaciones que requieran baja latencia.

```
1  // Se dispara cuando se inicia la conexion
2  var conexion = new WebSocket('ws:URL');
3  conexion.onopen = function () {
4    connection.send('Hola'); // Enviar un mensaje al servidor
5  };
6
7  // Log de errores
8  connection.onerror = function (error) {
9    console.log('WebSocket Error ' + error);
10 };
11
12 // Mostrar por consola mensajes recibidos del servidor
13 connection.onmessage = function (e) {
14   console.log('Servidor dice: ' + e.data);
15 };
```

Código 2.2: Ejemplo del uso de WebSocket.

2.2. JavaScript

Desarrollado originalmente por Brendan Eich con el nombre de *Mocha*, JavaScript ha ido evolucionando hasta convertirse en uno de los lenguajes más importantes hoy en día. JavaScript es un lenguaje de programación interpretado y se define como orientado a objetos. La forma de uso más habitual en JavaScript es en el lado del cliente permitiendo mejoras en la interfaz de usuario y aplicaciones Web dinámicas. No obstante, puede ser usado en el lado del servidor así como en aplicaciones externas a la Web.

Algunas de las características básicas de JavaScript son:

- **Estructurado:** JavaScript soporta casi la totalidad de la estructura de programación del lenguaje C.

- **Dinámico:** El tipo de dato está asociado al valor y no a las variables.
- **Orientado a objetos:** JavaScript es un lenguaje orientado a objetos.
- **Funcional:** Las funciones son objetos en sí mismo y por tanto poseen propiedades y métodos.
- **Uso de prototipos:** Usa prototipos en lugar de clases para el uso de la herencia.

```
1 console.log("Hola Mundo");
```

Código 2.3: Hola Mundo en JavaScript.

El uso habitual de JavaScript es en el navegador Web embebido en páginas HTML que interactúan con el modelo de objetos del documento (DOM: Document Object Model) de la página. El DOM es una API que proporciona un conjunto estándar de objetos para la representación de documentos HTML y XML, y facilita una interfaz que permite acceder a estos objetos y modificarlos.

El navegador por norma general incluye una consola que puede ser usada para imprimir mensajes desde JavaScript, tal y como se muestra en el Código 2.3.

2.3. Python

A finales de los años ochenta Guido van Rossum, con la idea de suceder al lenguaje ABC, crea un nuevo lenguaje de programación que denomina Python. Como características básicas decir que Python:

- Es un lenguaje multiparadigna lo que permite a los programadores adoptar su propio estilo de programación, dándoles la posibilidad de usar diferentes paradigmas como son: programación orientada a objetos, programación imperativa o programación funcional.
- Permite la resolución dinámica de nombres, es decir, en tiempo de ejecución es capaz de escoger el método que responderá a un determinado mensaje, constituyendo así un tipo de polimorfismo.
- Es fácilmente extensible, permitiendo escribir módulos nuevos de forma sencilla mediante el lenguaje C o C++.
- Al igual que en JavaScript, las variables se definen de forma dinámica, permitiéndose así no tener que especificar el tipo de dato sino que puede tomar distintos valores en un momento u otro.

- Se puede usar para aplicaciones Web, y aunque esto puede ser una ardua tarea, no lo es tanto si se usan framework u otras herramientas que han sido diseñadas para ayudar a los desarrolladores a crear aplicaciones Web de una forma más rápida y segura. Por tanto, es posible desarrollar haciendo uso de los framework de modo que sólo es necesario escribir una pocas líneas de código mientras las tareas más duras como la responsabilidad de las comunicaciones o la infraestructura de bajo nivel son llevadas a cabo por el framework mientras los desarrolladores se centran en la lógica de su propia aplicación. Una lista de frameworks para Python está disponible en [7].

2.4. Web Real Time Communication

WebRTC (Web Real Time Communications) [26] es un proyecto de software libre mantenido por Google, Mozilla y Opera, que pretende convertirse en un nuevo estándar que extiende las capacidades del navegador Web. La gran novedad en cuanto a la situación anterior a la aparición de esta API es la capacidad que otorga al navegador de comunicarse directamente y en tiempo real con otros navegadores utilizando la arquitectura peer-to-peer.

Esta API está siendo definida conjuntamente por el W3C (World Wide Web Consortium) y por el IETF (Internet Engineering Task Force). La misión de ambas organizaciones es conseguir una API JavaScript que junto con las etiquetas necesarias del estándar HTML5 pueda definir un nuevo protocolo de comunicación con el que sea posible comunicar los navegadores directamente entre sí. Además, la API define también otros elementos que permiten acceder por ejemplo a la webcam y al micrófono (entre otros periféricos) sin necesidad de plugins.

2.4.1. PeerConnection

PeerConnection es un componente de WebRTC que permite manejar una comunicación estable y eficiente de streaming de datos entre peers, es decir, permite una comunicación directa de navegador a navegador. Esto representa una asociación con otro navegador remoto (peer) que está ejecutando otra instancia de la misma aplicación JavaScript. Antes de conseguir la conexión entre los peers es necesario un intercambio previo de mensajes para acordar los parámetros de la comunicación, lo que se hace a través de un servidor de señalización (signaling server), ver Sección 2.4.5. Una vez que se ha establecido la conexión, es posible enviar mensajes entre los navegadores directamente sin pasar por ningún servidor intermedio. En el Código 6.8 se muestra cómo crear una conexión de este tipo.

Por otra parte, como se mostrará en la Sección 2.4.5, el mecanismo usado por PeerConnection se basa en el uso del protocolo ICE junto con los servidores STUN y TURN para permitir a los streams basados en el protocolo UDP atravesar NATs y firewalls.

```
1  pc = new RTCPeerConnection({iceServers: [{ url: ...
       'stun:stun.l.google.com:19302' }]};);
```

Código 2.4: Crear un conexión mediante PeerConnection.

Nótese que en el Código 6.8 se ha usado un servidor STUN de Google disponible públicamente para desarrollos experimentales. En la Sección 6.3 se muestra cómo instalar y configurar un servidor STUN mediante soluciones de software libre.

2.4.2. DataChannel

Hasta la aparición de WebRTC, enviar datos entre varios navegadores era un proceso ineficiente, poco escalable y falto de privacidad porque era necesario enviar todos los datos a un servidor intermedio que se encargara de reenviar la información al resto de navegadores involucrados.

La API `DataChannel` de WebRTC permite al navegador Web una comunicación de datos bidireccional directamente de un peer a otro, así como un conjunto flexible de tipos de datos. Entre los tipos de datos soportados están `Blob`, `ArrayBuffer` y `ArrayBufferView`.

Además, `DataChannel` cuenta con dos modos de funcionamiento: el modo no confiable (similar a UDP) y el modo confiable (similar a TCP). El modo no confiable (unreliable mode) no tiene overhead y permite un funcionamiento más rápido pero presenta la desventaja de que no garantiza ni la entrega de los mensajes ni el orden de llegada al otro lado. El modo fiable (reliable mode) garantiza la transmisión de todos los mensajes y su llegada en orden hasta el otro lado. Como desventaja, este modo presenta un overhead mayor y puede llegar a ser más lento, lo que lo hace el menos adecuado para aplicaciones en tiempo real.

Modo	TCP	UDO	SCTP
Transporte confiable	SI	NO	Configurable
Entrega ordenada	SI	NO	Configurable
Control de flujo	SI	NO	SI
Control de congestión	SI	NO	SI

Tabla 2.3: Resumen de características de los protocolos TCP, UDP y SCTP.

Algunas de las opciones que pueden ser configuradas de `DataChannel` son:

- `id`: Sobrescribe el id por defecto del canal. Acepta valores de tipo `unsigned short`.
- `ordered`: Indica si es necesario garantizar el orden. Puede tomar los valores `true` o `false`.
- `maxPacketLifeTime`: Máximo limite de tiempo que el canal intentará retransmitir el dato. Acepta valores de tipo `unsigned short`.

- `maxRetransmits`: Máximo número de veces que el canal intentará retransmitir el mensaje si no se ha conseguido enviar. Acepta valores de tipo `unsiged short`.
- `negotiated`: Si se establece a `true`, elimina la configuración automática del data channel en el otro peer, lo que implica que se provee una configuración propia con el mismo `id` que en el otro lado.
- `protocol`: permite usar un subprotocolo para el `DataChannel`.

En el Código 2.5 se muestra un ejemplo de configuración de las opciones del objeto `DataChannel` que se han mostrado anteriormente.

```
1 var opcionesDataChannel = {
2   ordered: false, // No garantiza el orden
3   maxRetransmitTime: 3000, // tiempo en milisegundos
4   maxRetransmits: 5, // numero de intentos maximo
5 };
```

Código 2.5: Configurar opciones `DataChannel`.

Todos los navegadores que tienen implementado WebRTC hacen uso del protocolo SCTP (Stream Control Transmision Protocol) para la API `DataChannel`, que además es encapsulado sobre DTLS (Datagram Transport Layer Security), un derivado de SSL (lo cuál indica que el cifrado de información es obligatorio en WebRTC y hace de este un protocolo de intercambio de información seguro).

```
1   peerConnection.createDataChannel("channel", opcionesDataChannel);
```

Código 2.6: Crear un `DataChannel`.

El Código 2.6 muestra la forma correcta de crear un `DataChannel` mediante la API `DataChannel` de WebRTC.

2.4.3. Arquitectura WebRTC

El propósito de WebRTC es construir una plataforma estable para la comunicación en tiempo real entre los navegadores, incluso en diferentes plataformas. Para este fin se ha definido la arquitectura de WebRTC que se muestra en la Figura 2.1. Como se puede ver, `RTCPeerConnection` permite a los desarrolladores Web crear aplicaciones WebRTC mediante el uso de JavaScript de forma sencilla, sin necesidad de conocer la complejidad que la API esconde en los niveles inferiores. Los códecs y protocolos hacen prácticamente todo el trabajo para que la comunicación en tiempo real sea posible.

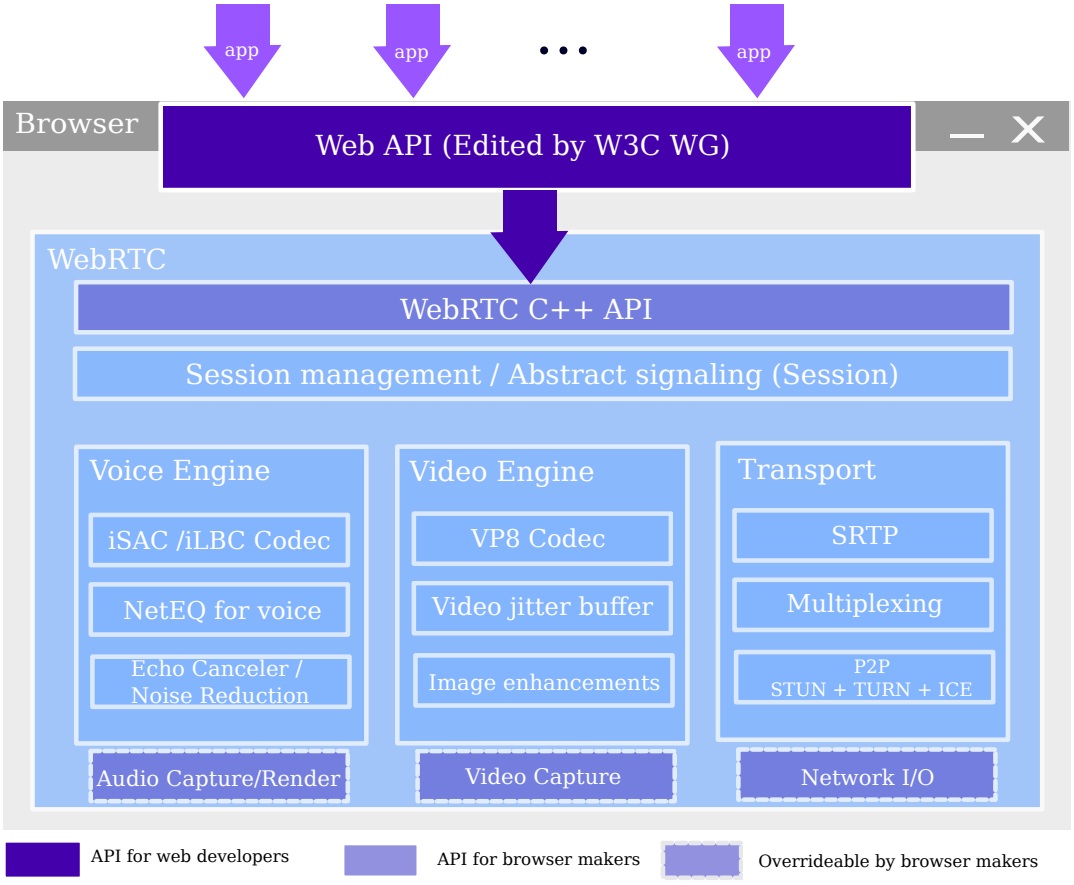


Figura 2.1: Arquitectura WebRTC.

Elemento	Chrome	Firefox	Internet Explorer	Safari	Opera
PeerConnection	SI	SI	NO	NO	SI
ORTC API	NO	NO	Parcial	NO	NO
getUserMedia	SI	SI	Parcial	NO	NO
mediaConstraints	Parcial	NO	NO	NO	Parcial
soporte TURN	SI	Parcial	NO	NO	Parcial
MediaStream	SI	Parcial	NO	NO	SI
WebAudio	SI	SI	NO	NO	SI
DataChannel	SI	SI	NO	NO	SI
Screen Sharing	SI	NO	NO	NO	NO
Stream rebroadcasting	Parcial	NO	NO	NO	NO
Solid interoperability	Parcial	Parcial	NO	NO	Parcial
Echo cancellation	SI	NO	NO	NO	Parcial

Tabla 2.4: Soporte de la API WebRTC en los navegadores actuales.[1]

2.4.4. Soporte de la API en los navegadores Web

Debido a que WebRTC aún no es un estándar definitivo, no todos los navegadores Web están implementándolo por igual. De hecho, algunos de ellos no tienen pensado hacerlo por el momento. Aunque se está trabajando en esta línea, es posible que WebRTC no llegue a ser un estándar tal y como se conoce hoy en día. Para ello es necesario que el IETF y el W3C junto con el resto de proveedores implicados en la Web alcancen un consenso. La Tabla 2.4 muestra un resumen de los elementos de la API que ya han sido desarrollados en cada uno de los navegadores Web actuales.

Por otra parte ya existe un grupo dentro del W3C dedicado a trabajar sobre ORTC (Object Real Time Communications), cuyos integrantes pretenden discutir acerca de la API WebRTC y la forma en que se está llevando a cabo. Este grupo propone alcanzar el estándar modificando la API WebRTC actual.

Como muestra la Tabla 2.4, Google Chrome, Mozilla Firefox y Opera son los navegadores Web que más están trabajando en la implementación de la API. En el lado opuesto se encuentran Internet Explorer y Safari, los cuales, no han implementado nada relacionado con la API WebRTC en sí misma.

2.4.5. Elementos y proceso de señalización

Como se ha mencionado en la Sección 2.4.1, WebRTC usa la clase `PeerConnection` para realizar una comunicación directa entre los navegadores (peers), pero es necesario un mecanismo que permita coordinar esa comunicación. Este mecanismo recibe el nombre de proceso de señalización (signaling). Sin embargo, la especificación no define ningún mecanismo de señalización en WebRTC. Esto significa que es tarea del desarrollador elegir el protocolo de señalización que crea oportuno.

La señalización consiste básicamente en intercambiar tres tipos de mensajes:

1. **Mensajes de control de sesión:** permiten inicializar o cerrar la comunicación e informar de errores.

2. **Configuración de la red:** contienen información de la red necesaria para iniciar la conexión P2P.

3. **Características multimedia:** indica los códecs soportados por el navegador y las características de la información que se desea enviar.

Para el intercambio de información se utiliza el protocolo SDP (Session Description Protocol) que nos permite describir la información de las sesiones de comunicación multimedia. La API WebRTC define un objeto de tipo SDP para almacenar y compartir dicha información. Un ejemplo del contenido de un mensaje SDP puede verse en Código 2.7.

```
1  v=0
2  o=- 5148251955052202386 2 IN IP4 127.0.0.1
3  s=-
4  t=0 0
5  a=group:BUNDLE audio data
6  a=msid-semantic: WMS
7  m=audio 1 RTP/SAVPF 111 103 104 0 8 107 106 105 13 126
8  c=IN IP4 0.0.0.0
9  a=rtcp:1 IN IP4 0.0.0.0
10 a=ice-ufrag:MrZfMOyTeofdkalZ
11  a=ice-pwd:Zi68MAZJcGR8WVQEPHzIWeBQ
12 a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
13 a=sendonly
14 a=mid:audio
15 a=rtcp-mux
16 a=crypto:1 AES_CM_128_HMAC_SHA1_80 ...
       inline:T93pY5PBv4fgDmVIuz7qmL3y9UXozTZdjUCKYlrn
17 a=rtpmap:111 opus/48000/2
18 ...
```

Código 2.7: Ejemplo del contenido de un mensaje SDP.

Antes de comenzar la comunicación peer-to-peer es necesario que el proceso de intercambio de información realizado mediante la señalización se haya llevado a cabo de forma satisfactoria.

Intercambio de información Offer/Answer

Cuando dos peers (A y B) quieren comenzar una comunicación entre ellos mediante `PeerConnection` deben seguir los siguientes pasos:

1. A crea un objeto `RTCPeerConnection` con un manejador de eventos del tipo *onicecandidate*. El manejador se disparará cuando existan candidatos disponibles.

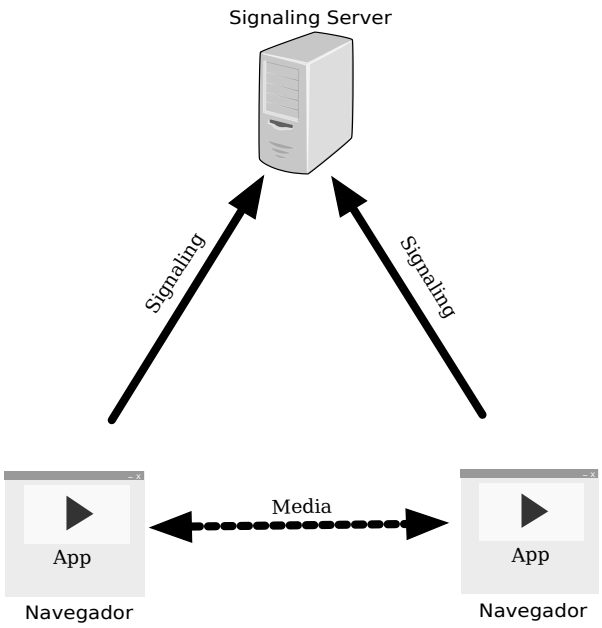


Figura 2.2: Proceso de señalización (signaling).

2. *A* envía los datos del candidato a *B* mediante un canal de señalización.

3. *B* recibe el mensaje de *A* y añade el candidato al SDP remoto correspondiente a ese peer.

Además de esta información, es necesario intercambiar los mensajes SDP que contienen las características de la información multimedia:

1. *A* crea una oferta llamando al método `createOffer()` con el que obtiene el objeto `SessionDescription` que contiene el mensaje SDP con la información de la sesión de *A*.

2. *A* usa el método `setLocalDescription()` para establecer la descripción de su sesión (objeto `SessionDescription`) de forma local y envía el objeto a *B* mediante el canal de señalización.

3. *B* recibe el objeto y establece la descripción de la sesión de *A* usando `setRemote Description()`.

4. *B* ejecuta el método `createAnswer()` pasándole la descripción de sesión que obtuvo de *A* y se genera un mensaje SDP compatible con el recibido desde *A*. *B* establece ese mensaje de respuesta como su descripción de sesión local y lo envía a *A*.

5. Cuando A recibe la descripción de la sesión (SDP) desde B, A la establece como descripción remota con el método `setRemoteDescription`.

6. A y B se pueden comunicar.

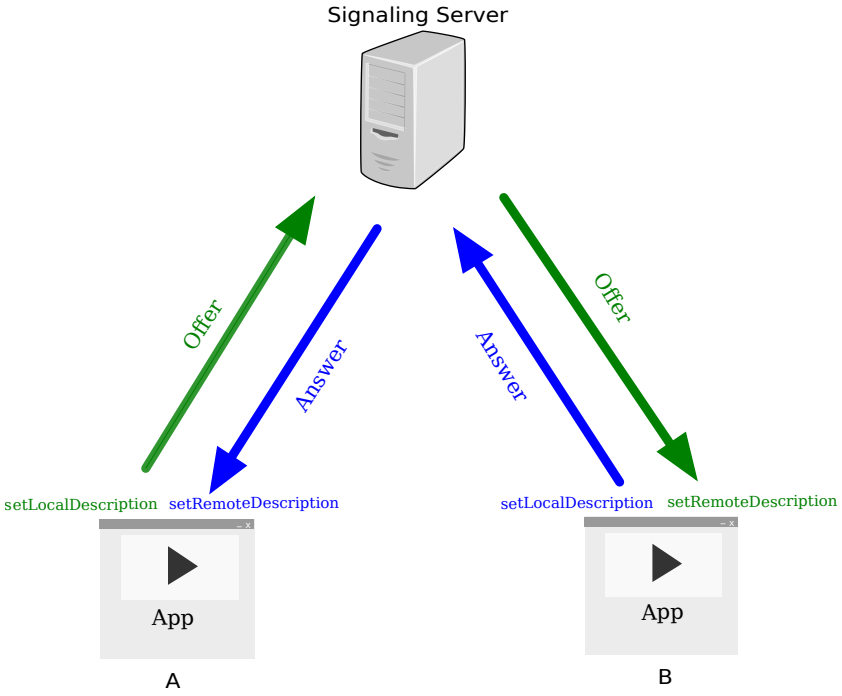


Figura 2.3: Intercambio de información Offer/Answer.

En Código 2.8 se implementa en código JavaScript mediante la API WebRTC todo el proceso que se acaba de describir.

```
1  var signalingChannel = new SignalingChannel();
2  var configuration = { "iceServers": [{ "url": ...
       "stun:stun.example.org" }] };
3  var pc;
4
5  // call start() to initiate
6  function start() {
7      pc = new RTCPeerConnection(configuration);
8
9      // send any ice candidates to the other peer
10     pc.onicecandidate = function (evt) {
11         if (evt.candidate)
12             signalingChannel.send(JSON.stringify({ "candidate": ...
                   evt.candidate }));
13     };
```

```
14
15      // let the "negotiationneeded" event trigger offer generation
16      pc.onnegotiationneeded = function () {
17          pc.createOffer(localDescCreated, logError);
18      }
19
20      // once remote stream arrives, show it in the remote video ...
            element
21      pc.onaddstream = function (evt) {
22          remoteView.src = URL.createObjectURL(evt.stream);
23      };
24
25      // get a local stream, show it in a self-view and add it to ...
            be sent
26      navigator.getUserMedia({ "audio": true, "video": true }, ...
            function (stream) {
27          selfView.src = URL.createObjectURL(stream);
28          pc.addStream(stream);
29      }, logError);
30  }
31
32  function localDescCreated(desc) {
33      pc.setLocalDescription(desc, function () {
34          signalingChannel.send(JSON.stringify({ "sdp": ...
                pc.localDescription }));
35      }, logError);
36  }
37
38  signalingChannel.onmessage = function (evt) {
39      if (!pc)
40          start();
41
42      var message = JSON.parse(evt.data);
43      if (message.sdp)
44          pc.setRemoteDescription(new ...
                RTCSessionDescription(message.sdp), function () {
45              // if we received an offer, we need to answer
46              if (pc.remoteDescription.type == "offer")
47                  pc.createAnswer(localDescCreated, logError);
48          }, logError);
49      else
50          pc.addIceCandidate(new RTCIceCandidate(message.candidate),
51              function () {}, logError);
52  };
53
54  function logError(error) {
55      log(error.name + ": " + error.message);
56  }
```

Código 2.8: Ejemplo de intercambio de mensajes SDP e ICE (W3C).

ICE Framework

Para conseguir la información de la red atravesando firewalls y NATs se hace uso del protocolo ICE. ICE permite que se prueben distintas rutas para comunicar dos terminales entre sí, acordando una común. De forma que, por ejemplo, si están en la misma red local, se comparte la información de forma local sin necesidad de utilizar otros servicios. Pueden existir 3 tipos de candidatos:

1. **Host candidates:** Son candidatos locales, como por ejemplo las tarjetas de red del equipo. Contiene direcciones IP privadas.
2. **Reflexive candidates:** Se obtienen realizando consultas a servidores STUN y contiene IPs públicas. Permite atravesar NATs de tipo Cone.
3. **Relay candidates:** Se obtienen realizando consultas a servidores TURN y contiene IPs públicas. Todos los datos se transmiten a través de él. Permite atravesar NATs simétricos.

En primer lugar ICE, mediante un servidor STUN (ver Sección 2.4.5), intenta conectar los peers directamente haciendo uso de UDP para conseguir la latencia más baja posible. Si no es posible realizar la conexión mediante UDP, ICE intenta hacerlo vía HTTP, que usa TCP. En caso de que la conexión vuelva a fallar (normalmente porque al menos uno de los peers está detrás de un NAT simétrico o un firewall) ICE hace uso de un servidor TURN (ver Sección 2.4.5). Una vez se obtienen los candidatos (IP y puerto) se añaden al mensaje SDP y se comparten.

```
1 a=candidate:1 1 UDP 6556159 192.168.1.1 50036 typ host
2 a=candidate:6 2 UDP 6556159 171.211.100.26 50135 typ srflx raddr ...
      192.168.1.1
3 rport 50039
4 a=candidate:2 1 UDP 6556159 120.50.141.51 50099 typ relay raddr ...
      120.50.141.51
5 rport 50099
```

Código 2.9: Ejemplo de cada uno de los tipos de candidatos.

TURN/STUN

STUN

STUN (Session Traversal Utilities for NAT) [11] es un protocolo de red que permite a un host que está detrás de un NAT descubrir su dirección IP pública, el puerto y el tipo de NAT. Es muy usado para permitir atravesar el NAT a aplicaciones de mensajería, vídeo y voz.

STUN soporta los siguientes tipos de NAT:

- Full Cone: Una dirección (IP:Puerto) interna es mapeada en una dirección externa, de modo que cualquier paquete de la dirección interna es enviado a través de la externa. Por otro lado, un host externo puede hacer llegar un paquete a la dirección interna enviándolo a la externa.
- Restricted Cone: Igual al anterior con la salvedad de que un host externo puede hacer llegar un paquete a la dirección interna sólo si desde la dirección interna previamente se ha enviado un paquete a la dirección IP del host externo (no importa el puerto).
- Port Restricted Cone: Similar al NAT Restricted Cone, pero en este caso si importa el puerto, es decir, para poder recibir de un host externo, es necesario haberle enviado previamente un paquete a su dirección IP y puerto.

El único tipo de NAT que no soporta es el NAT simétrico, en cuyo caso se hace uso de un servidor TURN.

TURN

TURN (Trasversal Using Relay NAT) [12] es un protocolo de red que permite a un host situado detrás de un NAT simétrico o un firewall enviar y recibir datos mediante TCP o UDP. Cuando no hay una alternativa para comunicar dos peers directamente es necesario utilizar un host para que actúe como repetidor haciendo de intermediario entre los peers, de forma que es el encargado de hacer el intercambio de paquetes entre dos o más peers.

El protocolo TURN fue diseñado en el contexto del ICE framework para dar una solución al problema de atravesar NAT simétricos o cortafuegos, aunque es posible hacer uso de TURN sin que ICE esté implicado.

Signaling Server

Como ya se ha dicho, la señalización es un mecanismo abstracto, es decir, WebRTC no especifica cómo ha de llevarse a cabo. Sin embargo, lo que parece claro es que debe existir un servidor en Internet que mediante algún protocolo de comunicación haga las funciones de intermediario entre los peers, de forma que un grupo de varios peers sea capaz de informar a todos y cada uno de los peers sobre la existencia del resto de peers.

Un Signaling Server puede ser cualquier equipo que tenga conexión de cara a Internet. Existen multitud de técnicas para implementar el servidor de señalización. De hecho, pueden ser protocolos estándar que ya se usan para fines similares —como lo son SIP o XMPP— o puede ser cualquier protocolo que permita comunicación bidireccional.

Signaling vía WebSocket

WebSocket (ver Sección 2.1.4) es una buena elección para llevar a cabo el proceso de señalización. Este nuevo protocolo de HTML5, que permite la comunicación bi-direccional del navegador con el servidor, es ideal para crear fácilmente un Signaling Server que permita presentar cada peer al resto de peers. Por otro lado, cualquier navegador Web que soporte WebRTC con total seguridad soportará WebSocket. Usar JSON junto a WebSocket es una buena opción para realizar el intercambio del objeto "session description".

```
1  //Se inicializa el signaling channel
2  var signalingChannel = new WebSocket("ws://ip:port/");
3  ...
4  //Se envia SDP local
5  signalingChannel.send(JSON.stringify({ "sdp": localDescription }));
6  ...
7  signalingChannel.onmessage = function (evt) {
8    //Se recibe el SDP remoto
9    var message = JSON.parse(evt.data);
10 ...
```

Código 2.10: Ejemplo genérico de intercambio del mensaje SDP.

La Figura 2.4 muestra el proceso completo de señalización entre dos peers. El peer *A* se encuentra detrás de un NAT simétrico (o de un firewall), y por tanto en el proceso debe intervenir un servidor TURN, además de un servidor STUN y el servidor de señalización (Signaling).

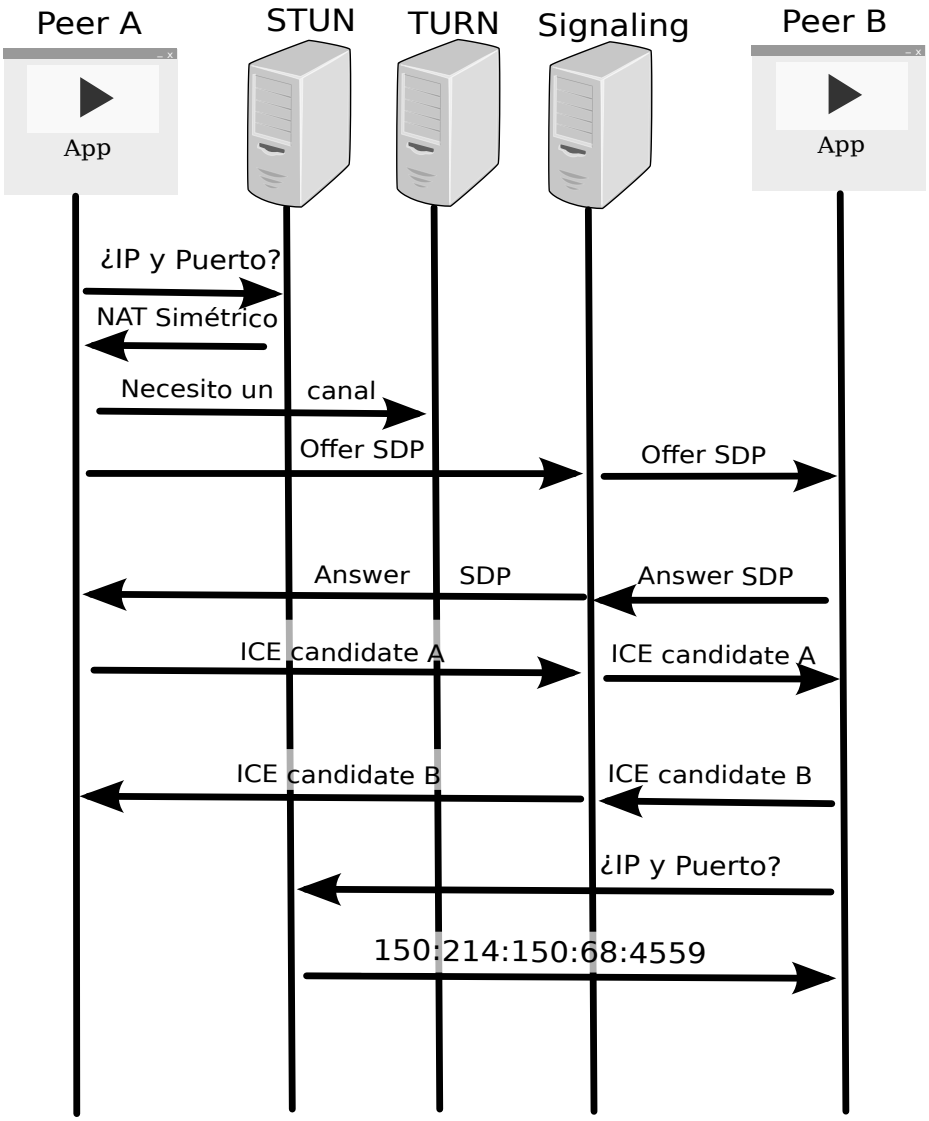


Figura 2.4: Proceso completo de señalización.

Capítulo 3

Estado del arte en P2PTV

El streaming de vídeo sobre Internet, o IPTV, consiste en distribuir contenido multimedia sin interrupciones a través de la red de manera que el usuario final reproduce el contenido al mismo tiempo que lo está recibiendo. El streaming en directo, además de lo anterior, permite transmitir eventos que están sucediendo justo antes de la difusión. Gracias a esta tecnología ya no es necesario descargar los ficheros de vídeo completamente antes de ser reproducidos.

Existen diversas soluciones para el streaming de vídeo sobre Internet. Las más comunes, como ya se apuntaba en la Introducción (Capítulo 1), suelen estar basadas en el modelo cliente/servidor, del que un claro ejemplo es YouTube. Esta tecnología también es muy popular en servicios de vídeo-llamada como Skype y recientemente Hangouts de Google. Por otra parte, existen soluciones de streaming de vídeo basadas en tecnología P2P (P2PTV), las cuales han sido un campo de investigación intensiva en los últimos años, surgiendo propuestas como el PPSP [8], BitTorrent [2] y P2PSP [18] entre otras muchas.

Tanto las soluciones basadas en el modelo cliente/servidor como las basadas en el modelo P2P suelen utilizar protocolos de transporte ligeros, tales como UDP (User Datagram Protocol) o RTSP (Real Time Streaming Protocol), ya que el control de congestión aplicado por protocolos como el TCP (Transmission Control Protocol) puede ser contraproducente para este tipo de aplicaciones.

En las siguientes secciones se profundizará en los protocolo PPSP y BitTorrent. EL protocolo P2PSP, objeto de este trabajo, se describirá en el próximo capítulo.

3.1. PPSP

El protocolo PPSP (Peer-to-Peer Streaming Peer Protocol) es un protocolo diseñado para difundir el mismo contenido a un grupo de clientes interesados. El protocolo soporta dos tipos de streaming: (1) bajo demanda y (2) contenido en directo. Está basado en el paradigma P2P donde los clientes (peers) consumen el contenido a la vez que lo comparten con el resto del grupo, lo que permite un sistema donde

cada peer puede compartir su ancho de banda.

PPSP incluye técnicas para expulsar peer maliciosos y se ha diseñado para ser flexible y extensible. Hace uso de diferentes mecanismos para controlar la solidaridad de los peers, dispone de un esquema de trackers centralizado y de una tabla de hash distribuida. Esta tabla usa el algoritmo Merkle hash tree que calcula el un árbol de hash de los chunks de forma recursiva, con la idea de mantener la integridad del contenido transmitido frente a ataques por envenenamiento del stream.

El PPSP ha sido diseñado para ser un protocolo genérico que permita ejecutarse sobre distintos protocolos de transporte, actualmente se ejecuta sobre UDP usando LEDBAT para el control de congestión.

3.1.1. Funcionamiento básico

La unidad básica de comunicación en PPSP es el mensaje. Varios mensajes son multiplexados dentro de un único datagrama para la transmisión. Un datagrama tendrá distintas representaciones dependiendo del protocolo de transporte utilizado.

Suponiendo que dos peer quieren compartir un vídeo, el proceso que se seguiría usando el PPSP se describe brevemente a continuación:

1. **Unión al swarm:** En primer lugar los peer tienen que conocerse entre ellos y para ello usan un tracker que los pone en contacto. Después los peer se comparten mensajes del tipo HANDSHAKE para comenzar la comunicación.

2. **Intercambio de chunks:** Para solicitar un chunk, A envía un mensaje de tipo REQUEST indicando el chunk que quiere descargarse desde B. B responde con un datagrama que contiene el tipo de chunk, los datos y un mensaje de integridad que incluye todos los hash necesarios para que A pueda comprobar la integridad del mensaje. Después A envía una confirmación de la recepción del chunk a B y le solicita un nuevo chunk.

3. **Dejar el swarm:** Para abandonar el swarm de forma correcta el peer A envía un mensaje informando del evento y abandona el tracker. Los peers que reciben el datagrama eliminarán a A de su lista de peer.

3.1.2. Implementaciones

Swirl

Swirl [21] ese es el nombre de una de las primeras implementaciones conocidas para el protocolo PPSP. Swirl es de código abierto y actualmente es una implementación en fase alpha. Usa el lenguaje de programación Erlang para el núcleo de la aplicación y ya tiene implementadas algunas características del protocolo. Aunque

todavía no existe una versión funcional, Swirl permite establecer la comunicación entre dos peers.

Una de las preocupaciones actuales es conseguir implementar el sistema Merkle hash tree, que permita desarrollar la parte de integridad en la comunicaciones. Por el momento el equipo de desarrollo está trabajando en una versión de escritorio que está en constante evolución.

Debido a que el PPSP es un protocolo aún demasiado joven y de reciente aparición, no es fácil encontrar implementaciones completas o al menos funcionales del mismo. Esta tarea es más difícil si se intenta encontrar una implementación para el navegador Web.

3.2. BitTorrent

BitTorrent es un protocolo de red para distribución de archivos mediante redes P2P. BitTorrent permite a los usuarios unirse a un swarm para descargar y subir contenido de forma simultánea. Además, permite distribuir grandes ficheros o vídeo en streaming a distintos receptores. Por ejemplo, se podría configurar la descarga en modo secuencial, esperar un tiempo para conseguir el buffer suficiente y comenzar su reproducción.

3.2.1. Entidades

Las entidades de las que está formada un red BitTorrent son:

- **Peer:** Cada uno de los usuarios que forman la red BitTorrent.
- **Leechers:** Se denomina de esta forma a todos los peers que están en la red descargando un determinado archivo pero que aún no se lo ha descargado completamente.
- **Seeders:** Reciben este nombre los usuarios de la red que tienen el archivo completo.
- **Swarm:** Es el conjunto de todos los peers conectados a la red BitTorrent.
- **Trackers:** Son servidores especiales que contienen la información necesaria para que los peers se conecten entre sí. Normalmente sólo hay un tracker por swarm.

3.2.2. Funcionamiento básico

El funcionamiento básico de BitTorrent es el siguiente:

1. Un usuario (peer) *A* crea un archivo `.torrent` y lo distribuye al resto de usuarios interesados en ese contenido. El protocolo BitTorrent es ajeno a esta distribución.

2. *A* hace que el contenido esté disponible en la red mediante un nodo del tipo seeder.

3. El resto de usuarios que quieren el contenido primero deben obtener el fichero `.torrent` y crean otros nodos BitTorrent que actúan como clientes (leechers), intercambiando partes del archivo entre sí. Cuando un cliente distinto de *A* obtiene el archivo completo se convierte en un seeder.

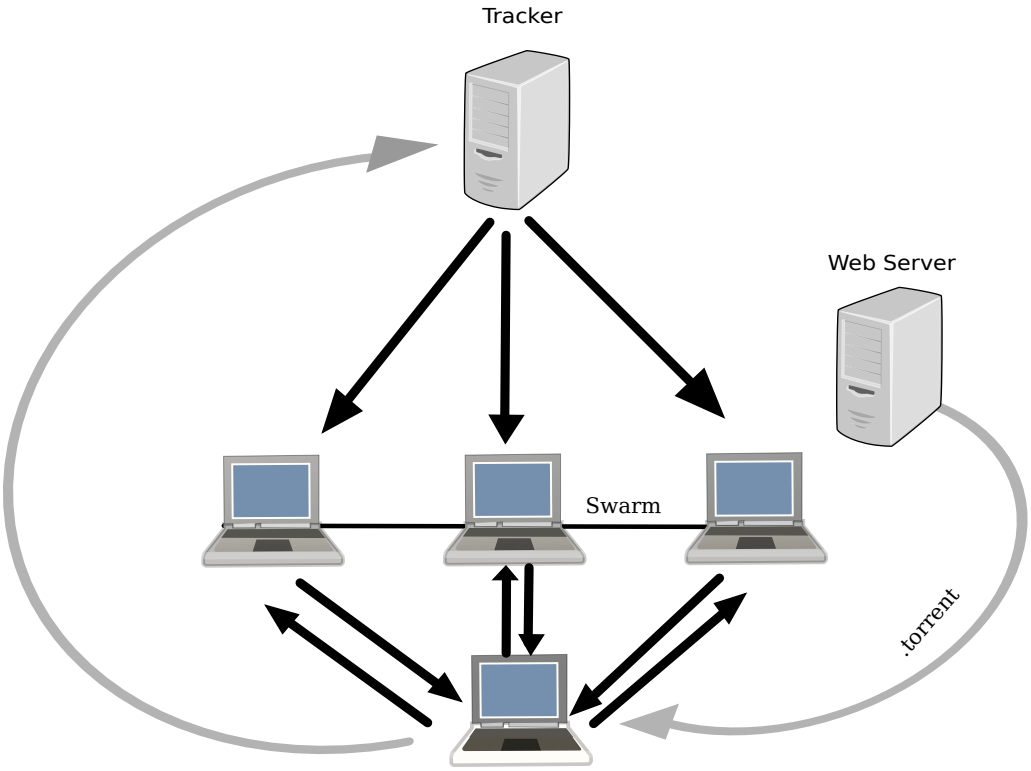


Figura 3.1: Escenario típico de una red BitTorrent.

3.2.3. Implementaciones

Existen multitud de implementaciones del protocolo BitTorrent para descargas de archivos, tanto genéricos como archivos de streaming de contenido multimedia, algunas de las más conocidas son: Vuze [9], uTorrent [10] o Transmision [22]. De hecho, se podría usar un software de descarga de archivos regulares para obtener un fichero de vídeo e ir reproduciéndolo a medida que se está descargando después de dejar un pequeño espacio de tiempo para mantener un buffer en la reproducción.

Sin embargo, al igual que ocurre en la implementación del protocolo PPSP, por el momento no existe una implementación en HTML5.

Por otra parte, el creador del protocolo BitTorrent experimentó a principios de año (2014) con un software que denominaban BitTorrent Live. Dicha aplicación permitía que el navegador reprodujera la transmisión de secuencias de vídeo en directo mediante la instalación de un software adicional (plugin) que se instalaba en la máquina del cliente. Una vez instalado el software, era posible reproducir el stream a tráves del navegador web en HTML5. Desafortunadamente, en la actualidad dicho experimento ya no está disponible habiendo en su lugar un anuncio que indica que ha cambiado el concepto y desde ahora BitTorrent Live será una aplicación para teléfonos móviles inteligentes.

3.3. Streaming P2P en el navegador

En las Secciones 3.1 y 3.2 se ha comentado que no se ha podido encontrar ninguna implementación de los protocolos PPSP y BitTorrent para el navegador Web que funcionen de forma nativa en el navegador sin necesidad de plugins. Esto se debe, probablemente, a que la mayoría de las tecnologías necesarias (ver Capítulo 2) para llevar a cabo este objetivo aún están en proceso de desarrollo estando a medio camino de llegar a convertirse en un estándar. Por otro lado, podrían no llegar a convertirse en estándar nunca. Esto hace que algunas organizaciones o empresas no inviertan en desarrollos de este tipo ya que podrían no llegar a rentabilizar dicha inversión. Llegados a este punto es donde entra el gran potencial de la comunidad de software libre. Dado que la motivación de los desarrolladores no suele estar estrechamente relacionada con asuntos de negocios, es posible encontrar novedosas implementaciones que pueden marcar un hito en el desarrollo determinadas tecnologías, en este caso, el streaming P2P sobre la Web.

Sin embargo, existen empresas que apuestan por desarrollos de este tipo y un claro ejemplo es StreamRoot [23]. Se trata de pequeña startup con sede en París que ha desarrollado una plataforma de streaming basada en HTML5. Esta plataforma es un híbrido de streaming P2P y streaming cliente-servidor que permite a varios navegadores compartir parte del stream si están reproduciendo el mismo contenido multimedia en un instante determinado. Debido a que el sistema es cerrado (entendiéndose como tal que no es de código abierto) no es trivial comprobar el funcionamiento exacto ni qué protocolos intervienen en su funcionamiento. No obstante, en la Web de la empresa aseguran que usan tecnología P2P.

Capítulo 4

Protocolo P2PSP

El P2PSP (Peer to Peer Straightforward Protocol) es un protocolo de comunicación de la capa de aplicación para el streaming de contenido multimedia sobre Internet, donde los usuarios reproducen el stream de forma sincronizada. Esto puede ser usado para construir una variedad de servicios de transmisión en vivo que va desde pequeñas reuniones hasta grandes sistemas de IPTV. A diferencia del tradicional CS (Cliente-Servidor) y CDN (Content Delivery Network) basados en streaming de vídeo, el modelo P2P contribuye con su ancho de banda de subida al sistema. Por esta razón, en general, los sistemas P2P son mucho más escalables y robustos que las arquitecturas basadas en cliente/servidor.

El P2PSP se centra en dos aspectos fundamentales:

1. A pequeña escala, todos los peers se comunican entre sí, todos con todos, utilizando mensajes punto a punto (unicast).

2. En principio y para que el sistema escale suficientemente, todos los peers deben ser tan solidarios con el resto de peers como éstos lo son con él, o de lo contrario, no podrán formar parte de la red P2P ("team" en la jerga P2PSP). Dicha solidaridad se expresa en términos de ancho de banda, y por tanto, esto significa que, para que un peer forme parte de la red, debe de enviar a ésta, en promedio durante cada cierto intervalo de tiempo, tanto como recibe de ella.

El P2PSP ha sido especialmente definido para realizar streaming de secuencias en directo. De esta manera, se puede transmitir a través de una red de comunicaciones basada en la conmutación de paquetes un evento que se está produciendo en ese momento, de la misma manera que se puede difundir eventos ya creados o grabados.

4.1. Características principales

Las principales características que definen el protocolo son:

- **Idependecia del contenido:** El protocolo no comprende, ni trata de analizar, qué está transmitiendo.
- **Arquitectura modular:** El número de módulos depende de los requisitos finales.
- **Simplicidad:** El módulo más básico es muy simple, lo que permite que pueda ser ejecutado en sistemas poco potentes.
- **Multicasting real:** Si IP multicast está disponible, el P2PSP puede usarlo.
- **Soporte para peers "privados":** Los peer pueden estar en redes privadas, incluso detrás de NAT simétricos.
- **Soporte para modelos de transmisión de contenido escalable:** El protocolo es totalmente compatible con multiresolución y las técnicas de streaming adaptativo.
- **Alta integración:** Es compatible con el modelo cliente-servidor, convirtiéndose en un modelo híbrido.

4.2. Entidades

El protocolo P2PSP define los siguientes tipos de entidades:

- **Source (O):** El nodo source se encarga de producir el stream que es transmitido sobre la red P2PSP. Normalmente es un servidor de streaming usando el protocolo HTTP, por ejemplo, Icecast. El source controla el bit-rate de la transmisión en la red.
- **Player (L):** Los players se encargan de solicitar y consumir (descodificar y reproducir) el stream. Normalmente, varios players pueden recibir el stream desde el source, en paralelo, si hay suficiente ancho de banda disponible.
- **Splitter (S):** Esta entidad recibe el stream desde el source, lo divide en trozos del mismo tamaño y los envía a los nodos peers.
- **Peer (P):** Un peer recibe los trozos desde el splitter y desde otros peers, los vuelve a juntar para crear el stream original y lo envía normalmente a un player. Algunos trozos son también enviados a otros peers.

Un escenario típico de un team simple del P2PSP de tamaño 3 (el splitter se excluye para calcular el tamaño del team) se muestra en la Figura 4.1. Nótese que la figura no muestra el player ni el source.

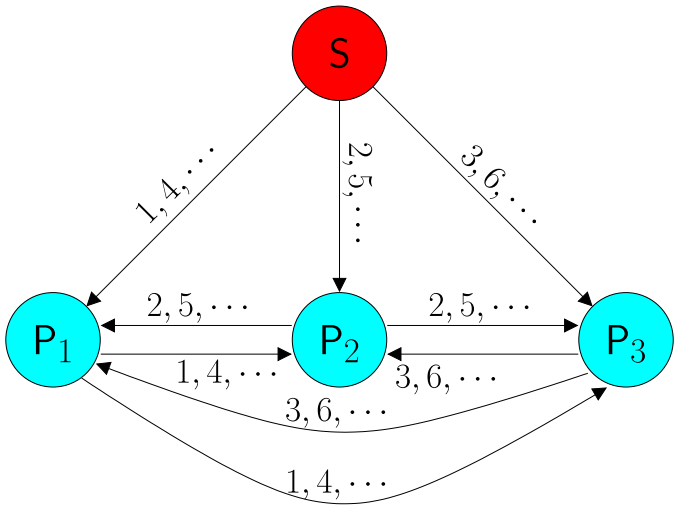


Figura 4.1: P2PSP team. Las flechas y sus etiquetas indican la retransmisión de chunks. S envía un chunk diferente a cada peer que es seleccionado usando un modelo Round Robin. Los peers envían cada chunk recibido de S al resto de peers del team.

4.3. Diseño modular

El protocolo está dividido en varios módulos y cada uno de ellos está definido por un conjunto de reglas que proveen funcionalidades distintas. Sólo es obligatorio un módulo (IMS) para crear un team P2PSP. El resto son opcionales, y en general, no hay dependencia entre ellos. Por tanto, cualquier combinación de módulos es posible.

Los módulos que forman el P2PSP son:

- **IP Multicast Set (IMS) of rules**: Este módulo implementa el comportamiento más básico del protocolo que puede ser usado cuando está disponible IP multicast. Puede ser útil en redes LAN donde este modo de transmisión funciona.
- **Data Broadcasting Set (DBS) of rules**: Este módulo ha sido diseñado para ser eficiente en la transmisión de un stream de datos desde un nodo fuente (source) al resto de peers de la red, cuando no hay disponibilidad IP multicast global.
- **Full-cone Nat Set (FNS) of rules**: El módulo DBS no permite la comunicación entre los peer que están tras un NAT de tipo Full-cone porque los peers inicialmente crean una entrada en sus NATs sólo para el TCP (la lista de peers que es lo primero que reciben los peers desde el splitter se transmite utilizando este protocolo) y el stream se transmite mediante el protocolo UDP. Para crear una entrada en el NAT asociada con el protocolo UDP, usando el mismo punto

final que utiliza la conexión TCP, el peer envía al splitter el mensaje `[Hello]` utilizando UDP justo después de cerrar la conexión TCP con el splitter.

- **Lost chunks Recovery Set (LRS) of rules**: Existe la posibilidad de que un chunk enviado desde el splitter se pierda en la ruta hacia el peer destino. También puede ocurrir que un peer abandone el team sin avisar al resto de peers. En ambos casos, uno o mas chunk no serán recibidos por el resto de peers, lo que significaría una pérdida en la calidad del servicio. Para minimizar el problema, el LRS cuenta con una técnica de recuperación que consiste en reenviar desde el splitter el bloque que se perdió a todos los peers.
- **Adaptative Chunk-rate Set (ACS) of rules**: El módulo DBS obliga a todos los peers a utilizar el mismo ancho de banda de subida. Este módulo relaja esta regla permitiendo que peers con un mayor ancho de banda puedan ayudar en sus tareas a otros peers con un ancho de banda reducido (el splitter siempre transmite una única copia del stream).
- **End-point Masquerading Set (EMS) of rules**: Este módulo se encarga de manejar las situaciones en las que dos o mas peers están detrás de un NAT que realiza enmascaramiento IP (una red privada).
- **NAT Traversal Set (NTS) of rules**: El filtrado NAT es cada vez más frecuente. Este módulo define una funcionalidad extra para manejar peers que están detrás de NAT de tipo restricted-cone o simétricos.
- **Multi-Channel Set (MCS) of rules**: En escenarios donde hay suficiente ancho de banda disponible, los peers pueden decidir suscribirse a más de un stream (canal) al mismo tiempo. Un peer que implementa este módulo puede comunicarse con varios team al mismo tiempo.
- **Data Integrity Set (DIS) of rules**: En algunos casos, la red necesita protegerse de peers malisiosos que podrían envenenar el stream, hacer un ataque de denegación de servicio, etc. Estos peers serán identificados y expulsados del team gracias a este módulo.
- **Data Privacy Set (DPS) of rules**: Contiene una colección de reglas que aseguran que el stream transmitido es reproducido sólo por peer legítimos. Puede ser útil para implementar, por ejemplo, servicios de pay-per-view.

4.3.1. El módulo DBS (*Data Broadcasting Set of rules)*

Este conjunto de reglas define los algoritmos y los métodos de comunicación necesarios para una red P2PSP cuando IP multicast no está disponible. En el contexto de uso de los navegadores Web, esta va a ser la situación más probable.

1. **Manejo de chunks:** Los chunks son transmitidos desde el splitter hasta el/los peer(s), que después se encargarán de repartírselos entre ellos. EL splitter envía

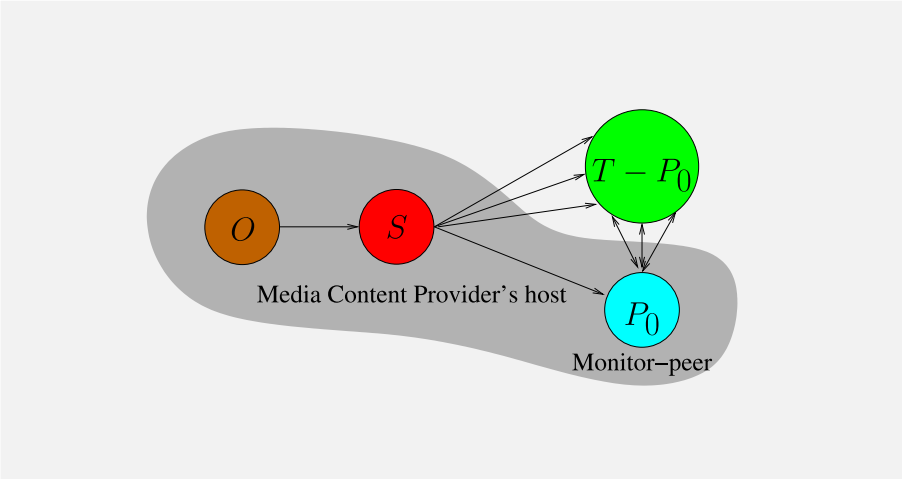


Figura 4.2: Configuración del P2PSP usando un peer monitor. Nótese que el peer monitor, el source y el splitter se están ejecutando en el mismo host.

el n-ésimo chunk al peer P_i si

$$(i + n) \text{ mód } |T| = 0, \tag{4.1}$$

donde $|T|$ es el número de peers en el team. Después, P_i debe reenviar este chunk al resto de peers en el team. Los chunks recibidos desde otros peers no son retransmitidos.

2. **La lista de peers:** Cada nodo del team (splitter y peers) conocen el punto extremo P (dirección IP y puerto) del resto de peers en el team. Se construye una lista con esta información que más tarde será usada por el splitter para enviar los chunks a los peers y por los peers para reenviar los chunks recibidos al resto de peers.

3. **Control de solidaridad (free-riding) en los peers:** Cada peer P_i asigna un contador al resto de peers del team. Cuando un chunk se envía a un peer P_j, su contador se incrementa y cuando un chunk recibe desde el mismo, el contador se decrementa. Si dicho contador alcanza un límite, el peer insolidario se elimina de la lista de peers de P_i, y por tanto, no le volverá a enviar más chunks.

4. **Entrada de peers:** Un nuevo peer que quiera unirse al team debe primero ponerse en contacto con el splitter. Después, el splitter envía al nuevo peer la lista actual de peers en el team y la cabecera del stream sobre TCP.

5. **Peers monitores:** Un peer monitor (ver P_0 en Figura 4.2) suele ejecutarse cerca del splitter y toma diferentes roles dependiendo de los módulos que se implementan. Entre otros:

a) Lo usa el administrador del team para monitorizar la sesión de streaming, ya que, si el contenido se puede reproducir correctamente en el peer monitor probablemente el resto del team también pueda reproducir el contenido correctamente.

b) Al menos un peer monitor se crea antes que cualquier otro peer en el team. El ratio de transmisión del primer peer monitor es 0. Sin embargo, el ratio de transmisión del segundo peer, es:

$$B/2,$$

donde B es la media del ratio de codificación del stream. Cuando el tamaño del team es $|T|$, el ratio de transmisión de todos los peers (incluido el peer monitor) del team es:

$$B\frac{|T|}{|T|+1}. \tag{4.2}$$

Por tanto, gracias al peer monitor ningún peer que entra posteriormente en el team lo hace sin enviar chunks. Nótese que

$$\lim_{|T|\to\infty} B\frac{|T|}{|T|+1} = B, \tag{4.3}$$

lo que significa que cuando el team es suficientemente grande, los peers tienden a transmitir tanto como reciben.

c) Para minimizar el número de quejas por pérdidas en el team, los peers monitores son las únicas entidades que puede quejarse al splitter de los chunks que se han perdido.

6. **Salida de peers:** Los peers tienen que enviar un mensaje de despedida `[Goodbye]` al splitter cuando dejen el team, de forma que el splitter pueda dejar de enviarles chunks tan pronto como sea posible. Sin embargo, si un peer P_i abandona el team sin notificarlo previamente al splitter, no se recibirán más chunks desde dicho peer. Esto debería lanzar la siguiente sucesión de eventos:

   a) En el resto de peers, el algoritmo que evita el free-riding eliminará P_i de la lista de peers.

   b) Todos los peer monitores se quejarán al splitter de los chunks que el splitter ha enviado a P_i.

   c) Después de recibir un número suficiente de quejas, el splitter eliminará a P_i de su lista.

7. **Seguimiento de Chunks en el splitter:** Para identificar a los peers insolidarios (free-riding), el splitter recuerda el número de chunks enviados a cada peer entre los últimos b chunks. Sólo los peers monitores se quejarán al splitter de los chunks perdidos usando mensajes de queja `[lost chunk number]`. En el módulo DBS, un chunk se considera como perdido cuando llega el momento de enviarlo al player y el chunk no está disponible.

8. **Control del free-riding en el splitter:** En el módulo DBS es obligatorio que los peers contribuyan al team con la misma cantidad de datos que ellos reciben del team (siempre bajo las condiciones impuestas por la Ecuación 4.3). Para garantizar esto, el splitter cuenta el numero de quejas (enviadas por los peers monitores) que un peer produce. Si este número excede un límite establecido, entonces el peer insolidario será expulsado del team (primero se elimina de la lista del splitter y después de la lista de todos los peers del team).

9. **Evitar la congestión en los peers:** Cada peer envía chunks usando una estrategia de bit-rate constante para minimizar la congestión del enlace de subida. Nótese que el ratio de chunks que llega a un peer es una buena métrica para llevar a cabo este control en la red con un ratio de pérdida de paquetes razonablemente bajo.

10. **Modo ráfaga en los peers:** El modo usado para evitar la congestión en los peers es abandonado inmediatamente si un nuevo chunk se recibe desde el splitter antes de que el anterior chunk sea retransmitido a todos los peers del team. En el modo ráfaga el peer envía el último chunk recibido por el splitter al resto de peers del team. En otras palabras, el peer envía el último chunk recibido al resto del team tan rápido como sea posible. Nótese que, si bien este comportamiento es una potencial fuente de congestión, es esperable que sólo un pequeño número de chunks sean enviados en el modo ráfaga en virtud de la baja relación de pérdida de paquetes.

Capítulo 5

Propuesta

5.1. El navegador Web

El navegador Web es una herramienta indispensable en cualquier equipo porque es la forma más común de acceder a la información disponible en Internet para un usuario. Esto se debe a que el navegador es fácil de usar, y para acceder a un recurso es suficiente con recordar el nombre de dominio donde se aloja o llegar hasta él mediante motores de búsqueda disponibles en la red.

Por otra parte, después de analizar la tecnologías Web disponibles (ver Capítulo 2) es fácil llegar a la conclusión de que HTML5 es una opción ideal para implementar un cliente P2PSP. Una implementación en HTML5 permite a cualquier usuario conectarse a un team P2PSP y comenzar a recibir (y compartir) el contenido multimedia. Aunque HTML5 incluye la etiqueta `<video>` para incrustar el contenido multimedia, esto no es suficiente para ejecutar un cliente P2PSP porque para ello necesitamos hacer uso de protocolos de transporte ligeros.

Por otra parte, HTML5 nos brinda la oportunidad de usar sockets TCP para permitir una comunicación bidireccional entre el navegador y el servidor en distintos puertos gracias a WebSocket (Sección 2.1.4). Sin embargo, existen dos inconvenientes en esta solución: (1) no es posible comunicar directamente los navegadores y (2) la transmisión de bloques de vídeo necesitamos realizarla sobre UDP por motivos de eficiencia y escalabilidad.

Por suerte, este problema es posible solventarlo haciendo uso de una nueva API que ahora mismo está en proceso de estandarización por el W3C y la IETF. Concretamente, dicha API, que se denomina WebRTC (véase la Sección 2.4), es ideal para nosotros porque define el recurso denominado interface `DataChannel` que permite establecer un canal de datos bidireccional entre los navegadores mediante socket sobre UDP.

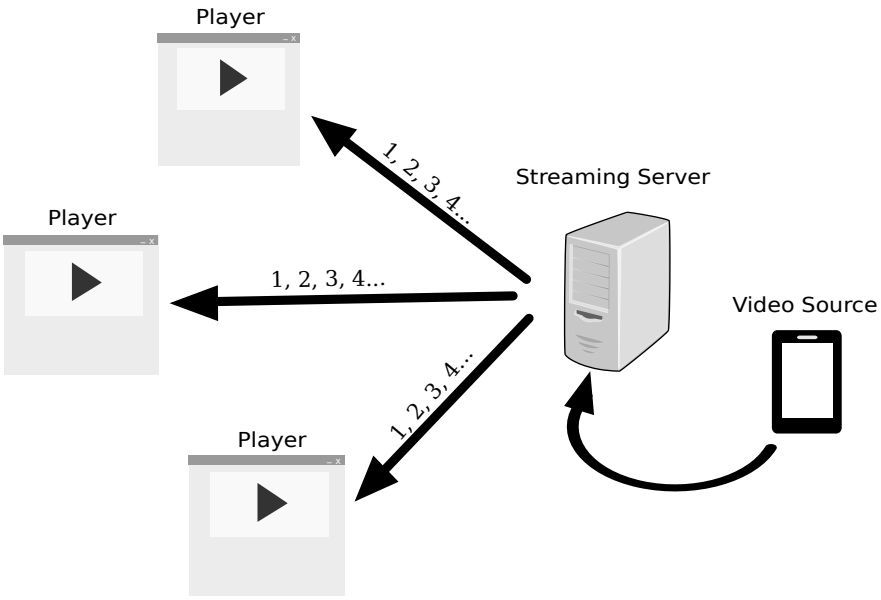


Figura 5.1: Un escenario típico de streaming en vivo usando el modelo cliente-servidor. La fuente de vídeo (en este caso el dispositivo móvil) captura el vídeo y lo envía al servidor de streaming. Cada cliente recibe una copia del stream desde el servidor de streaming quién envía el contenido tantas veces como clientes hay conectados.

5.2. Streaming P2PSP en el navegador

La Figura 5.1 representa un ejemplo de un sistema cliente-servidor que transmite el vídeo capturado por la cámara del teléfono móvil a tres clientes. En ella se pueden identificar los siguientes elementos principales:

1. **Video Source:** Dispositivo que genera el vídeo y lo envía al Streaming Server (servidor de streaming).
2. **Streaming Server:** Un proceso corriendo en un host accesible desde Internet que se encarga de reenviar el vídeo en tiempo real.
3. **Player:** Un navegador compatible con HTML5. La página Web que el navegador se descarga puede estar alojada en cualquier servidor Web de Internet. Sólo tiene que obtener el stream desde el servidor de streaming.

La API WebRTC y el protocolo P2PSP se usan al unísono para desarrollar un sistema de streaming más eficiente basado en los sistemas IPTV representados en la Figura 5.1 [19]. Más específicamente, permite ejecutar los peers P2PSP en el navegador Web y usar la interface `DataChannel` de WebRTC como protocolo de transporte.

La Figura 5.2 muestra una configuración similar a la de la Figura 5.1, pero en este modelo, el Streaming Server envía una única copia del stream. Para ello divide el vídeo en trozos y como si de una baraja infinita (vídeo) de cartas se tratase, el servidor va repartiendo las cartas (trozos de vídeo) de una en una entre los clientes. Los clientes son los encargados de repartirse dichos trozos para que finalmente todos puedan tener el vídeo completo, siguiendo la misma filosofía que los sistemas P2P convencionales para el intercambio de archivos. Para ello necesitamos añadir los siguientes elementos:

1. **Splitter + Signaling Server:** Recibe el vídeo desde el Streaming Server (sobre HTTP) y lo envía al resto del team.
2. **Player + Peer:** Recibe, comparte y reproduce el stream. Las funciones principales son: (1) descargar la lista de peers desde el Splitter + Signaling Server y (2) establecer una comunicación P2PSP con el resto de Player + Peers.

Básicamente, las ventajas de este nuevo sistema propuesto son:

1. Al igual que en el modelo Cliente-Servidor, **los usuarios sólo necesitan descargarse una página Web** para empezar a disfrutar del contenido. Sin plugins o software adicional. Gracias a WebRTC, ahora también es posible hacer esto con conexiones peer to peer.
2. **Los clientes aportan su capacidad de subida** con el fin de hacer el sistema mucho más escalable.
3. **El modelo Cliente-Servidor y el modelo P2P no son incompatibles.** Por ejemplo, un usuario premium (que no estaría obligado a compartir ancho de banda con el team) podría recibir el stream directamente del Streaming Server usando el modelo Cliente-Servidor, tal y como se muestra en la Figura 5.3.

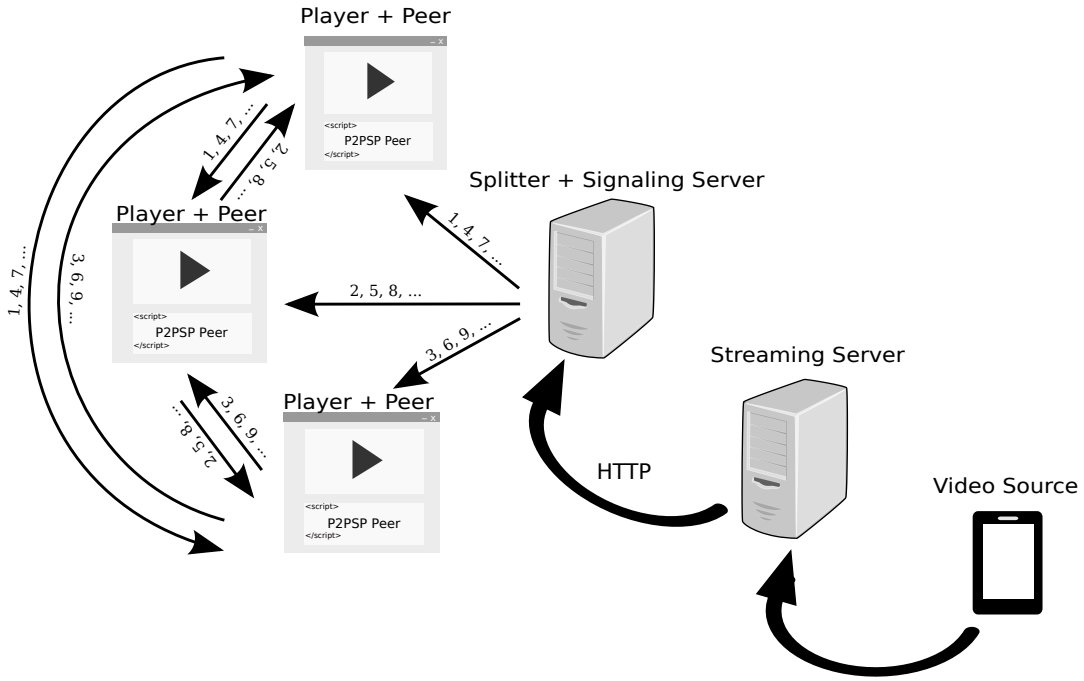


Figura 5.2: Un escenario equivalente al de la Figura 5.1 pero usando el P2PSP. Cabe destacar que en este caso el Streaming Server sólo envía una copia del stream. Al igual que éste, el Splitter sólo envía una copia del stream y los peer son los encargados de compartir el contenido entre ellos.

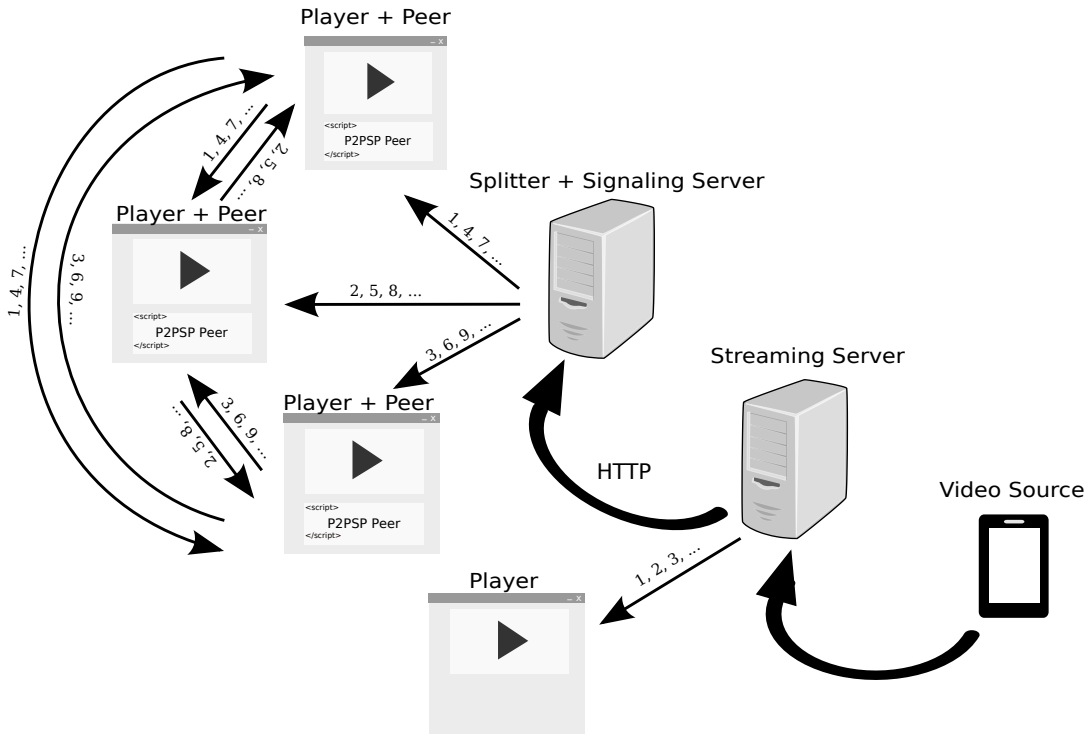


Figura 5.3: Un escenario similar al de la Figura 5.2 pero en este caso se ha añadido un player premium que se conecta directamente al Streaming Server. Nótese que este usuario no tiene que compartir su ancho de banda de subida con el resto de usuarios.

Capítulo 6

Implementación

La implementación del P2PSP usando WebRTC consistirá en desarrollar los elementos básicos del protocolo P2PSP, de forma que pueda ser ejecutado en el navegador Web del usuario sin necesidad de instalar plugins u otro software adicional. El objetivo de esta implementación es confirmar la teoría de que una implementación de este tipo (al menos con las funciones básicas) es posible hoy en día gracias a la situación actual de las tecnologías Web. Por tanto, se llevará a cabo la implementación del módulo *Data Broadcasting Set of rules* en la forma que sea posible y se intentarán salvar las barreras que la tecnología Web impone dando prioridad a obtener una versión funcional en la que varios peers puedan unirse a un team para consumir (y compartir) contenido multimedia al mismo tiempo.

Puesto que todas las tecnologías expuestas hasta este punto son compatibles con cualquier plataforma, se mantendrá en mente un desarrollo responsive que permita a la aplicación ejecutarse y adaptar su interfaz a entornos multidisciplinares, tales como: ordenadores, smartphone o tablet. Además, y puesto que se prevé que la API WebRTC evolucione en todos los navegadores en un futuro muy cercano, se tendrá en cuenta este punto para hacer un desarrollo fácilmente adaptable a estos posibles cambios.

6.1. Splitter + Signaling Server

Como se ha visto en capítulos anteriores, uno de los elementos básicos y necesarios para llevar a cabo una implementación del P2PSP es la entidad `Splitter`. El `Splitter` será el encargado de recibir el stream, dividirlo en chunks del mismo tamaño y enviarlo al team, pero además tiene otra función, la de presentar cada peer entrante al resto del team. Ésta última capacidad es similar a la que debe llevar a cabo el `Signaling Server` en WebRTC durante el proceso de señalización. Los dos procesos son obligatorios, el primero, para cumplir la especificación del protocolo P2PSP y el segundo, para iniciar una `PeerConnection` en WebRTC. Por tanto, parece lógico pensar en fusionar ambas entidades en una, de modo que se ahorran recursos evitando que ambas entidades hagan dos funciones tan parecidas.

6.1.1. Python

Python es el lenguaje escogido para desarrollar la parte servidora de la versión para el navegador Web del P2PSP. Se ha elegido este lenguaje por dos sencillas razones: (1) Python es un lenguaje potente, fácil de usar y open source y (2) una versión funcional del P2PSP para escritorio está escrita en este lenguaje en Launchpad [24], lo que permite hacer un branch de dicha versión y aplicar los avances de desarrollo en todas las ramas.

En la versión de escritorio del P2PSP los peers se conectan por primera vez con el splitter mediante socket haciendo uso del protocolo TCP. Como ya se ha visto en el Capítulo 2, HTML5 ofrece una alternativa similar, los denominados WebSocket. Gracias a WebSocket se puede llevar a cabo un proceso similar al realizado en la versión de escritorio. La única pega en esta solución es que actualmente el protocolo WebSocket no está implementado en las librerías estándar del lenguaje de programación Python.

Sin embargo, una de las ventajas del software libre, es que dependiendo de la licencia, es posible incorporar mejoras a tu trabajo partiendo de conocimientos de otros desarrolladores de la comunidad, del mismo modo que cualquiera puede incorporar mejoras a tu propio software u obtener ideas del código desarrollado. Tras algunas búsquedas y comparativas en la red sobre librerías de código abierto para Python que implementen el protocolo WebSocket se ha decidido incorporar a este proyecto la librería `SimpleWebSocketServer` [4], una librería de software libre sin adornos, fácil de entender, que simplemente hace lo que tiene que hacer, y lo hace bien.

A continuación se explicará detalladamente cómo se han implementado las distintas funciones del Splitter+Signaling Server.

6.1.2. Extracción de la cabecera del nodo source

El stream es servido al splitter por el source. Según está definido en el protocolo, el source suele ser un servidor de streaming HTTP, por ejemplo, Icecast. Sin embargo, debido a problemas con el códec de vídeo y Media Source Extensions no ha sido posible obtener el contenido directamente de un servidor de streaming, por tanto, en esta implementación el source será un fichero de vídeo. No obstante, no se descarta que muy pronto esto sea posible gracias a las constantes actualizaciones que en los últimos días se están haciendo de los navegadores Web. En este caso, la obtención de la cabecera de vídeo consiste en abrir el fichero en modo lectura y obtener los primeros n bytes de información, donde n será un número de bytes suficiente que contenga la cabecera.

```
1 file=open("test.webm","rb")
2 header = file.read(n)
```

Código 6.1: Obtener cabecera desde fichero.

6.1.3. Entrada y salida de peers

Cuando un peer quiere unirse al team es necesario que se lo haga saber al splitter y para ello se seguirá el proceso de señalización que se ha descrito en la Sección 2.4.5 mezclado con el proceso de entrada de peers del protocolo P2PSP descrito en la Sección 4.3.1. El splitter recibirá una petición de conexión desde el peer a la que contestará con la lista de peers que actualmente existe en el team y con la cabecera del vídeo.[1]

```
1 #Se envia la lista de peers
2 self.sendMessage(str('{"peerlist":"'+str(peerlist)+'"}'))
3
4 #Se envia la cabecera del stream
5 message=struct.pack("H1024s", 0, header)
6 self.sendMessage(buffer(message))
```

Código 6.2: Envio de lista de peers y cabecera desde el splitter.

Una vez que el splitter ya ha enviado la cabecera del stream y la lista de peers, comienza el proceso de señalización para presentar el peer al resto del team. Este proceso, como se mostrará más adelante, es iniciado por el peer. El splitter recibirá un mensaje SDP dirigido a un peer concreto y la función del splitter es hacerle llegar el mensaje al peer destino (resto de peers en el team). Puesto que el splitter conoce tanto el remitente como el destinatario, lo único que tiene que hacer es reenviar el mensaje, si necesidad de interpretarlo. Del mismo modo actúa cuando recibe un mensaje de tipo ICE (Ver Código 2.9) con la información de los candidatos.

```
1 peeridlist[int( decoded['idreceiver'].replace('"','') ...
      )].sendMessage( str(self.data) )
```

Código 6.3: Reenvio de mensajes SDP e ICE.

6.1.4. División del stream y envío de chunks al team

El splitter alimenta al team haciendo uso de la planificación *Round Robin* para enviar los chunks, es decir, envía un chunk distinto a cada peer comenzando por el primer elemento de la lista de peers hasta llegar al último. Una vez que alcanza el último elemento de la lista repite el ciclo desde el primer elemento.

Para obtener el chunk, el splitter lee los siguientes m bytes del fichero, siendo m un parámetro configurable que representa el tamaño de los chunks y envía el chunk al peer correspondiente.

[1] Nótese que todos los envíos vía WebSocket se realizan serializando el objeto mediante JSON.

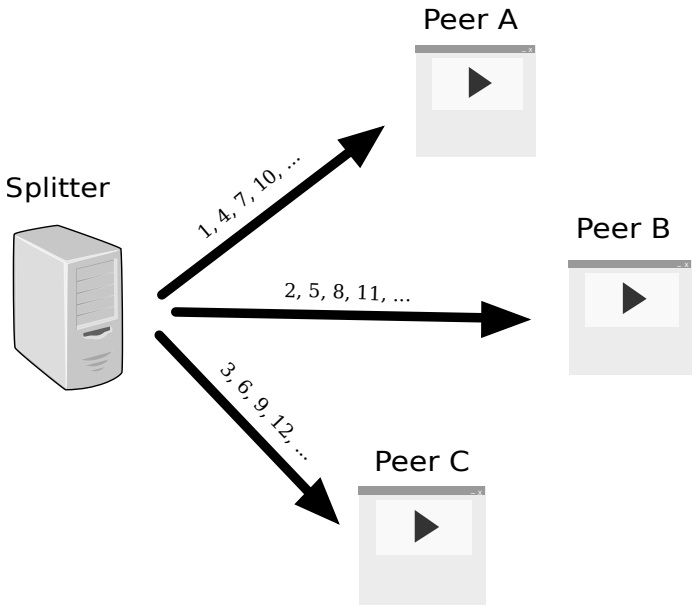


Figura 6.1: Envío de chunks con planificación Round Robin.

```
1  def nextChunk(client):
2      global numblock
3      global file
4      try:
5              chunk = file.read(m)
6          numblock=numblock+1
7          message=struct.pack(chunk_format_string, numblock, ...
               chunk) # H1024s
8          client.sendMessage(buffer(message))
9
10     except Exception as n:
11         print "Unknow error in nextChunk: "+n
12         chunk = None
13         file.close()
14
15     return chunk
16
17 def feedCluster(client):
18
19     chunk=nextChunk(client)
20
21     while chunk is not None:
22         chunk=nextChunk(client)
23
24     file.close()
```

Código 6.4: División del stream y envío de chunks.

6.2. Peer + Player

En la implementación para el navegador Web, el peer, además de su función básica de recibir y reenviar chunks a otros peers tendrá una función añadida, la de reproducir el stream. La finalidad es que el usuario sólo tenga que abrir una instancia del navegador Web para disfrutar del contenido embebido directamente en la página Web que está visitando. Debido a su doble funcionalidad, esta entidad puede recibir indistintamente el nombre de *Peer* o *Player*.

A diferencia del splitter y por la naturaleza de la aplicación, esta implementación no se realizará en el lenguaje python sino que será una implementación que implicará el uso de varios lenguajes de programación para la Web. La interfaz gráfica se realizará mediante el uso de HTML5 y CSS3. Para la lógica de la aplicación se usará el lenguaje de programación JavaScript.

Para compartir los chunks entre los navegadores Web necesitamos comunicación directa entre los peers. En la versión de escritorio esto se consigue gracias al protocolo UDP, pero esto no es tan sencillo para la versión Web. En este caso necesitamos, como ya se apuntaba en el Capítulo 5, hacer uso de WebRTC que permite precisamente esta funcionalidad. Esta nueva tecnología nos permitirá crear un canal de comunicación directo entre los distintos navegadores usando un protocolo de transporte ligero conocido como SCTP.

A continuación se explicará detalladamente cómo se han implementado las distintas funcionalidades del Peer + Player.

6.2.1. Interfaz

Un diseño responsive

Unificar la Web, ese es el objetivo que se debe mantener en mente cuando se comienza un nuevo proyecto para la Web. El término *Responsive Web Design* apareció por primera vez de la mano de Ethan Marcotte [16]. La filosofía es hacer un diseño adaptable al entorno del usuario mediante hojas de estilo CSS y dejar atrás la idea de crear aplicaciones Web distintas dependiendo del entorno en el que se ejecutará. Así pues, se consigue que una única implementación esté optimizada y sea capaz de adaptarse a los diferentes dispositivos usados actualmente: PCs, tablet, teléfonos móviles, etc.

Este proyecto aplica la filosofía responsive para llevar a cabo la implementación de la interfaz. Para ello, se ha creado una hoja de estilos CSS que define el aspecto de los diferentes elementos gráficos.[2] Para el tamaño de las capas u otros elementos que ocupan un espacio físico se usa el porcentaje y para las fuentes se usan tamaños relativos estándares. Un ejemplo se muestra en Código 6.5.

[2]Por ejemplo, se ha evitado dar valores absolutos de modo que todos los elementos tienen valores relativos.

```
.super{
    background-color: #252D35;
    color: #FFF;
    height: 20%;
}
.title{
    width: 80%;
    font-size: x-large;
    margin-left: auto ;
    margin-right: auto ;
    text-align: center;
}
```

Código 6.5: Fragmento de hoja de estilos CSS del proyecto.

Layout

La palabra que mejor define el diseño y la distribución de los elementos en la aplicación es: sencillez. Un diseño sencillo, limpio y minimalista permite al usuario centrarse en su objetivo principal y alejarse de elementos sobrantes que lo único que consiguen es distraer.

Por otra parte, como ya se ha indicado anteriormente, CSS se utiliza para especificar las características de los elementos, pero estos deben estar previamente definidos. Para esta tarea se hace uso de HTML5. Se definen varias capas (`div`) que especifican zonas que contienen elementos. Básicamente esta implementación cuenta con un elemento principal: el reproductor de vídeo. Para insertar el reproductor se hace uso de la etiqueta `<video>`, y a ésta se añaden características como los controles de vídeo y un póster que se muestra cuando no hay nada para reproducir.

```
<video poster="imgs/p2pspposter.png" controls autoplay ...
   id="player" width="80%" height="80%"></video>
```

Código 6.6: Reproductor en HTML5.

La Figura 6.2 muestra la interfaz de la aplicación. En ella se observa cómo se centra principalmente en el reproductor de vídeo ocupando la mayor parte de la pantalla. Para interactuar con la aplicación el usuario dispone de un único botón situado bajo el reproductor, al pulsarlo, se inicia todo el proceso y comienza la reproducción del stream. Adicionalmente es posible pausar la reproducción o avanzarla dependiendo del contenido que haya sido descargado hasta el momento. Para salir de la reproducción (abandonar el team) basta con cerrar el navegador Web.

6.2.2. Lógica de la aplicación

El peer necesita conocer cuál va a ser su canal de señalización, es decir, donde está ubicado el splitter. Además, debido al uso de la API WebRTC y tal como se

Figura 6.2: Interfaz de la aplicación Web.

establece en la especificación de la misma, es necesario definir también a qué servidor `TURN/STUN` se solicitará la dirección externa del peer. Una vez que el peer conoce esta información ya está listo para comenzar a formar parte de un team P2PSP.

```
1  var signalingChannel = new WebSocket("ws://127.0.0.1:9876/");// ...
       <-- Your splitter.py IP:PORT here
2  var configuration = {iceServers: [{ url: ...
       'stun:stun.l.google.com:19302' }]};// <-- Your STUN IP:PORT ...
       here
```

Código 6.7: Indicar la URL del splitter y del servidor STUN.

Para explicar la implementación de esta parte de la aplicación se describirá el proceso básico que se realiza y los elementos que intervienen desde que el usuario pulsa el botón `Initiate Connection` hasta que comienza a reproducir y compartir el stream.

Entrada al team

Cuando el usuario pulsa el botón `Initiate Connection` el peer se comunica inmediatamente con el splitter, quién le contesta con la lista de peers que actualmente forman parte del team y la cabecera del stream. El peer necesita distinguir el tipo de mensaje que ha recibido ya que podría ser un chunk o un mensaje del proceso de señalización (SDP o ICE). Dependiendo del tipo de mensaje que reciba actuará de una u otra forma.

En este caso, el peer recibe en primer lugar la lista de peers. Su misión es almacenar dicha lista para conocer quién forma el resto del team. Inmediatamente después, el peer recibe del splitter la cabecera del stream que se está reproduciendo en el team en ese momento. El peer, por su parte, envía la cabecera al reproductor que la almacenará en su propio buffer.

Señalización

Como cualquier aplicación que use la API WebRTC, la implementación del P2PSP para el navegador debe seguir el proceso de señalización, en este caso mediante el splitter. Nótese además que, aunque se use el splitter como `Signaling Server`, es necesario seguir el protocolo para el descubrimiento de peer establecido por la especificación de la API WebRTC. Por tanto, el peer se encargará de generar mensajes SDP de tipo `Offer` y `Answer` para conseguir establecer la comunicación mediante `PeerConnection` con todos y cada uno de los peers del team. El proceso de señalización se describe con más detalle en la Sección 2.4.5.

```
1  pcs[i] = new webkitRTCPeerConnection(configuration, {optional: []});
2
3  // send any ice candidates to the other peer
4  pcs[i].onicecandidate = function (evt) {
5      if (evt.candidate){
6          signalingChannel.send(JSON.stringify({ "candidate": ...
               evt.candidate , "idtransmitter":'"'+idpeer+'"', ...
               "idreceiver":'"'+i+'"'}));
7      }
8  };
9
10
11 // let the "negotiationneeded" event trigger offer generation
12 pcs[i].onnegotiationneeded = function () {
13     pcs[i].createOffer(function(desc){ localDescCreated( desc, ...
           pcs[i], i);});
14     console.log("Create and send OFFER");
15 }
```

Código 6.8: Crear PeerConnection y enviar Offer/ICE.

Cuando un nuevo peer aparece, crea una `PeerConnection` por cada uno de los peers existentes en la lista de peers, a los que enviará un mensaje SDP de tipo

`Offer` que contiene su descripción local `localDescription`. Por su parte, el resto de peers (que ya forman parte del team) almacenarán el mensaje `Offer` como `remoteDescription` y responderán con su descripción local mediante un mensaje SDP de tipo `Answer`. Una vez que ocurre este intercambio de mensajes SDP, se inicia el intercambio de mensajes ICE que contiene los puntos extremos de cada peer obtenidos a través del servidor TURN/STUN.

```
1  if (message.sdp && idreceiver==idpeer){
2          //console.log(message.sdp);
3      pcs[id].setRemoteDescription(new ...
           RTCSessionDescription(message.sdp), function () {
4              console.log("remoteDescription is Set");
5              // if we received an offer, we need to answer
6              if (pcs[id].remoteDescription.type == "offer"){
7                          console.log("Create and send ANSWER");
8                      pcs[id].createAnswer(function(desc){ ...
                           localDescCreated(desc, pcs[id], id);});
9                  }
10         });
11 }
12
13 if (message.candidate && idreceiver==idpeer){
14     console.log("Received ice candidate: "+ ...
           message.candidate.candidate);
15     pcs[id].addIceCandidate(new RTCIceCandidate(message.candidate));
16 }
```

Código 6.9: Establecer Offer y enviar Anwser.

Reproducción del stream

Una vez se ha llevado a cabo la señalización, el peer comenzará a recibir chunks desde el splitter y desde el resto de peers que forman parte del team. El peer añade cada chunk recibido a un buffer en la posición correspondiente. Dicha posición se indica en los dos primeros bytes de cada chunk y corresponde con el identificador temporal del chunk. Además, el identificador indica el orden de reproducción de los mismos.

```
1 buffer[numblock%buffer_size]=chunk;
```

Código 6.10: Almacenar chunk en buffer.

Antes de este proceso es necesario establecer el tipo de stream que se enviará al reproductor. Esto es posible gracias a Media Source Extensions. Los tipos de códec compatibles con cada navegador se pueden ver en la Tabla 2.2.

```
1 var mediaSource = new MediaSource();
2 ...
```

```
3 var sourceBuffer = mediaSource.addSourceBuffer('video/webm; ...
      codecs="vorbis,vp8"');
```

Código 6.11: Definir el formato del stream.

Cuando el buffer tiene al menos la mitad de los chunks, se comienza a enviar el contenido al reproductor añadiendolos al buffer del objeto `MediaSource`.

```
1 var chunk = new Uint8Array((buffer[current]).slice(2));
2 current=(current+1)%buffer_size;
3 mediaSource.sourceBuffers[0].appendBuffer(chunk);
```

Código 6.12: Enviar chunks al reproductor.

Reenvío de chunks al Team

Una parte fundamental para el funcionamiento del sistema es que el peer, además de alimentar al reproductor, reenvíe todos los chunks que recibe del splitter al resto de peers. Nótese que los chunks recibidos desde otros peers no son reenviados. La tarea de reenviar los chunks es fácil para el peer ya que conoce a todos los peers del team gracias a la lista de peers que recibió del splitter y que posteriormente ha ido ampliando con la llegada de cualquier nuevo peer.

Más concretamente, en caso de que llegue un nuevo peer, se añadirá a la lista de peers en el momento que se reciba un mensaje SDP `Offer` o un mensaje ICE del nuevo peer. Para ello, se comprueba si existe un objeto `PeerConnection` para el nuevo peer, en caso de que no exista, se añade a la lista de peers y se crea el objeto `PeerConnection` asociado.

```
1 if (!pcs[id]) {
2   console.log('%cCreate a new PeerConection','background: #222; ...
        color: #bada55');
3   peerlist.push(id); //Se agrega a la lista de peers
4   console.log("PEER LIST UPDATE: "+peerlist);
5   ...
6   pcs[id] = new webkitRTCPeerConnection(configuration, ...
        {optional: []}); //se crea el PeerConnection asociado.
7   ...
8 }
```

Código 6.13: Actualizar lista de peers.

Nótese que algunos elementos de la API se escriben con un prefijo delante. En este caso, el prefijo que usa Google Chrome es *webkit*. Este prefijo se mantiene en la implementación del navegador mientras la API no sea un estándar. Otros navegadores como Firefox utilizan el prefijo *moz*.

El proceso de reenvío de chunks se lleva a cabo a través del objeto `DataChannel`, es decir, el intercambio de datos no se realiza mendiante el splitter (signaling chan-

nel). El intercambio se realiza de navegador a navegador mediante el protocolo SCTP de `DataChannel` y se lleva a cabo con la configuración en modo `unrealible`, similar a UDP, para conseguir la menor latencia posible.

```
1  ...
2  channel[i] = pcs[i].createDataChannel("cha"+i, dataChannelOptions);
3  ...
4  function sendChunk(chunk) {
5    for (i in peerlist){
6      if (peerlist[i]!=idpeer){
7        //console.log("send to "+peerlist[i]);
8        try{
9          channel[peerlist[i]].send(chunk);
10       }catch(e){
11         console.log(i+" said bye!");
12       }
13     }
14   }
15 }
```

Código 6.14: Reenvio de chunks al resto de peers.

La Figura 6.3 muestra la secuencia completa de mensajes que se intercambian entre los servidores y los peers desde la llegada de un peer hasta que se comunican directamente entre sí y reciben el stream.

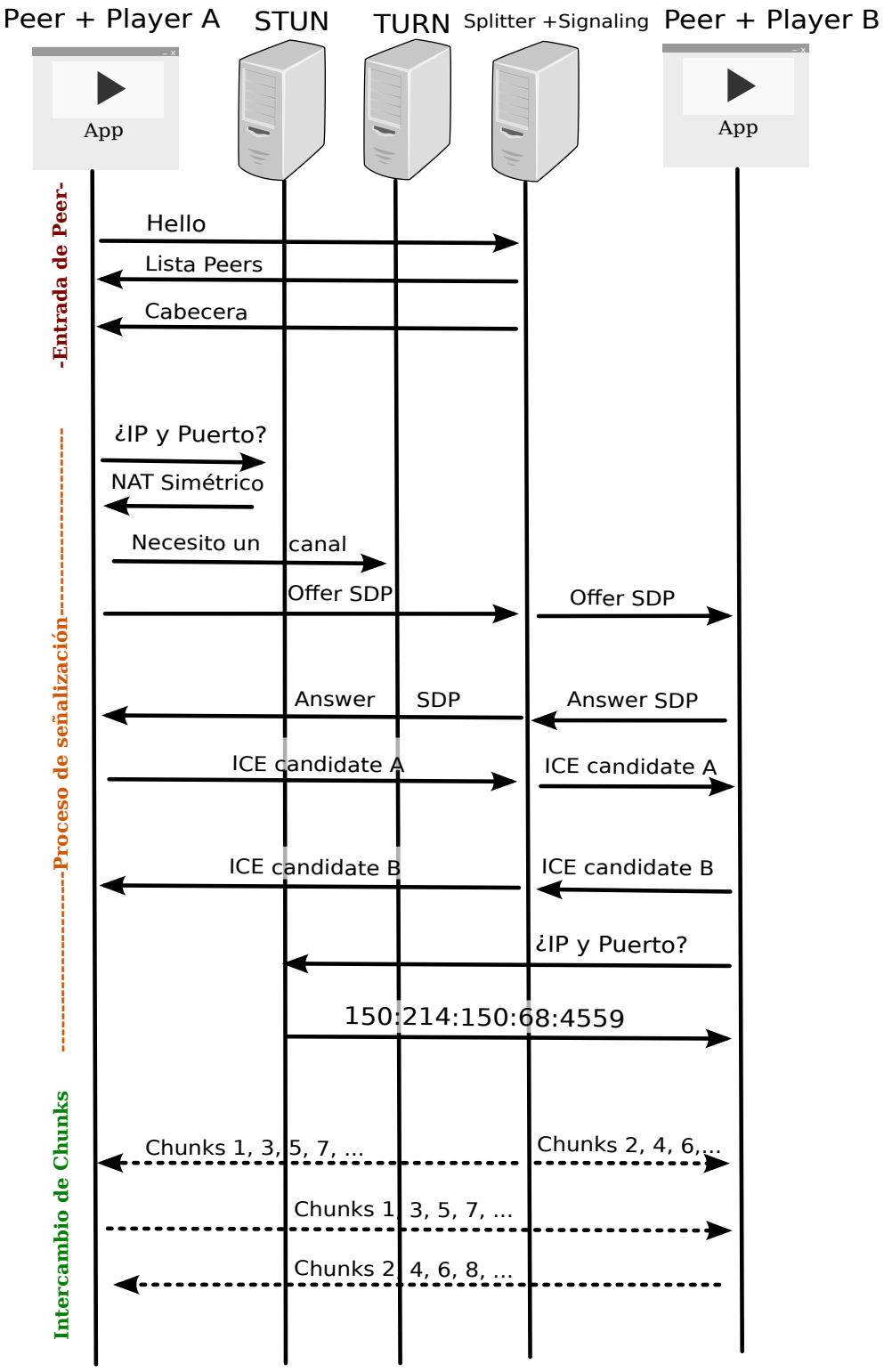


Figura 6.3: Secuencia de intercambio de mensajes a la llegada de un peer en P2PSP.

6.3. Instalación y configuración del servidor STUN

En el Capítulo 2 en la Sección 2.4.5 ya se apuntaba que en el proceso de señalización de WebRTC era necesario usar un servidor TURN/STUN para conseguir las direcciones IP y puerto de los peers que están ubicados tras un NAT o un firewall. En algunos de los ejemplos anteriores se ha usado como servidor STUN uno de los servidores gratuitos y accesibles públicamente de Google `stun.l.google.com:19302`. Sin embargo, lo ideal es contar con un servidor STUN propio que permita conocer las direcciones IP de los peer sin necesidad de depender de servicios de terceros.

A continuación se explicará de forma sencilla el proceso de instalación y configuración de un servidor TURN/STUN gratuito. El cuál, además, será software libre.

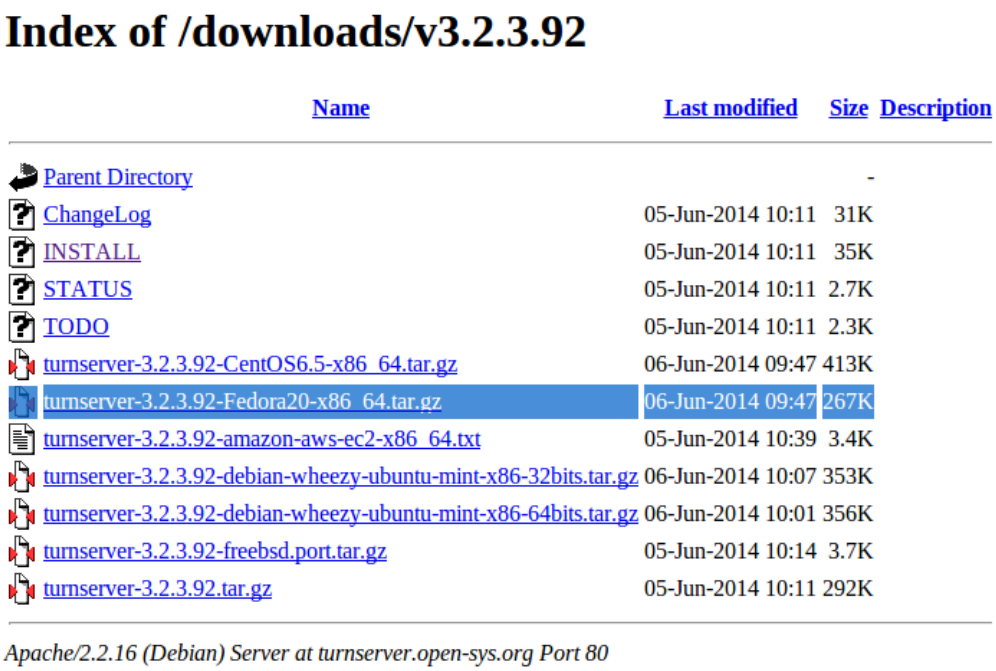
6.3.1. rfc5766-turn-server

Tras estudiar los diferentes servidores de software libre disponibles en Internet para este fin, se ha decidido usar la implementación denominada *rfc5766-turn-server* [20]. El proyecto implementa un servidor TURN/STUN junto con otras librerías para pruebas. Los elementos principales de la implementación son:

- `turnserver`: Implementación del servidor TURN/STUN.
- `turnadmin`: Herramienta de administración. Se puede usar para:
  1. Generar claves para cuentas de usuario TURN.
  2. Listar los usuarios disponibles en un fichero de texto o en una base de datos.
  3. Añadir o eliminar usuarios en un fichero de texto o una base de datos.
  4. Actualizar las contraseñas de los usuarios.
  5. Establecer un secreto compartido para `TURN REST API`.
- **turnutils_uclient:** Emula múltiples clientes UDP, TCP, TLS o DTLS.
- **turnutils_peer:** Un simple "echo" server.
- **turnutils_stunclient:** Un ejemplo de cliente STUN.
- **turnutils_rfc5769check:** Utilidad para comprobar el correcto funcionamiento de la implementación del protocolo STUN/TURN.

6.3.2. Instalación

Existen versiones de este proyecto empaquetadas para los diferentes sistemas operativos y distribuciones Linux. En este caso se instalará sobre un host con Fedora.



Index of /downloads/v3.2.3.92

Name	Last modified	Size	Description
Parent Directory		-	
ChangeLog	05-Jun-2014 10:11	31K	
INSTALL	05-Jun-2014 10:11	35K	
STATUS	05-Jun-2014 10:11	2.7K	
TODO	05-Jun-2014 10:11	2.3K	
turnserver-3.2.3.92-CentOS6.5-x86_64.tar.gz	06-Jun-2014 09:47	413K	
turnserver-3.2.3.92-Fedora20-x86_64.tar.gz	06-Jun-2014 09:47	267K	
turnserver-3.2.3.92-amazon-aws-ec2-x86_64.txt	05-Jun-2014 10:39	3.4K	
turnserver-3.2.3.92-debian-wheezy-ubuntu-mint-x86-32bits.tar.gz	06-Jun-2014 10:07	353K	
turnserver-3.2.3.92-debian-wheezy-ubuntu-mint-x86-64bits.tar.gz	06-Jun-2014 10:01	356K	
turnserver-3.2.3.92-freebsd.port.tar.gz	05-Jun-2014 10:14	3.7K	
turnserver-3.2.3.92.tar.gz	05-Jun-2014 10:11	292K	

Apache/2.2.16 (Debian) Server at turnserver.open-sys.org Port 80

Figura 6.4: Página de descargas del proyecto rfc5766-turn-server.

Por tanto, el primer paso es ir a la página de descargas del proyecto y elegir la versión para el sistema operativo en el que correrá el servidor.

Los comandos necesarios para descargar el servidor y descomprimirlo se muestran en Código 6.15.

```
1 $ wget http://turnserver.open-sys.org/downloads/v3.2.3.92/ ...
    turnserver-3.2.3.92-Fedora20-x86_64.tar.gz
2 $ tar -zxvf turnserver-3.2.3.92-Fedora20-x86_64.tar.gz
```

Código 6.15: Descargar y descomprimir servidor TURN/STUN.

Una vez que se ha descargado el servidor es necesario instalarlo. El empaquetado incluye un fichero ejecutable que permite instalarlo de forma transparente para el usuario, simplemente hay que ejecutar `install.sh`.

```
1 $ cd turnserver-3.2.3.92-Fedora20-x86_64
2 ¬/turnserver-3.2.3.92-Fedora20-x86_64$ sudo ./install.sh
```

Código 6.16: Instalar el servidor TURN/STUN.

El proceso de instalación se llevará a cabo de forma automática. Cuando este proceso finalice se mostrará un mensaje indicando que el proceso se ha realizado correctamente.

6.3.3. Configuración

Los ficheros de configuración del servidor TURN/STUN están ubicados en `/etc/turnserver/`. Existen dos ficheros:

- `turnserver.conf:` Contiene la información de configuración relativa al servidor. Por ejemplo: dirección IP, puertos (TCP, UDP, TLS o DTLS), restricción de ancho de banda, etc.

```
...
# Listener interface device (optional, Linux only).
listening-device=eth0

# TURN listener port for UDP and TCP (Default: 3478).
listening-port=3478

# TURN listener port for TLS (Default: 5349).
tls-listening-port=5349

# Listener IP address of relay server.
listening-ip=172.17.19.101
...
```

Código 6.17: Ejemplo de fichero `turnserver.conf`.

- `turnuserdb.conf:` Almacena las credenciales de los usuarios (nombres de usuario y contraseñas). Las contraseñas se pueden almacenar en texto plano o generando una `key` asociada con la herramienta `turnadmin`.

```
...
#username1:key1
john:0xbc807ee29df3c9ffa736523fb2c4e8ee

#username1:password1
joe:password
...
```

Código 6.18: Ejemplo de fichero `turnuserdb.conf`.

Si la configuración ya está lista, el último paso es lanzar el servidor TURN/STUN. Para esta tarea basta con ejecutar el comando que se muestra en Código 6.19.

```
# Run turnserver
$ turnserver

# Run turnserver permanently
$ nohup turnserver &
```

Código 6.19: Cómo ejecutar servidor TURN/STUN.

6.4. Limitaciones impuestas por la tecnología

Existen algunas limitaciones que hacen que esta implementación no tenga las mismas cualidades que la versión de escritorio o incluso que no pueda ser ejecutado en determinados navegadores Web, entre ellas, se pueden destacar:

1. **Independencia del contenido:** En la actualidad no existe un formato de vídeo universal aceptado por todos los navegadores Web comerciales, provocando que elegir un formato determinado implique que el vídeo no pueda ser reproducido por algunos navegadores.

2. **Estado de la API WebRTC:** WebRTC es sólo un borrador que algunos navegadores han comenzado a implementar y no se sabe cuándo se convertirá en estándar ni qué cambios le esperan a la API hasta llegar a estar en un estado de producción.

3. **Media Source Extensions (MSE):** Para conseguir "enviar" un stream de vídeo a la etiqueta vídeo de HTML5 es necesario que las MSE estén disponibles. Al igual que la API WebRTC, las MSE no están soportadas por todos los navegadores. Sin embargo, esta tecnología está más cerca de convertirse en un estándar ya que es una *Candidate Recommendation* del W3C. Por otra parte, en los navegadores que está soportado, no dispone de una gran compatibilidad con variedad de códecs. Esta situación obliga a usar dos códecs específicos: (1) El formato WebM con códec VP8+Vorbis específicamente codificado para streaming de vídeo o (2) el formato MPEG-DASH (MP4 segmentado).

4. **Prefijos en la implementación:** Mientras la API WebRTC siga sin ser un estándar, los navegadores la implementan usando prefijos delante del nombre de los objetos. Esto obliga a los desarrolladores a distinguir entre un navegador u otro para hacer la aplicación compatible con varios navegadores.

5. **Limitación en el número de `PeerConnection`:** Por defecto y aunque la especificación no dice nada con respecto al número máximo de peers, las implementaciones de la API WebRTC en los navegadores Web limitan el número de `PeerConnection` a 256 conexiones por peer [17]. Esta característica hace que el tamaño del team en la implementación del P2PSP en el navegador Web no pueda ser superior a 256 peers.

Capítulo 7

Evaluación y Resultados

En este capítulo se van a realizar una serie de experimentos con el objetivo de evaluar el comportamiento de la implementación del P2PSP en el navegador Web al ejecutarla en diferentes situaciones bajo entornos reales. Así mismo, se comprobarán los resultados obtenidos comparándolos con el comportamiento esperado.

Todos los experimentos se llevarán a cabo en la última versión del navegador Google Chrome (versión 35) ejecutándose bajo el sistema operativo Ubuntu 13.10. El stream retransmitido será el vídeo de unos diez minutos de duración *Big Buck Bunny* de la fundación Blender [6] en formato WebM a una resolución espacial de 640 x 360.

7.1. Comportamiento normal de un team P2PSP

7.1.1. Peer en LAN

Experimento

El primer experimento consistirá en ejecutar la implementación en un entorno bastante favorable. Este entorno estará formado por el splitter y dos peers en el team, y tanto el splitter como los peers que formarán el team serán ejecutados en la misma red local.

Resultado

Tras ejecutar el team tal y como se muestra en la Figura 7.1 se ha comprobado que el funcionamiento ha sido el esperado pudiendo visualizar el vídeo completamente desde el inicio hasta el final en ambos peers. Se ha comprobado también que no se ha perdido ningún chunk durante la retrasmisión.

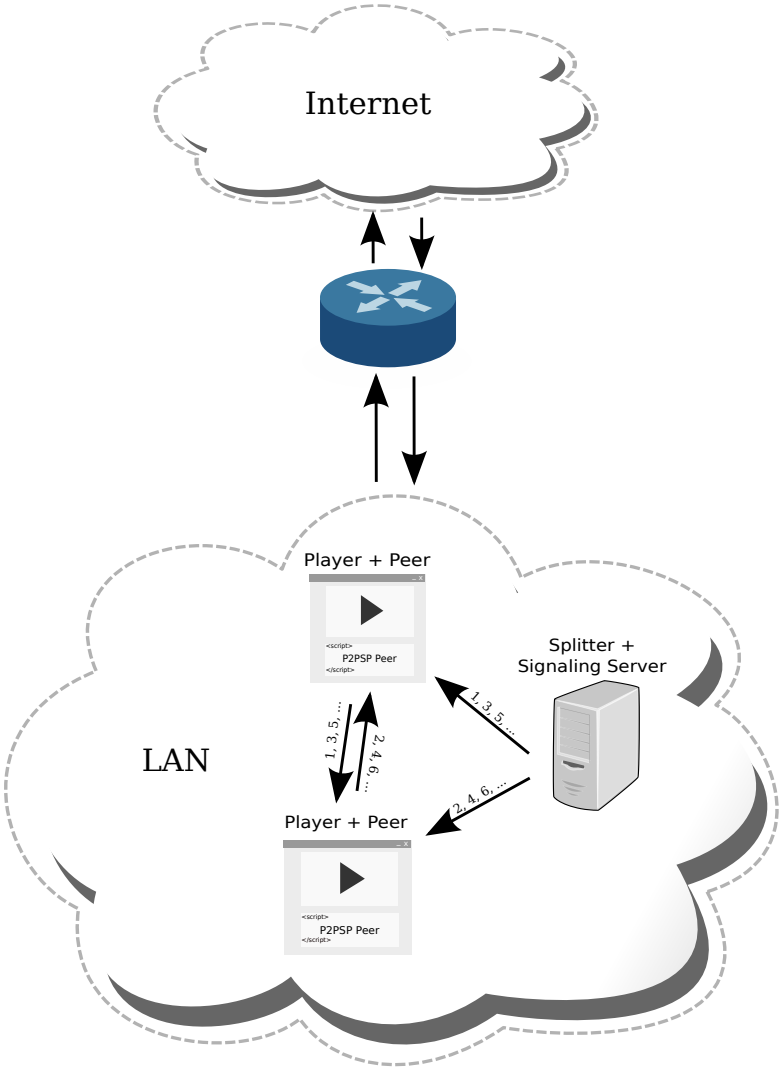


Figura 7.1: Experimento team P2PSP en red LAN. La figura muestra la arquitectura de red de un team donde todos los elementos se ejecutan en una misma red LAN.

7.1.2. Peers en diferentes redes

Experimento

Este experimento será similar al anterior ejecutando un team P2PSP formado por dos peers y el splitter. En esta ocasión el team no está en la red local sino que cada elemento estará en diferentes redes. De forma que el splitter estará en una red y cada peer en otras redes independientes.

Resultado

Una vez se ha realizado el experimento con la configuración de red que se muestra en la Figura 7.2, se ha comprobado que el comportamiento de la implementación ha sido el esperado. Se ha reproducido correctamente y de forma fluida el vídeo completo.

7.2. Pérdida de chunks

7.2.1. Experimento

En los experimentos anteriores todo ha ido bien, pero como se comentaba en el Capítulo 2.4 (Sección 2.4.2), debido al uso del protocolo SCTP para la comunicación entre peers configurado en modo no confiable, es posible que ocurran dos cosas: (1) Los chunks lleguen desordenados, lo cuál no es un problema ya que el protocolo P2PSP tiene esta característica en cuenta y ordena los chunks antes de reproducirlos y (2) se pierda algún chunk. En la versión de escritorio la pérdida de un chunk no supone un problema, ya que los reproductores normalmente reconstruyen el bloque de vídeo que les falta para continuar con la reproducción haciendo que sea prácticamente imperceptible para el usuario.

En este experimento se forzará al splitter a dejar de enviar un chunk de modo que no llegará al peer y por tanto nunca alcanzará el team. El objetivo del experimento es comprobar que ocurre en el navegador Web cuando tiene que reproducir un chunk que no está disponible.

7.2.2. Resultado

Tras hacer las modificaciones pertinentes en el splitter, esto es, forzar que no envíe un chunk a uno de los peers del team. Se ha ejecutado un team de tamaño dos. La reproducción ha ido bien hasta que se ha enviado al reproductor el chunk siguiente al que faltaba, es decir, se ha enviado al reproductor el chunk número 245 e inmediatamente después el chunk número 247. En ese momento, se ha lanzado la excepción mostrada en Código 7.1 desde el elemento `MediaSource`. Esta excepción

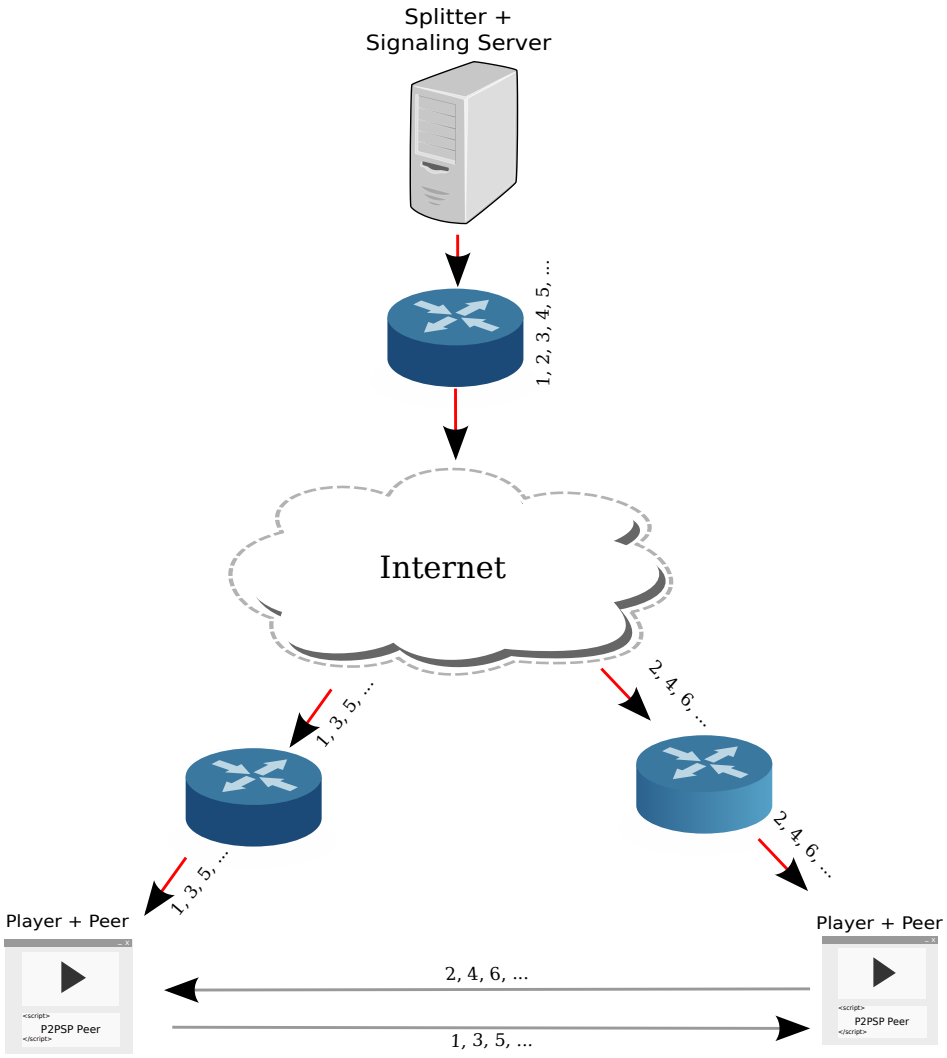


Figura 7.2: Experimento team P2PSP en distintas redes. La figura muestra la arquitectura de red de un team donde cada elementos está en una red distinta.

lleva a pensar que el elemento `MediaSource` necesita (al menos en el estado actual de su implementación como `Draft`) todos los trozos de vídeo para poder reproducir un stream. Otros reproductores como VLC reconstruyen los bloques faltantes de modo que permite reproducir todo el contenido sin problemas.

```
1   "An attempt was made to use an object that is not, or is no ...
       longer,
2   usable.", name: "InvalidStateError", code: 11
```

Código 7.1: Excepción lanzada por `MediaSource`.

7.3. Team mayor de 256 peers

7.3.1. Experimento

Ni la especificación de WebRTC, ni la especificación del P2PSP limitan el tamaño máximo de los teams. Sin embargo, la implementación de la API WebRTC en Google Chrome no permite más de 256 `PeerConnection`'s. En este experimento se intentará crear un team P2PSP en la implementación del navegador con el objetivo de comprobar que ocurre al llegar a esta limitación.

7.3.2. Resultado

Al crear más de 256 `PeerConnection`'s únicamente las primeras 256 disparan el manejador `onicecandidates`, por tanto, el resto nunca llegarán a compartir los candidatos ICE necesarios para el proceso de señalización. Se comprueba entonces que es una limitación impuesta por la implementación de WebRTC en el navegador, posiblemente para evitar colapsar el navegador Web con demasiadas conexiones.

7.4. Peers en diferentes dispositivos

7.4.1. Experimento

Una de las capacidades de esta implementación es la de tener un diseño responsive. Lo que significa que debe ser capaz de adaptarse a cualquier dispositivo ofreciendo una experiencia de usuario de calidad. Para este experimento se ha ejecutado la aplicación en diferentes plataformas con tamaños de pantalla distintos y se ha observado la integración de la interfaz de usuario en las mismas.

Los resultados de esta integración se muestran en el siguiente punto.

7.4.2. Resultado

El experimento se ha llevado a cabo en los siguientes dispositivos:

- Equipo de escritorio a pantalla completa (Chrome en Ubuntu).
- Equipo de escritorio a media pantalla (Chrome en Ubuntu).
- Teléfono móvil con pantalla de 4.7 pulgadas (Chrome en Android).
- Tablet con pantalla de 7 pulgadas (Chrome en Android).

Después de analizar las imágenes se puede afirmar que el diseño se adapta correctamente a cada uno de los entornos en los que ha sido probada la aplicación.

Figura 7.3: Captura de la interfaz a pantalla completa.

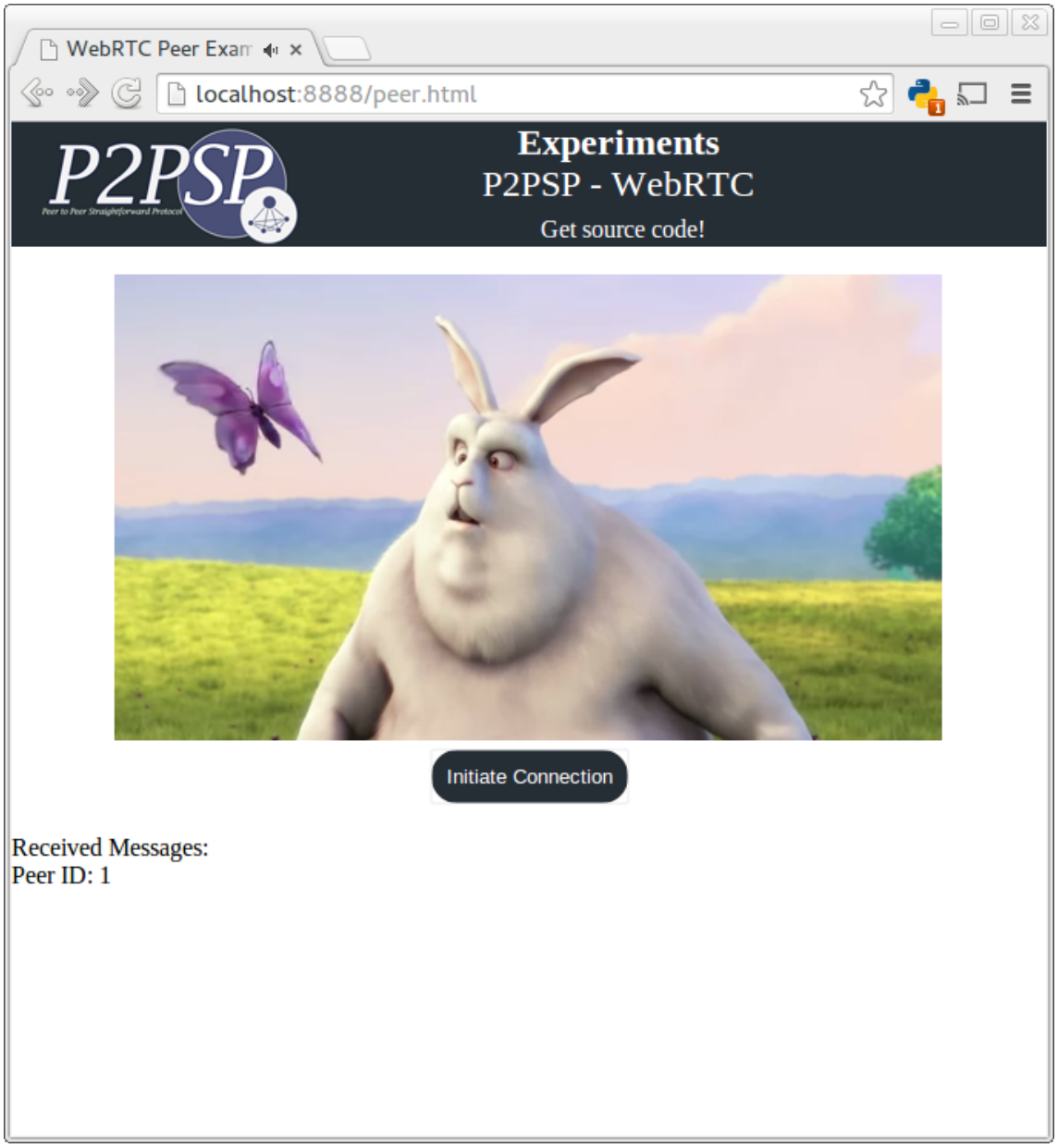


Figura 7.4: Captura de la interfaz a media pantalla.

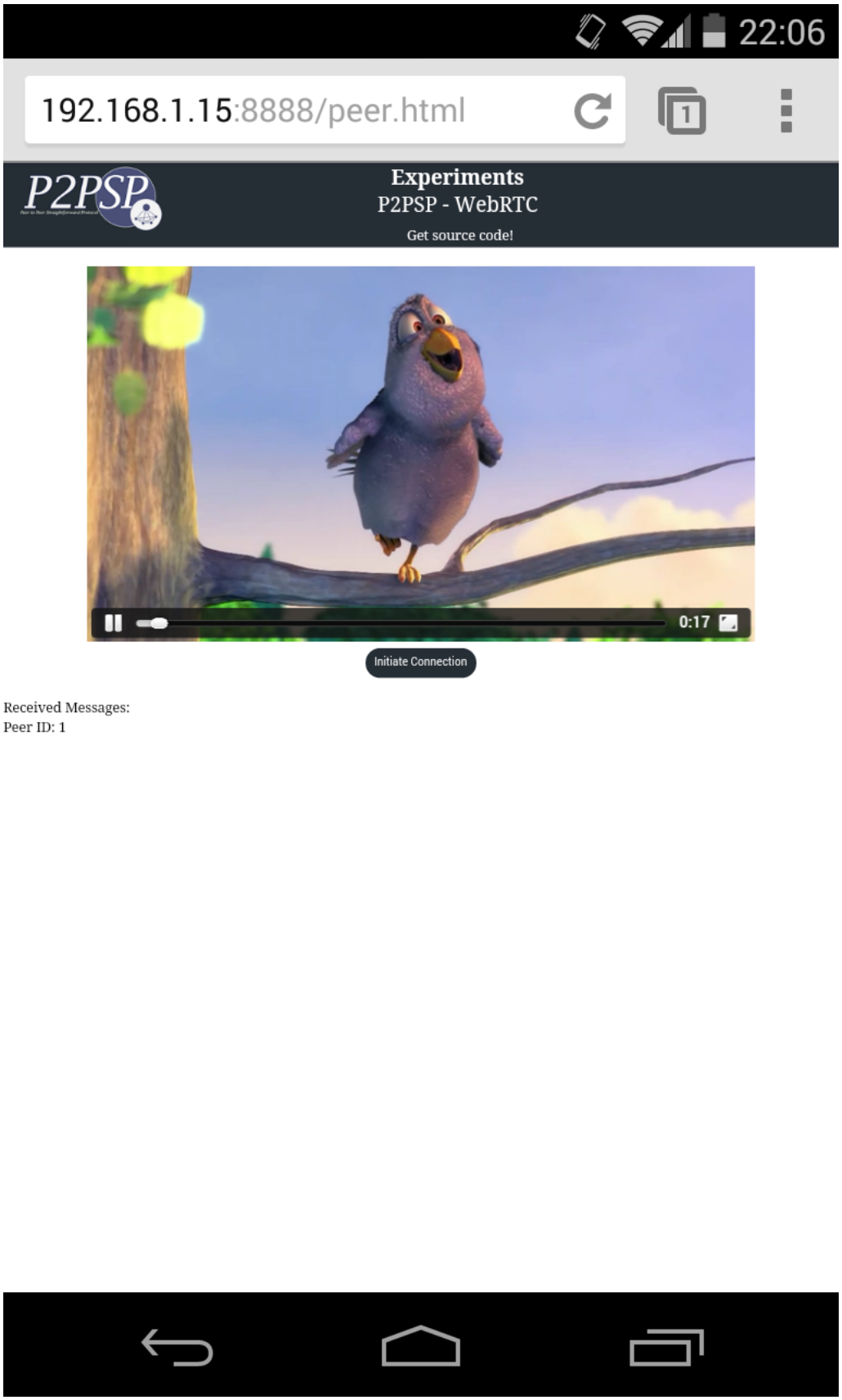


Figura 7.5: Captura de la interfaz en smartphone Nexus 4.

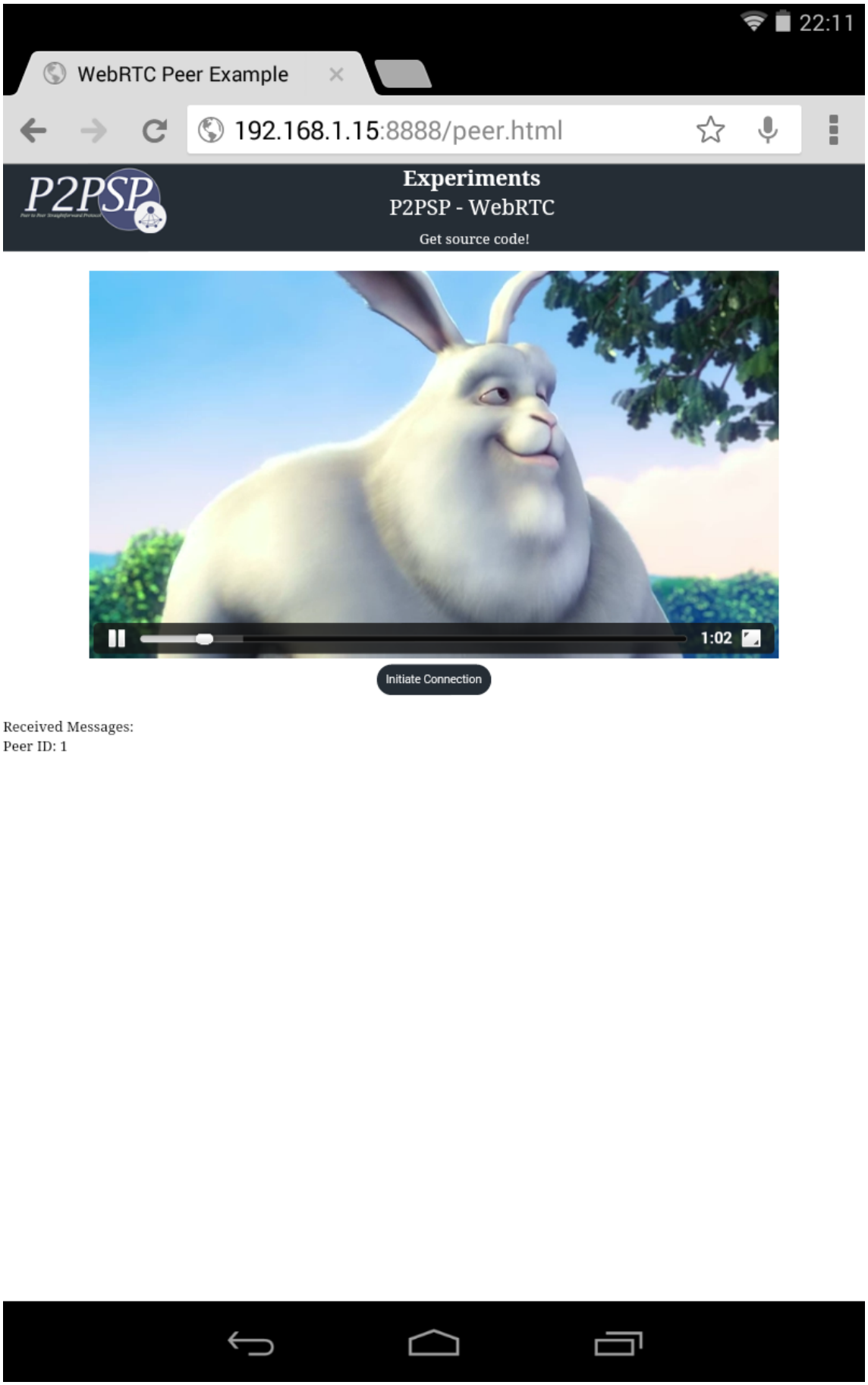


Figura 7.6: Captura de la interfaz en tablet Nexus 7.

Capítulo 8

Conclusiones

En este proyecto se ha realizado un análisis de las tecnologías Web actuales con el objetivo de estudiar si una implementación del protocolo P2PSP podría llevarse a cabo en el navegador Web del usuario y ser ejecutada sin necesidad de plugins o software adicional. Manteniendo en mente la idea de que el usuario sólo tendría que visitar un página Web para comenzar a consumir (y compartir) el contenido.

Tras el periodo de estudio, análisis e implementación, se ha obtenido una aplicación en la que se ha podido comprobar que una implementación de este tipo es posible, alcanzando así los objetivos marcados y estando ante una de las primeras implementaciones de un protocolo de streaming de vídeo, que usando el modelo P2P, es capaz de ejecutarse en el navegador Web del usuario sin necesidad de plugins, y siendo posible esto, en gran parte, gracias a la tecnología WebRTC.

No obstante, y debido a los problemas relacionados con el estado actual de las tecnologías (ver Sección 6.4), es necesario un avance en el desarrollo de todas las tecnologías que se han tratado a lo largo de este trabajo, así como que se conviertan en estándar, para conseguir llegar a una implementación completa del protocolo P2PSP en la Web.

En la Tabla 8.1 se muestran las principales características necesarias en los navegadores Web que hacen que el P2PSP sea o no compatible con cada uno de ellos. Cabe destacar que esta nueva implementación es compatible con Chrome (incluido la versión Canary) y con pequeñas variaciones en el código llegaría a ser compatible con Firefox Nightly. Sin embargo, por el momento, no es posible mezclar peers en el mismo team ejecutándose en navegadores Web de diferentes proveedores.

Navegador	MSE	WebRTC	P2PSP
Chrome	SI	SI	SI
Chrome Canary	SI	SI	SI
Firefox	NO	SI	NO
Firefox Nightly	SI	SI	SI
Internet Explorer	SI	NO	NO
Opera	NO	SI	NO
Safari	NO	NO	NO

Tabla 8.1: Compatibilidad de la implementación del P2PSP con los navegadores Web actuales.

Capítulo 9

Trabajo futuro

Una vez comprobado que es posible llevar a cabo la implementación del módulo *Data Broadcasting Set of rules* en el navegador Web, una de las próximas líneas de trabajo consistiría en seguir implementando el resto de los módulos que forman la totalidad del protocolo P2PSP, aunque para ello es necesario esperar la evolución de la API WebRTC y sobre todo del componente Media Source Extensions de HTML5.

Otra de las prioridades futuras en este proyecto consistiría en mantener actualizada la implementación para que sea compatible con todos los navegadores Web —actualmente sólo funciona en Google Chrome— de acuerdo con el progreso de integración de WebRTC en los mismos y los cambios en la API actual. Además, la evolución natural es convertir esta implementación en una API fácil de usar, de modo que cualquier desarrollador que lo desee pueda incrustar, fácilmente y de forma totalmente transparente, un reproductor (peer P2PSP incluido) en sus propias aplicaciones Web añadiendo simplemente unas pocas líneas de código.

Por otra parte, se está barajando la posibilidad de portar el sistema desarrollado a Google Chromecast (un dispositivo con conexión a Internet que se conecta al televisor y que posee toda la funcionalidad de Google Chrome). Por tanto, se podría ejecutar un peer en un Google Chromecast y reproducir el stream directamente en el televisor, lo que facilitaría sin duda la difusión de este trabajo.

Anexos

Anexo A

Manual de uso de la aplicación

A.1. Splitter

El splitter es la parte de la aplicación que se ejecuta en el lado del servidor y está escrita en Python. Para ejecutarla es necesario tener python instalado en el host y lanzar el siguiente comando en la consola:

```
1 $ python splitter.py
```

Código A.1: Cómo ejecutar splitter en el servidor.

Aparecerá un mensaje indicando que el fichero de vídeo se ha abierto y por tanto, el splitter estará preparado para recibir peers.

A.2. Peer

El peer es la parte cliente de la aplicación que se ejecuta en el navegador Web del usuario. Para que los usuarios puedan ejecutarla en sus navegadores es necesario ponerla a su disposición en un servidor Web en Internet, por ejemplo, Apache. Normalmente bastará con instalar el servidor y mover los ficheros del proyecto al directorio `www` o `public`. Básicamente, al directorio donde esté públicamente accesible a los usuarios.

No hay que olvidar modificar el fichero `peer.js` con la información necesaria para conectarse con el splitter y con el servidor TURN/STUN, tal y como se muestra en Código A.2.

```
1 // Your splitter.py IP:PORT here
2 var signalingChannel = new WebSocket("ws://127.0.0.1:9876/");
3 // Your TURN/STUN server IP:PORT here
4 var configuration = {iceServers: [{ url: ...
      'stun:stun.l.google.com:19302' }]};
```

Código A.2: Configureción de un peer.

Una vez ya está publicada la Web, los usuarios sólo tienen que acceder a la dirección `http://URL/peer.html` desde un navegador compatible.

Anexo B

Experimentos con WebRTC

B.1. Chat multiusuario

Para entender un poco mejor como se va a conseguir transmitir vídeo en el navegador mediante P2P sin necesidad de plugins gracias a WebRTC, primero es necesario entender cómo funcionan los elementos básicos de la API que nos permitirán hacer esto, hablamos de RTCPeerConnection y RTCDataChannel. Para ello, se va a construir un pequeño ejemplo de un chat multiusuario con el objetivo de entender como trabajar con la API WebRTC.

El chat estará compuesto por los siguientes elementos:

- Peer: Es la parte del cliente, cada usuario ejecutará un peer en su navegador. El peer se escribirá en HTML5+JavaScript+WebRTC
- Server (SignalingServer): Es la parte servidora y su misión es simple: hacer que los peer se conozcan entre sí. Los mensajes nunca pasarán por el servidor, estos serán intercambiados directamente entre los peers. El Server se escribirá en Python.

Una explicación más extensa junto al código fuente está disponible en [13]

B.1.1. Server (Signaling Server)

Para que los clientes (peer) se conozca entre sí es necesario hacer uso de un servidor de señalización que facilite la información necesaria de cada uno de los peers para poder establecer una comunicación directa entre ellos. La forma más sencilla y directa es usar WebSocket, para ello se usará la librería `SimpleWebSocketServer` by *Opiate*, que es software libre y facilita su implementación en Python.

Partiendo del módulo `SimpleWebSocketServer` sólo hay que reescribir los métodos `handleMessage`, `handleConnected` y `handleClose` para que intercambien la información de cada peer con el resto de peers.

```
#hanleMessage se ejecuta cuando un mensaje es recibido
def handleMessage(self):
      #recibimos un mensaje (json)
    datos=str(self.data)

        #decodificamos el mensaje (json)
        try:
        decoded = json.loads(datos)
    except (ValueError, KeyError, TypeError):
        print "JSON format error"

        #Reenviamos el mensaje a resto de clientes
    for client in self.server.connections.itervalues():
            if client != self:
              try:
                client.sendMessage(str(self.data))
              except Exception as n:
                  print n

#handleConnected se ejecuta cuando un nuevo cliente se conecta
def handleConnected(self):
    global nextid

    try:
          #enviamos al cliente su id de peer
      self.sendMessage(str('{"numpeer":"'+str(nextid)+'"}'))
          #enviamos al cliente la lista de peer actual
      self.sendMessage(str('{"peerlist":"'+str(peerlist)+'"}'))
          #agregamos el nuevo peer a la lista
      peerlist.append(nextid)
      peeridlist[self]=nextid
      nextid=nextid+1
    except Exception as n:
      print n

#handleClose se ejecuta cuando un cliente se desconecta
def handleClose(self):
      #eliminamos el peer de la lista
    peerlist.remove(peeridlist[self]);
```

Código B.1: Signaling Server para Chat multiusuario.

B.1.2. Peer

Como ya se ha dicho, la parte del cliente se ejecuta en el navegador y toda la funcionalidad se escribe en JavaScript. A continuación sólo se van a comentar algunas partes del código que son interesantes.

Debido a que la parte del proceso de señalización ha sido ampliamente descrita a lo largo de este trabajo, se mostrará únicamente el código que interviene en el intercambio de mensajes vía `DataChannel`.

Se puede acceder al código fuente completo de los experimentos en [5].

```
1  //Enviar un mensaje por DataChannel (al resto de peer)
2  function sendChatMessage() {
3      document.getElementById("receive").innerHTML+="<br ...
           />"+document.getElementById("login").value+ ": "+msg.value;
4    //Para cada peer de la lista...
5      for (i in peerlist){
6      if (peerlist[i]!=idpeer){
7        console.log("send to "+peerlist[i]);
8              //Se envial el mensaje con el nombre de usuario y el ...
                   texto a enviar.
9        try{
10         channel[peerlist[i]].send(document.getElementById( ...
               "login" ).value+ ": "+msg.value);
11       }catch(e){
12         console.log(i+" said bye!");
13       }
14
15     }
16   }
```

Código B.2: Enviar mensaje por `DataChannel`.

Una versión de este experimento está disponible en [15]. Si al introducir un nickname aparece el mensaje "Conecting..." pero no aparece el cuadro de texto con el botón "send" es posible que no haya ningún otro peer (o esté detrás de un NAT simétrico). En este caso, para comprobar que funciona correctamente, se puede abrir otra vez la misma URL en otra pestaña del navegador o incluso en otro equipo de la red.

B.2. Streaming de vídeo entre navegadores

Este experimento permite enviar un vídeo en streaming entre uno o varios navegadores Web. En este ejemplo con WebRTC la aplicación Web nos permitirá cargar un fichero de vídeo en el navegador, dividirlo en bloques, enviar los bloques a otros navegadores y reproducir el vídeo en todos ellos. ¡Todo en tiempo real!

Para conseguir esto, que se puede probar en [14], es necesario utilizar los elementos `Datachannel` de la API WebRTC y Media Source Extensions (MSE). El primero nos permite compartir datos binarios entre los navegadores a través de peer to peer y el segundo nos permite enviar los trozos de vídeo directamente a la etiqueta `<video>` mediante un modelo de buffering.

Este experimento consta de tres partes principales que son:

1. **Leer fichero desde el navegador:** Para cargar el vídeo al navegador se hace uso de la API de archivos que permitirá cargar en memoria trozos del fichero de vídeo de una longitud determinada e ir enviándolos al resto de navegadores para que puedan reproducirlo en tiempo real.

2. **Alimentar la etiqueta vídeo:** Este proceso es el mismo que sigue la implementación del P2PSP en el navegador Web que se describe en el Capítulo 6, en la subsección 6.2.2.

3. **Enviar bloques al resto de peer:** Aquí se controla lo que se envía y recibe de los `DataChannel`, tal y como se muestra en la subsección 6.2.2 del Capítulo 6.

En el Código B.3 se muestra el código que permite dividir el vídeo en trozos directamente en el navegador Web.

```
1  function readBlob(size) {
2
3      var files = document.getElementById('files').files;
4      if (!files.length) {
5        alert('Please select a file!');
6        return;
7      }
8
9      var file = files[0];
10     var start = size;
11     var stop = file.size - 1;
12     if (size>file.size)
13   return;
14     var reader = new FileReader();
15
16     //Se dispara cuando se ha completado la solicitud.
17     reader.onloadend = function(evt) {
18       //Es necesario comprobar el estado
19       if (evt.target.readyState == FileReader.DONE) { // DONE == 2
20   handleChunk(evt.target.result); //Se envia el bloque de video ...
         al reproductor (etiqueta video)
21   sendChatMessage(evt.target.result); //Se comparte el bloque de ...
         video con el resto de navegadores (via DataChannel)
22       }
23      };
24
25      reader.onload = function(e) {
26     feedIt(); //Continuamos leyendo
27      };
28
29      //Extraemos un trozo de video (1024 bytes)
30      var blob;
31      blob = file.slice(start, start + 1024)
32      reader.readAsArrayBuffer(blob);
33
34   }
35
36   //Solicitar leer los siguientes 1024 bytes.
37   function feedIt(){
38   readBlob(chunksize);
39   chunksize+=1024;
40   }
```

Código B.3: Dividir vídeo en el navegador Web.

Bibliografía

[1] AndYet. Browser support scorecard. http://iswebrtcreadyyet.com/.

[2] BitLet.org. Torrent video streaming. http://www.bitlet.org/video.

[3] Creative Commons. Licencia reconocimiento – compartirigual (by-sa). http://creativecommons.org/licenses/by-sa/4.0/legalcode.

[4] Opiate en GitHub. Simple websocket server. https://github.com/opiate/SimpleWebSocketServer.

[5] P2PSP Team en Launchpad. Experimentos con webrtc. http://bazaar.launchpad.net/~p2psp/p2psp/webrtc/.

[6] Blender Foundation. Big buck bunny. http://www.bigbuckbunny.org/.

[7] Python Software Foundation. Web frameworks for python. https://wiki.python.org/moin/WebFrameworks.

[8] IETF. Peer to peer streaming protocol (ppsp). http://datatracker.ietf.org/wg/ppsp/charter/.

[9] Azuerus Software Inc. Cliente vuze bittorrent. http://www.vuze.com/.

[10] BitTorrent Inc. utorrent: Pequeño y sin límites. http://www.utorrent.com/intl/es/.

[11] P. Matthews J. Rosenberg, R. Mahy and D. Wing. Session traversal utilities for nat (stun). http://tools.ietf.org/html/rfc5389, October 2008.

[12] R. Mahy J. Rosenberg and P. Matthews. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). http://tools.ietf.org/html/draft-ietf-behave-turn-16, July 2009.

[13] Cristóbal Medina López. Blog. http://www.cristobalmedinalopez.es/.

[14] Cristóbal Medina López. Video streaming system over webrtc. http://www.p2psp.org/webrtc-streaming/.

[15] Cristóbal Medina López. Webrtc multi-party chat example. http://p2psp.org/chat/.

[16] Ethan Marcotte. Responsive web design. http://www.alistapart.com/articles/responsive-web-design/.

[17] Vikas Marwaha. Onicecandidate not called for more than 10 peer connections. https://code.google.com/p/webrtc/issues/.

[18] Cristobal Medina-López, J.A.M. Naranjo, Juan Pablo García-Ortiz, L. G. Casado, and Vicente González-Ruiz. Execution of the P2PSP protocol in parallel environments. In Guillermo Botella y Alberto A. Del Barrio Garcia, editor, *Actas XXIV Jornadas de Paralelismo*, pages 216–221, Madrid, Septiembre 2013.

[19] Cristóbal Medina-López, Juan Pablo García Ortiz, J.A.M. Naranjo, L.G. Casado, and Vicente González-Ruiz. IPTV using P2PSP and HTML5+WebRTC. In *The Fourth W3C Web and TV Workshop*, Munchen, Germany, March 2014.

[20] Oleg Moskalenko. rfc5766-turn-server. https://code.google.com/p/rfc5766-turn-server/.

[21] Swirl project. Bringing peer-to-peer streaming to the world. http://swirl-project.org/.

[22] Transmission Project. Transmission bittorrent. https://www.transmissionbt.com/.

[23] StreamRoot. Live streaming delivery. http://www.streamroot.io/.

[24] P2PSP Team. Implementation of the P2PSP (Peer to Peer Straightforward Protocol) in Lauchpad. https://launchpad.net/p2psp.

[25] P2PSP Team. Peer to peer straightforward protocol. http://p2psp.org/p2psp-protocol.

[26] WebRTC Team. Web real-time communication. http://www.webrtc.org/.

[27] W3C Tim Berners-Lee. First mention of html tags on the www-talk mailing list. http://lists.w3.org/Archives/Public/www-talk/1991SepOct/0003.html.

[28] W3C. Hypertext markup language, versión 5. http://www.w3.org/html/wg/drafts/html/master/.

[29] W3C. Media source extensions. http://www.w3.org/TR/media-source/.

[30] W3C. Webrtc 1.0: Real-time communication between browsers. http://www.w3.org/TR/webrtc/.