UNIVERSIDAD DE LA REPÚBLICA
Facultad de Ingeniería

Instituto de Computación

Informe de Proyecto de Grado

Uso de las tecnologías PPSP (Peer-To-Peer Streaming Protocol) y WebRTC (Web Real-Time Communications) en servicios de *broadcasting* de video en Internet.

Autores:
Braulio MARTÍNEZ
Marcelo CASIRAGHI
Martín RODRÍGUEZ BLANCO
Sebastian SUTTNER

Tutor:
Pablo RODRÍGUEZ BOCCA

Agosto de 2014

Resumen

Las redes Peer-To-Peer (P2P) son foco de discusión hoy en día gracias a sus diversas aplicaciones. Se han implementado variados sistemas que se benefician de las propiedades de las arquitecturas distribuidas y colaborativas.

A su vez, el contenido multimedia goza de una gran demanda por parte de los usuarios en Internet. Grandes compañías que sirven contenido estudian posibles maneras de abaratar costos de distribución aplicando soluciones P2P y sin comprometer la calidad percibida de sus clientes.

Esta problemática, en conjunto con las discusiones actuales sobre la estandarización de tales arquitecturas y el auge de nuevas tecnologías relacionadas, son los principales motivadores de este proyecto.

Se plantea entonces desarrollar un prototipo que ataque la aplicación de estrategias P2P en un sistema de *streaming* multimedia, utilizando WebRTC y otras tecnologías de punta como herramienta de implementación. Para cumplir con las necesidades establecidas, se define como objetivo adicional el lograr un intercambio eficiente de contenido entre los participantes del sistema, y así abaratar los costos de distribución de manera considerable.

Se presenta un amplio estudio del estado del arte de las herramientas utilizadas para la implementación del prototipo y se logra validar su utilidad mediante pruebas en ambientes controlados. Finalmente, se obtienen resultados muy positivos, que demuestran que bajo un contexto Video On Demand (VoD) particular, el intercambio de contenido en forma P2P alcanza un promedio de 58%.

Índice General

Índice de Figuras

Índice de Tablas

1 Introducción

En los últimos años ha crecido el consumo de contenido *multimedia* a través de Internet y existe una gran cantidad de contenido disponible desde diversos proveedores. En particular, ha contribuido en gran medida el deseo por parte de los usuarios de poder disfrutar y compartir distintos tipos de espectáculos, tales como series televisivas, películas, eventos familiares, video-conferencias, entre otros. Entre las modalidades de transmisión de video hoy en día, se espera que el *broadcasting* (difusión de uno a muchos) sea el servicio con mayor crecimiento [35].

E l *streaming multimedia* en Internet ofrece importantes desafíos tecnológicos. Aunque muchas aplicaciones dedicadas a esta rama tecnológica todavía conservan arquitecturas puramente cliente/servidor, otras han optado por arquitecturas P2P puras o híbridas.

Para aquellos que se encuentren en un período de transición de su arquitectura o generando nuevas aplicaciones, existe la posibilidad de ofrecer *broadcasting* con tecnologías P2P mediante dos procesos muy diferentes:

- La Internet Engineering Task Force (IETF) se encuentra elaborando un estándar del protocolo Peer-to-Peer Streaming Protocol[1] (PPSP), y varios proveedores de tecnología se encuentra desarrollando productos que lo implementan.
- Los fabricantes de navegadores *web* encabezados por Google, se encuentran desarrollando la tecnología Web Real-Time Communications [20] (WebRTC), que permite implementar una red P2P de contenido *multimedia*.

Esta tesis plantea el estudio del enfoque tecnológico del *streaming multimedia* distribuido en l a *web*. Se realiza un amplio estudio del estado del arte, estudiando diversos tipos de tecnologías presentes en la actualidad, pero enfocándose particularmente en la tecnología WebRTC.

1.1 Objetivo

El objetivo fundamental de este trabajo consiste en desarrollar un prototipo funcional utilizando WebRTC con el fin de estudiar su viabilidad desde un punto de vista práctico y teórico, intentando incluir herramientas nuevas de HTML5 como *web sockets* y *data channels*.

El prototipo consiste en una arquitectura que permite a un usuario acceder a un sitio *web* y automáticamente participar de una red P2P para descargar y reproducir desde el mismo navegador un video que se encuentra almacenado en un servidor de contenido y en los navegadores de otros usuarios que se encuentren visitando el sitio en el mismo momento.

Las particularidades fundamentales de la solución consisten en que la conexión entre los navegadores se realiza en forma directa, sin la necesidad de instalación de librerías externas ni *plugins* sobre el navegador. Toda la tecnología utilizada se encuentra implementada de forma nativa en el navegador, lo cual libera a un usuario del tedioso procedimiento de instalar y mantener versiones de programas externos.

[1] Y. Zhang. *Problem Statement and Requirements of Peer-to-Peer Streaming Protocol (PPSP)*. RFC 6972. http://tools.ietf.org/rfc/rfc6972.txt. 14 de Julio de 2013.

1.2 Motivación

El principal motivo que impulsó la realización de este trabajo se refleja en que el flujo *multimedia* en la *web* se encuentra en su mayor momento. Para el año 2018, CISCO prevé que prácticamente el 84% de los datos accedidos globalmente se compongan de contenido de video [35].

A su vez, son muy pocas las aplicaciones que pretenden encarar un enfoque distribuido para el compartimiento de datos *multimedia*, considerando además que se trata de proyectos privativos. Algunos de los ejemplos que atacan esta problemática son estudiados en el *Capítulo 2 Estado Del Arte*.

Las tecnologías WebRTC y *web sockets* son muy nuevas y aún se encuentran en proceso de borrador, por lo que su estudio y utilización en el prototipo de solución resulta un gran aporte a la academia y a la comunidad.

1.3 Resultados

La solución implementada representa lo más posible un escenario real de una red distribuida. Para lograr este objetivo se tomó como base el protocolo BitTorrent que lideró el área de la arquitectura P2P, y la plataforma GoalBit[2], que surge como una extensión del primer protocolo para distribución de flujo de video en tiempo real a través de Internet.

El prototipo cumple con varios requerimientos técnicos tales como contar con una arquitectura similar a un sistema BitTorrent, e implementar algoritmos del protocolo que permitan un intercambio eficiente de partes del video. Además, lo logra con un porcentaje considerable de intercambio del video en forma P2P que alcanza un promedio de 58% en un contexto Video On Demand (VoD).

1.4 Estructura del documento

El presente documento se encuentra organizado en capítulos, y dentro de estos en secciones y sub-secciones.

Capítulo 2 Estado Del Arte. Este capítulo contiene una revisión del estado del arte del flujo *multimedia* en la *web*. En primer lugar, se introducen algunos conceptos necesarios para el entendimiento del trabajo. Luego, se realiza un pequeño estudio de los tipos de redes P2P existentes y se presentan varios proyectos que implementan arquitecturas P2P exhibiendo sus ventajas y desventajas. Finalmente, se estudian las tecnologías provistas por HTML5 que se utilizaron para el desarrollo del prototipo tales como WebRTC y *web sockets*.

Capítulo 3 Parte Central del Trabajo. El capítulo refiere al aporte de la tesis a la academia y a la comunidad. Se incluye un listado de los requerimientos que debe cumplir la solución desarrollada y las decisiones tomadas a partir de los mismos.

Capítulo 4 Implementación. Se realiza una descripción de la solución implementada, presentando su diseño y arquitectura, y los criterios que fueron tomados para definirlos. Además, se presenta un listado de restricciones que surgieron a raíz del uso de ciertas tecnologías. También se presentan los algoritmos desarrollados en la implementación y algunas librerías de gran utilidad que facilitaron la codificación.

[2] *GoalBit Solutions*. http://www.goalbit-solutions.com/. Abril de 2013.

Capítulo 5 Pruebas realizadas. Se incluyen los escenarios y casos de prueba que se utilizaron para evaluar la solución. Se detallan los resultados obtenidos y un estudio de los mismos.

Capítulo 6 Conclusiones. En este capítulo se evalúan los resultados obtenidos y las dificultades encontradas. Se plantea una autocrítica de lo realizado y de elementos faltantes de la solución, como también un resumen de trabajo a futuro.

Bibliografía. Se incluye un listado de las referencias bibliográficas utilizadas para la realización del presente trabajo.

Anexos. Se incluye información adjunta al documento.

2 Estado Del Arte

Este capítulo plantea el estado del arte del área de *streaming multimedia* distribuido en la actualidad, e introduce conceptos generales necesarios para la comprensión del presente trabajo.

2.1 Conceptos Generales

2.1.1 Video Digital

Un video consiste en una sucesión de imágenes presentadas a una cierta frecuencia. El ojo humano es capaz de distinguir hasta 20 imágenes por segundo en forma aproximada. En consecuencia, cuando se presentan más de esta cantidad de imágenes por segundo, es posible engañar al ojo y crear la ilusión de una imagen en movimiento. La fluidez de un video se caracteriza por el número de imágenes por segundo y se expresa en frecuencia de cuadros por segundo (Frames Per Second, FPC) [1].

El video digital consiste en una representación digital de una señal de video y no en una señal analógica. Esto implica la captura, manipulación, distribución y almacenamiento de la señal de video en formato digital para facilitar su representación en un dispositivo móvil o PC. Dado que las imágenes digitales se muestran en una frecuencia determinada, es posible saber el número de *bytes* transferidos por unidad de tiempo.

Varios tipos de aplicaciones surgieron en torno al concepto de video digital, en particular las que emiten video vía satélite o terrestre, video-conferencias, entre otros. Algunas de las primeras aplicaciones de video digital que corrían en PCs fueron QuickTime Player[3] de Apple, que surgió en 1991 con una baja calidad de video digital, y RealPlayer[4] de RealNetworks principalmente utilizado para grabar audio y video. Este tipo de aplicaciones fue multiplicándose y evolucionando en forma muy rápida debido al veloz crecimiento de Internet.

2.1.1.1 Contenedores de Video

Un contenedor de video consiste en un formato que define cómo almacenar los componentes de un video en un archivo [1]. Un archivo de video usualmente contiene múltiples *tracks*: un *track* de video (sin audio), junto con uno o más *tracks* de audio (sin video). Éstos *tracks* se encuentran interrelacionados: un contenedor de audio contiene marcadores dentro del mismo que lo ayudan a sincronizar el audio con el video.

El concepto de contenedor de video es similar al de un archivo de tipo zip, que define cómo se almacenan las cosas que contiene, pero no qué tipo de datos almacena (la realidad es un poco más compleja, dado que no todos los flujos de video son compatibles con todos los formatos de contenedores). En el *Anexo 1 Formatos de contenedor y codecs* se encuentran los principales formatos de contenedores, por ejemplo, AVI, MPEG, entre otros.

2.1.1.2 *Codecs* de Video

La codificación de video es una técnica que consiste en eliminar similitudes y/o redundancias existentes en una señal de video. Una señal digital de video consiste en una secuencia de imágenes digitales consecutivas. Estas secuencias tienen redundancia temporal, ya que las mismas presentan ciertos movimientos sobre algunos objetos.

[3] *Apple QuickTime Player.* https://www.apple.com/quicktime/. Mayo de 2013
[4] *Real Player*. http://mx.real.com/realplayerformac.html. Mayo de 2013

Además, dentro de las imágenes existe también redundancia espacial, dado que, en general, existe una correlación entre *pixels* vecinos. Finalmente existe una redundancia estadística o de entropía.

Los codificadores de video tienen como cometido eliminar redundancias, de forma de poder comprimir el video considerablemente. Esto conlleva una pérdida irrecuperable de información, ya que codifican solamente las partes relevantes a la percepción del video.

Los métodos de compresión son definidos mediante su codificador (*coder*) y decodificador (*decoder*), a los cuales se les conoce como *codec* (*enCOder* / *decoDEC*).

Cuando un usuario “mira” un video, el reproductor de video realiza varias actividades al mismo tiempo:

1. Interpreta el formato del contenedor para descifrar qué *tracks* de video y de audio se encuentran disponibles, y cómo se encuentran las mismas almacenadas dentro del archivo, de forma tal de poder encontrar los datos que debe decodificar posteriormente.
2. Decodifica el flujo de video y muestra una serie de imágenes (*frames*) en pantalla.
3. Decodifica el flujo de audio y envía el sonido a los parlantes por el (los) canal(es) correspondientes.

Un *codec* de video se define entonces como un algoritmo por el cual el flujo de video es codificado. Muchos *codecs* de video modernos utilizan distintos tipos de trucos para minimizar la cantidad de información requerida para desplegar una imagen en pantalla luego de la anterior. Por ejemplo, en lugar de guardar cada imagen individual, se almacenan únicamente los cambios entre las imágenes.

Actualmente existen varios *codecs* de video en uso. En el *Anexo 1 Formatos de contenedor y codecs* se describen los más conocidos, por ejemplo, WebM, H.264, entre otros.

2.2 Redes P2P

En esta sección se ven las aplicaciones tradicionales que implementan arquitecturas P2P. Primero para compartir archivos, y luego para *streaming*.

2.2.1 Redes P2P para compartir archivos

2.2.1.1 BitTorrent

En esta sección se presenta el protocolo BitTorrent [2] para la compartición de datos P2P. BitTorrent se diseñó para facilitar la transferencia de archivos entre múltiples *peers* a través de redes no confiables.

Resulta conveniente explicar el funcionamiento de este protocolo, dado que muchas de las redes P2P orientadas a la distribución de video se basan en el mismo, incluido el protocolo de transporte código abierto GoalBit (GBTP) [3].

Dado que BitTorrent es un protocolo de gran complejidad, se decidió enfocarse únicamente en sus componentes y mecanismos fundamentales. En un sistema P2P típico como BitTorrent, los usuarios tienen información de interés para otros usuarios, tal como música, videos, *software*, fotos, entre otros.

Las entidades que comparten (descargan y suben) contenido en BitTorrent se conocen como *peers*. Un *peer* se define como cualquier tipo de cliente BitTorrent que se encuentra participando en una descarga o subida de contenido. Se plantean los siguientes problemas al momento de compartir contenido:

1. ¿Cómo hace un *peer* para encontrar otros *peers* que tengan el contenido deseado?
2. ¿Cómo replican el contenido los *peers* para proveer descargas de mayor velocidad?
3. ¿Cómo hacen los *peers* para fomentar la subida de contenido por parte de otros *peers*?

Para solucionar el primer problema, BitTorrent toma el enfoque de proveer de un archivo de meta-información denominado *torrent*. Este archivo tiene un tamaño pequeño y se utiliza para describir y verificar la integridad de los datos a ser descargados de otros *peers*.

El *torrent* tiene un formato de codificación específico conocido como *Bencoding* y posee dos tipos de información fundamentales. Una consiste en el nombre del *tracker* [2], que es un servidor que gestiona listas de *peers* que comparten un contenido en común. El otro tipo de información que maneja el *torrent* es una lista de pedazos de igual tamaño del contenido, conocidos en BitTorrent como *chunks*. El tamaño de los *chunks* varía generalmente entre 64KB y 512KB. El *torrent* contiene el nombre de cada *chunk*, dado por un *SHA-1 hash* de 160-bit del *chunk* que lo identifica en la red.

Para descargar el contenido descrito en el *torrent*, un *peer* contacta en primer lugar al *tracker* realizando pedidos HTTP o HTTPS. El *tracker* le responde con una lista de *peers* que se encuentran activamente descargando y subiendo el contenido buscado, los cuales conforman lo que se denomina un *swarm*.

Los *peers* miembros de un *swarm* contactan al *tracker* en forma regular para reportarle que aún se encuentran activos o para reportarle que desean abandonar el *swarm*. Cuando un nuevo *peer* contacta al *tracker* para unirse a un *swarm*, el *tracker* le informa acerca de los otros *peers* en el *swarm*.

El segundo problema planteado consiste en cómo compartir el contenido de forma tal que se proporcionen descargas de alta velocidad. Cuando un *swarm* se forma por primera vez, algunos *peers* deben tener todos los *chunks* que conforman el contenido a ser compartido. Estos *peers* se denominan *seeders*.

A medida que un *peer* participa en un *swarm*, se encuentra descargando *chunks* faltantes de otros *peers* en forma simultánea, y sube *chunks* que ya ha descargado para otros *peers* que aún lo precisen. Cuando un *peer* recoge todos los *chunks* del contenido, tiene la opción de abandonar el *swarm* y retornar cuando desee, o permanecer en el mismo. Los mensajes que se envían los *peers* entre ellos son transmitidos sobre el protocolo de transporte TCP.

Para que la metodología descrita funcione, cada *chunk* debe estar disponible para varios *peers*. Si todos los *peers* persiguieran descargar los *chunks* en el mismo orden, muchos dependerían de los *seeders* para obtener el próximo *chunk*, lo cual generaría un cuello de botella. En lugar de esto, los *peers* de BitTorrent intercambian listas de *chunks* que tienen entre ellos. Luego, cada *peer* selecciona los *chunks* más raros que son difíciles de encontrar para descargar (estrategia *rarest-first*). La idea consiste en descargar el *chunk* más raro, de

forma tal de hacer una copia del mismo, lo cual facilita la descarga de dicho *chunk* para otros *peers* a futuro.

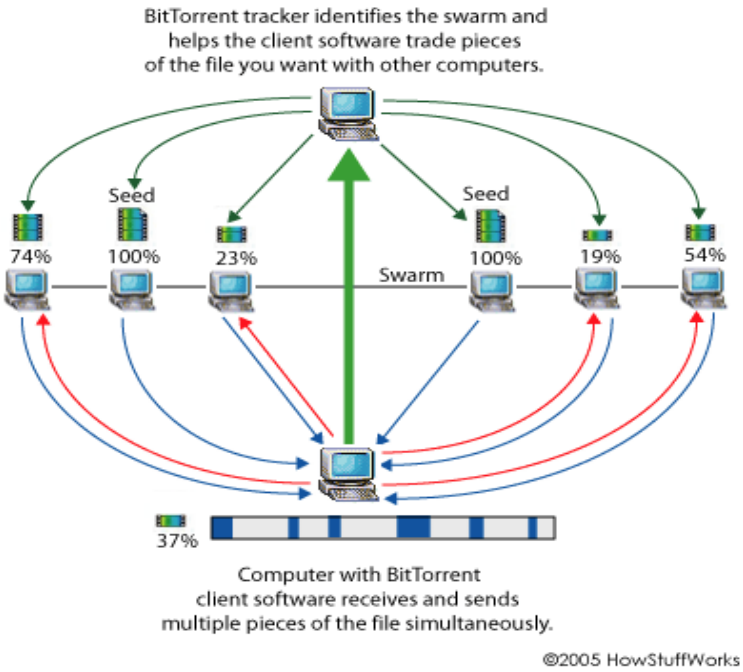

Figura 1. Arquitectura Básica de un sistema BitTorrent [2]

El problema de fomentar la subida de contenido por parte de otros *peers* consiste en uno de los más difíciles. Los nodos que interactúan con el sistema descargando y compartiendo contenido simultáneamente se denominan *leechers.* Sin embargo, pueden existir *peers* maliciosos que sólo descarguen y no compartan nada. En caso de haber muchos de estos, el sistema no funcionará en forma óptima.

La aproximación que toma BitTorrent es la de recompensar a los *peers* que demuestran subir mucho contenido. Primeramente, cuando un *peer* recibe un *swarm,* elige un subconjunto *peers* con los cuales comunicarse, descargando *chunks* de los mismos mientras también les comparte en forma paralela. El *peer* continúa intercambiando *chunks* con solo un pequeño número de *peers* que proveen la mayor *performance* de descarga, mientras también intenta conseguir otros buenos compañeros en forma aleatoria. Esto se conoce como algoritmo *optimistic unchoking*.

Con el transcurso del tiempo, el anterior algoritmo persigue juntar *peers* con tasas de descarga y subida similares. Mientras más contribuya un *peer* con los otros, más puede esperar en retorno. Si un *peer* no sube *chunks* de contenido a otros, o los sube con baja velocidad, dicho *peer* será desactivado o *choked*. Esta estrategia persigue desalentar comportamientos antisociales entre *peers* de un *swarm*. En BitTorrent se implementa esta estrategia por medio de la políticas de selección de *peers, tit-for-tat*.

Si se piensa en adaptar el protocolo BitTorrent para compartir un video o audio en vivo, se puede ver que surgen varias dificultades. En primer lugar, y como se explicó anteriormente, en BitTorrent los *peers* comparten *chunks* pero no en forma ordenada, lo cual no es aplicable para un video en vivo que requiere un flujo continuo de datos. A su vez, la

estrategia *rarest-first* de BitTorrent tampoco es considerada adecuada para compartir video en vivo en algunos escenarios, dado que genera tiempos grandes de espera en el comienzo de la reproducción del video.

2.2.2 Redes P2P para *streaming*

2.2.2.1 Redes de distribución *multimedia*

Las redes de distribución de video y audio en vivo deben satisfacer restricciones mucho más fuertes que las de un protocolo de distribución de archivos como BitTorrent, dado que por ejemplo, se cuentan con restricciones de tiempo real en la reproducción. En esta sección se comentan varios ejemplos de protocolos de *streaming multimedia* P2P que consideran estas restricciones y permiten una reproducción en tiempo real exitosa.

Cabe destacar que, dependiendo de la topología que pueda estar asociada con las conexiones entre *peers*, es posible distinguir entre los siguientes tres tipos de aplicaciones d e *streaming* P2P: basada en árboles (*Tree-based*), basada en mallas (*Mesh-based*) e Híbrida (*Hybrid*). A continuación se explican las mismas ejemplificando para cada caso, según lo definido en los borradores de PPSP de la IETF [3].

2.2.2.1.1 Distribución basada en árboles (*Tree-based*)

En este tipo de red P2P, los *peers* se organizan formando una red con un *overlay* en forma arborescente con nodo raíz siendo el origen del *streaming*. Un nodo que dirige datos hacia las hojas del árbol se denomina padre, y el nodo que recibe los datos se denomina nodo hijo. Este tipo de distribución se categoriza como *push-based* según [3], dado que la transmisión de contenido *multimedia* se realiza sin haber sido solicitada por el *peer* receptor.

Al nodo origen del árbol se le conoce como *Broadcaster*, quien se encarga de enviar los datos hacia los nodos hijos, y éstos a su vez lo reciben, consumen y reenvían a sus hijos. De esta forma, los *peers* van construyendo en forma automática un *overlay* de distribución. Un ejemplo de distribución arborescente simple se presenta en la Figura 4.

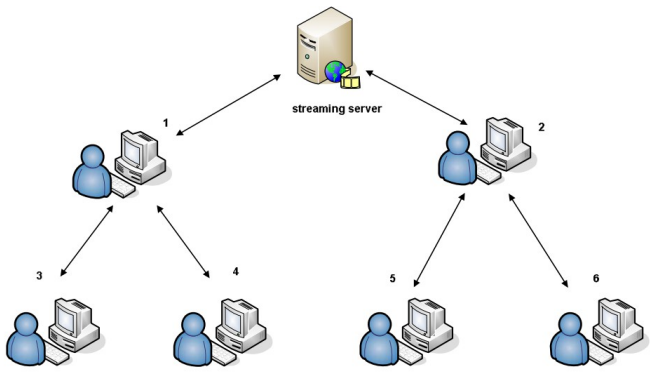

Figura 2. Distribución basada en Árboles [37]

Dada la estructura de este tipo de topología, se presentan dos claros problemas. El primero consiste en las cadenas arborescentes muy largas, las cuales pueden conllevar un largo tiempo de retraso en enviar el contenido *multimedia* desde el *peer* origen hasta las hojas. El segundo problema consiste en la vulnerabilidad ante la caída de un nodo predecesor. Ante estos casos se intenta que el contenido se envíe a más de un nodo predecesor.

La principal ventaja de esta tecnología es que es sencilla de entender y de implementar.

2.2.2.1.2 Distribución basada en mallas (*Mesh-based*)

Este tipo de *overlay* de distribución es de los más implementados hoy en día, y tiene mucha similitud con BitTorrent. No existe una topología definida en la red: los *peers* se organizan conectándose en forma aleatoria, y el contenido *multimedia* se distribuye en forma *pull-based* (el contenido se envía a los *peers* receptores a demanda de los mismos [3]). Por esta razón es que a este tipo de sistemas se les refiere también como *data-driven*.

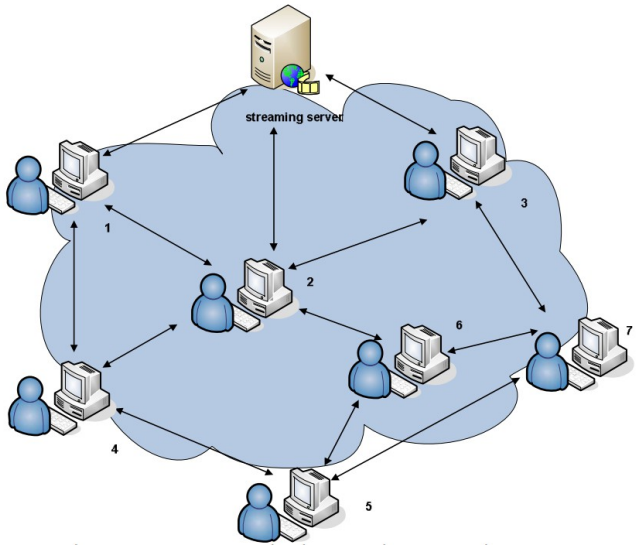

Figura 3. Distribución Basada en Mallas [37]

Dentro de la red se encuentra la información de qué *peer* se encuentra descargando qué contenido para un momento dado. Gracias a esta información, cuando un *peer* se conecta a la red, obtiene una lista de otros *peers* que se encuentran descargando el mismo contenido en el *swarm*. Los archivos se distribuyen en *chunks*, y cada *peer* conoce los *chunks* que poseen el resto de los compañeros en el *swarm*.

Dado que las distribuciones basadas en mallas no mantienen una relación padre-hijo en su estructura durante la transmisión de datos, este tipo de topología presenta desventajas cuando los *chunks* de contenido son descargados por los *peers* de un *swarm*. Por ejemplo, se requiere de *buffers* de mayor tamaño donde almacenar los *chunks* en cada *peer*.

En el *Anexo 3 Redes comerciales de streaming P2P* se comentan las distribuciones basados en mallas Octoshape y PPLive. Existen otros conocidos, tales como Tribler, PPStream y Zattoo que pueden ser consultados en [3] por más referencias.

2.2.2.1.3 Distribución híbrida (*Hybrid*)

Esta categoría de arquitecturas de *streaming* P2P incluye a todas las aplicaciones que no pueden ser clasificadas como basadas en árboles o basadas en mallas, y que a su vez presenta características de ambas categorías. Este tipo de topología además de ser mixta, presenta un modelo de tipo *pull-push*.

Dentro de esta categoría se encuentra la aplicación New Coolstreaming que se describe en el *Anexo 3 Redes comerciales de streaming P2P.*

2.2.2.2 Peer-to-Peer Streaming Protocol (PPSP)

El protocolo PPSP surge por parte del grupo de trabajo del PPSP (PPSP Working Group) con el fin de definir un protocolo estandarizado de *streaming* P2P para distribución de contenido *multimedia* bajo demanda o en vivo [32].

A modo de resumen, existen dos tipos de nodos en PPSP: los *trackers* y los *peers*. Los *últimos* son nodos que envían y reciben contenido *multimedia* que abarcan *hosts* conectados en forma estática y/o dinámica, por lo que pueden contar con dirección IP variable. Por otro lado, los *trackers* son entidades con conectividad estable que mantienen la meta-información del contenido *multimedia* del que se realiza el *stream*, y de los *peers* que se encuentran conectados.

El PPSP WG cuenta básicamente con dos protocolos. Existe un protocolo para el control y señalización entre *trackers* y *peers* (PPSP Tracker Protocol), y un protocolo para el control y señalización para la comunicación entre *peers* (PPSP Peer Protocol). Ambos protocolos permiten a los *peers* recibir *streaming* de datos dentro de restricciones de tiempo específicas. El protocolo del *tracker* maneja el intercambio inicial y periódico de meta-información entre *trackers* y *peers*, como por ejemplo la lista de *peers* conectados e información de contenido. El protocolo de los *peers* controla la publicación e intercambio de datos de disponibilidad del contenido *multimedia* entre los *peers*. Estrategias similares a las mencionadas fueron evaluadas para la implementación del prototipo de solución.

En el *Anexo 3 Redes comerciales de streaming P2P* se detalla la motivación, alcance y requerimientos del protocolo PPSP, como también una explicación de los dos protocolos que lo conforman.

2.2.2.3 GoalBit

GoalBit es la primera plataforma P2P de distribución de video de código abierto [3], cuya especificación se compone del GoalBit Transport Protocol (GBTP) y el Goalbit Packetized Stream (GBPS). El primero define la manera en la cual los *peers* intercambian el contenido, mientras que el segundo define la forma del contenido transportado.

GoalBit se basa en VLC[5] para la manipulación, codificación y reproducción de video, CTorrent[6] que implementa un cliente del protocolo BitTorrent y OpenTracker[7] que implementa un *tracker* también del protocolo BitTorrent.

La especificación de la arquitectura de GoalBit y de su funcionamiento se encuentra detallada en el *Anexo 4 GoalBit*.

2.3 Nuevas tecnologías de streaming

En esta sección se detallan algunas nuevas tecnologías que pueden cambiar la forma en se hace *streaming*. Éstas son:

- HTTP Live Streaming
- Lenguaje HTML5
- *Web Sockets*

[5] *VLC*. http://www.videolan.org/vlc/index.html. VideoLAN Organization. Julio de 2014.
[6] *CTorrent*. http://www.rahul.net/dholmes/ctorrent/. Julio de 2014.
[7] *OpenTracker*. http://erdgeist.org/arts/software/opentracker/. Julio de 2014.

- Media Source
- WebRTC

2.3.1 HTTP Live Streaming (HLS)

HTTP Live Streaming (HLS) es un protocolo de comunicación de *streaming multimedia* basado en el protocolo HTTP de capa de aplicación implementado por Apple [6][7]. El protocolo se encuentra como propuesta para ser estandarizado por la IETF y básicamente sigue una arquitectura de tipo cliente/servidor. A pesar de denominarse *Live Streaming*, HLS posee muchas ventajas para eventos en vivo y para videos bajo demanda.

Según HLS, una presentación *multimedia* se especifica por una URL a un archivo de *Playlist*, con formato M3U/M3U8 [7]. Este archivo se encuentra compuesto por una lista ordenada de URLs y *tags* de información, que juntos especifican una serie de segmentos *multimedia*.

Para reproducir el flujo, el cliente obtiene en primera instancia el archivo de *Playlist* a partir de la URL especificada y, posteriormente, obtiene y reproduce cada segmento *multimedia* del mismo. A su vez, debe refrescar el archivo de *Playlist* en forma periódica para descubrir segmentos adicionales. La arquitectura de HLS y la interacción entre un cliente y un servidor HLS se detalla en el *Anexo 5 HTTP Live Streaming*.

2.3.2 Lenguaje HTML5

HTML5 [1] consiste en la última versión del estándar del lenguaje HyperText Markup Language (HTML) que se encuentra aún en etapa de desarrollo. El término HTML5 suele comprender dos conceptos: la nueva versión del lenguaje HTML4 que comprende nuevos elementos, atributos y comportamientos, como también un amplio conjunto de tecnologías que permite la creación de aplicaciones y páginas *web* más diversas y poderosas.

Esta versión del lenguaje HTML resulta muy novedosa debido a su incursión en todo lo que refiere al área de *multimedia*. Más específicamente, se proveen de los *tags* <audio> y <video> que permiten reproducir audio y video en un documento respectivamente. A su vez, la tecnología WebRTC que forma parte del estándar permite obtener un flujo de audio y/o video desde un dispositivo (PC, *smartphone*, etc) o desde un cliente remoto para ser reproducido en el documento también.

En el *Anexo 6 Lenguaje HTML5* se explica la historia y evolución del lenguaje, como también detalles importantes de compatibilidad de los nuevos componentes del lenguaje en los distintos navegadores.

2.3.3 *Web Sockets*

Como se detalla en el *Anexo 6 Lenguaje HTML5*, HTML5 introduce un gran número de APIs que revolucionan la forma en que funciona la Internet hoy en día. Hasta hace poco tiempo, los navegadores soportaban el modelo de conexión de tipo cliente/servidor, como es el conocido caso del protocolo HTTP. Una de las nuevas APIs provistas por HTML5 es la de los *web sockets* [18], que podría suplantar a futuro la tecnología Asynchronous JavaScript + XML (AJAX)[8].

[8] *Asynchronous JAvaScript + XML (AJAX)*. https://developer.mozilla.org/en/docs/AJAX. Mayo de 2013.

Los *web sockets* constituyen una API [15] y también un protocolo que surgió hace relativamente poco tiempo, y que permite abrir una sesión de comunicación interactiva entre un navegador (cliente) y un servidor. Con esta API, el cliente y el servidor pueden enviarse y recibir mensajes con una orientación a eventos, sin la necesidad de que el cliente deba realizar *polling*.

Una ventaja sustancial que presentan los *web sockets* frente a la tecnología AJAX, consiste en la notoria disminución del *overhead* de los paquetes HTTP, que son demasiado grandes para aplicaciones que requieren de baja latencia. La tecnología WebSocket permite bajar la latencia, debido a que, una vez establecida una conexión por *web sockets*, el servidor puede enviar mensajes al cliente apenas se encuentre disponible para hacerlo (en forma asíncrona). Por ejemplo, a diferencia del *polling*, los *web sockets* realizan un solo *request*, y tanto el cliente como el servidor pueden enviarse mensajes en cualquier momento.

En el *Anexo 7 Web Sockets* se detalla la arquitectura y funcionamiento del protocolo WebSocket.

2.3.4 Media Source

Media Source es una API de JavaScript [19] que permite construir en forma dinámica flujos de datos *multimedia* para los *tags* <video> y <audio> de HTML5. Esta API define objetos que permiten al laneguaje JavaScript pasar segmentos de datos *multimedia* a un elemento HTMLMediaElement (del cual derivan HTMLVideoElement y HTMLAudioElement) como se puede ver en la Figura 4.

En la especificación de la API se incluye un modelo de *buffering* para describir cómo los navegadores deben actuar al momento de concatenarse distintos segmentos de datos *multimedia* en distintos momentos.

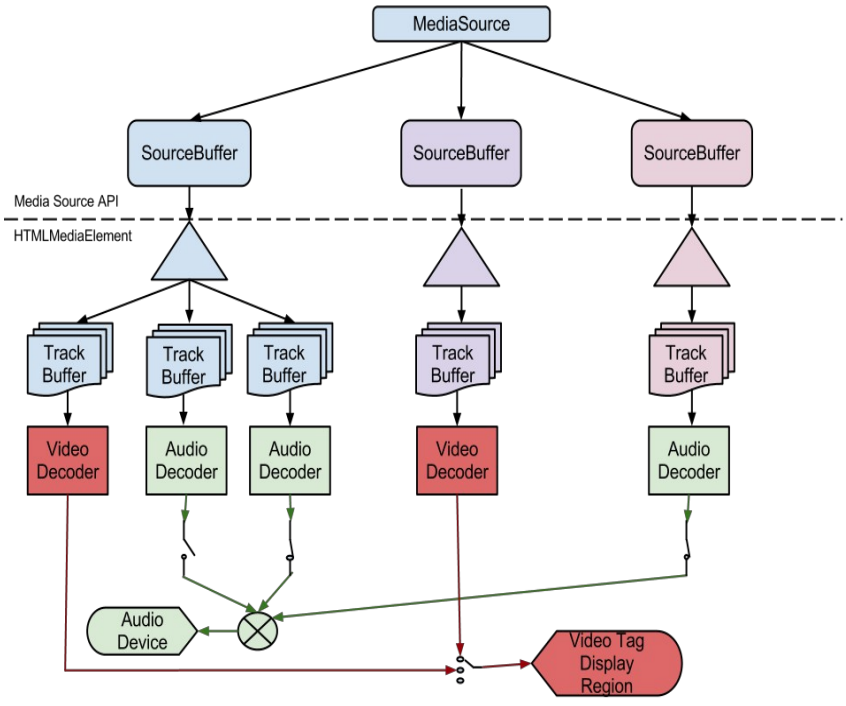

Figura 4. Extensión Media Source de <video> y <audio> [19].

En la especificación de Media Source se incluye también una especificación de WebM, ISO Base Media File Format, y MPEG-2 Transport Streams a modo de detallar el formato esperado de los flujos de *bytes* usados con estas extensiones (Para el caso de estudio desarrollado, el interés se encuentra sobre la especificación de WebM).

En la Tabla 1 se pueden ver los navegadores que soportan Media Source. Cabe destacar que la implementación de la API se encuentra totalmente implementada solamente para el navegador Chrome por el momento, el cual también es el único navegador que soporta el formato WebM. Esto restringió en gran manera en la etapa de implementación del prototipo.

Un objeto Media Source representa una fuente de datos *multimedia* para un HTMLMediaElement. El mismo mantiene un seguimiento del atributo *readyState* para esta fuente, como también una lista de objetos de tipo SourceBuffer que pueden ser utilizados

	IE	**Firefox**	**Chrome**	**Safari**	**Opera**	**iOS Safari**	**Android Browser**
Versión de navegador	11+ (parcial)	-	30+	-	-	-	Jellybean KitKat

Tabla 1. Soporte de Media Source por navegador [19]

para agregar datos *multimedia* a la presentación.
Los objetos Media Source son creados por la aplicación *web* y luego concatenados a un HTMLMediaElement. La aplicación utiliza los objetos SourceBuffer para agregar datos *multimedia* a la fuente. El HTMLMediaElement hace *fetch* de datos *multimedia* directamente desde el objeto Media Source cuando sea necesario durante la reproducción.

Un objeto Media Source puede ser concatenado a un elemento *multimedia* al asignar una URL para dicho objeto en el atributo *src* de un *tag* <video> o <audio>, o en el atributo *src* de un <source> dentro un elemento *multimedia*.

2.3.5 WebRTC

Web Real-Time Communications (RTC) [20], más conocido como WebRTC, consiste en una tecnología aún en proceso de borrador por parte de la World Wide Web Consortium (W3C) y de la IETF, que provee comunicación de video y audio en tiempo real entre navegadores en forma directa.

Por primera vez, los navegadores pueden interactuar entre sí directamente. Estas capacidades de comunicación son provistas a través del uso de los nuevos *tags* de HTML5 y APIs de JavaScript.

A modo de comparación, la gran mayoría de funcionalidades provistas por Skype[9] son también provistas por WebRTC en Chrome, pero sin la necesidad de instalar ningún *software* adicional ni *plugin*. Todas las funcionalidades de WebRTC se encuentran implementadas en forma nativa en el navegador. Para que una página *web* o aplicación *web* permita este tipo de funcionalidad (sin importar el navegador en el cual sea cargada), se precisa desarrollar un estándar [20].

[9] *Skype*. http://www.skype.com/en/. Mayo de 2013.

Históricamente, la comunicación en tiempo real (RTC) fue corporativa y complicada de implementar, requiriendo de tecnologías de audio y de video de licenciamiento costoso. Uno de los actores que más incursionó en esta área en la *web* últimamente fue Google, a través del *chat* de video provisto por Gmail.

Este *chat* se volvió exitoso a partir de 2008, siendo en 2011 que introdujo su aplicación Google Hangouts[10], que utiliza el servicio Google Talk[11] por debajo. Posteriormente, Google compró Global IP Solutions (GIPS)[12], una compañía que se dedicaba al desarrollo de componentes *multimedia* requeridos para RTC, tales como técnicas de codificación *multimedia* y afines. Google hizo públicas estas tecnologías desarrolladas por GIPS y se puso en contacto con actores de la W3C y la IETF para llegar a un consenso en esta área. En mayo de 2011, Ericsson Labs[13] construyó la primer implementación de WebRTC.

Actualmente, WebRTC lleva implementado estándares abiertos para comunicación de datos, video y audio en tiempo real y sin necesidad de uso de *plugins*. La necesidad de esta tecnología por parte de los usuarios y de las empresas parece ser grande hoy en día:

- Muchas páginas y aplicaciones *web* ya cuentan con RTC, pero precisan de *plugins* o instalación de aplicaciones nativas. Entre varios se incluyen Skype, Facebook (que utiliza Skype), y Google Hangouts (que utiliza el *plugin* de Google Talk).
- La descarga, instalación y actualización de *plugins* puede resultar tediosa y compleja por parte de los usuarios, lo cual puede resultar en el abandono de la tecnología seleccionada.
- Los *plugins* pueden resultar difíciles de liberar, de *debuggear*, de realizarle *testing* y de mantener por parte de los desarrolladores. A su vez, pueden requerir de licencias e integración de tecnología cara y compleja. La mayor parte de las veces resulta difícil persuadir a los usuarios de instalar *plugins* para poder correr aplicaciones.
- La seguridad es otro aspecto fundamental que buscan los usuarios para proteger los datos que se comunican. WebRTC encripta toda la información enviada y recibida por parte de los participantes.

La misión de WebRTC según su página *web* oficial [21], es la de construir una API que debe ser abierta, libre, estandarizada, simple de usar, integrada en forma nativa en los navegadores, y ser la más eficiente de las tecnologías de comunicación de audio y video en tiempo real.

En la secciones siguientes se explica cómo funciona WebRTC, los protocolos que utiliza por debajo, las APIs que implementa, y los desafíos que enfrenta. Esta documentación se basa fuertemente en las publicaciones [22][23][24][25].

2.3.5.1 Elementos participantes en un sistema WebRTC

Para realizar una comunicación en tiempo real a través de la *web* de manera exitosa, es necesaria la interacción de distintos tipos de componentes. Como mínimo, deben coexistir dos o más nodos que se quieran comunicar de alguna forma. Dichos nodos deben estar conectados a la red y ambos deberán estar corriendo algún programa que implemente y

[10] *Google Hangouts*. http://www.google.com/+/learnmore/hangouts/. Junio de 2013.
[11] *Google Talk*. https://support.google.com/talk/topic/1186?hl=en&ref_topic=4509884. Junio de 2013.
[12] *Google IP Solutions (GIPS)*. http://www.gipscorp.com/. Junio de 2013.
[13] *Ericsson Labs*. https://labs.ericsson.com/. Junio de 2013.

utilice los estándares necesarios de WebRTC para que sirva como interfaz para la comunicación.

Dicho programa podría estar corriendo en diferentes tipos de dispositivos, tanto en una computadora de escritorio, como en un teléfono celular. Incluso existen elementos adicionales, como *gateways* para interconectar el tráfico WebRTC con la red de telefonía pública, u otro tipo de herramientas de Internet como teléfonos Session Initialization Protocol (SIP)[14] o clientes Jingle[15] (protocolos de señalización para comunicación en tiempo real) utilizados hoy en día para realizar comunicaciones VoIP o multi-conferencias, que también son propensos a ser utilizados en implementaciones de WebRTC.

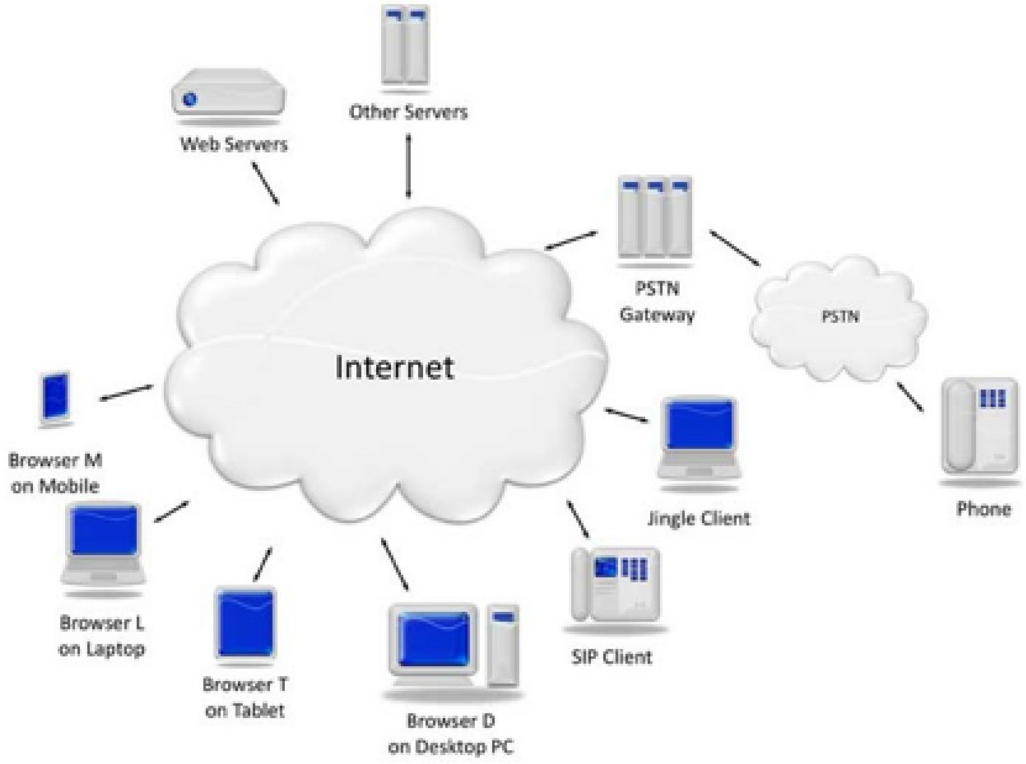

Figura 5. Elementos en un ambiente WebRTC [25].

En particular, interesa estudiar el caso más común: dos o más dispositivos corriendo la misma aplicación WebRTC en un navegador de una computadora, teléfono celular o *tablet*.

En el caso en que los dispositivos que quieran comunicarse sean exactamente dos, se produce el denominado Triángulo WebRTC (Figura 6). En este caso, ambos nodos realizan inicialmente un pedido HTTP al mismo servidor *web* a través de un navegador, del que se responde un contenido HTML y un programa Javascript que hará uso de la API WebRTC del navegador para establecer una conexión P2P con el otro punto.

[14] *SIP: Session Internet Protocol*. RFC 3261. https://www.ietf.org/rfc/rfc3261.txt. Junio de 2002.
[15] *Jingle Protocol*. Draft IETF. http://xmpp.org/extensions/xep-0166.html. 28 de Febrero de 2012.

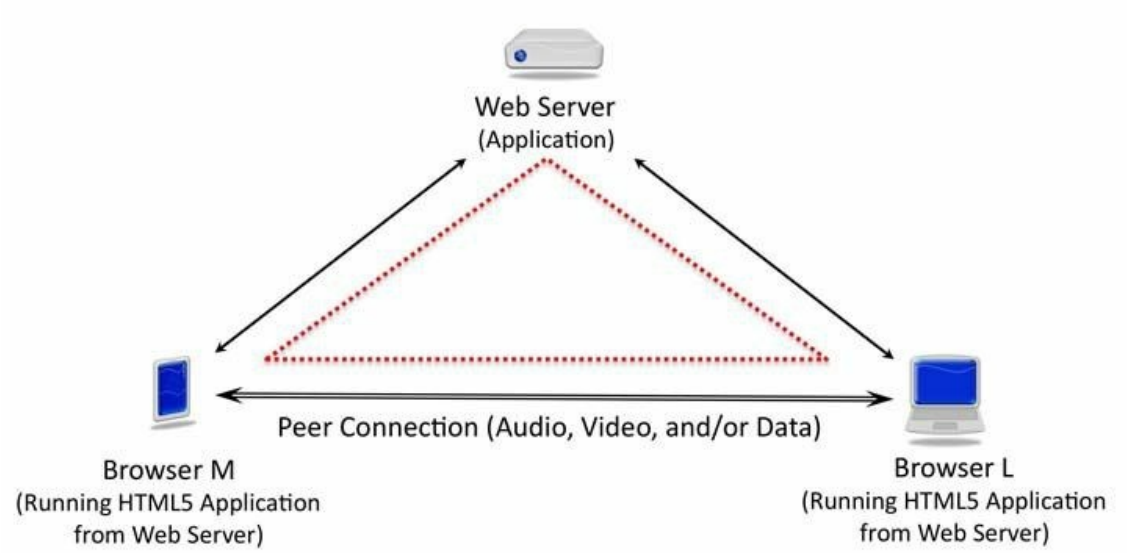

Figura 6. Triángulo WebRTC [25].

Por lo tanto, el modelo más simple para llevar a cabo una comunicación en tiempo real consiste de un servidor *web*, un programa JavaScript y un navegador que soporte WebRTC. La interacción entre dichos componentes puede verse en la Figura 7.

Lo novedoso de esta arquitectura es el componente del navegador *Browser RTC Function* y su habilidad para interactuar con el código JavaScript. El mismo accede a las funciones RTC a través de la API definida, y a su vez, el núcleo RTC accede a los recursos del sistema operativo a través del navegador.

Otro aspecto importante es la interacción que ocurre de navegador a navegador, conocida como Peer Connection, donde el *Browser RTC Function* del primer navegador se comunica con su contraparte utilizando ciertos protocolos (distintos de HTTP) para transmitir audio, video y/o datos. En principio, no hay restricciones en cuanto al protocolo de capa de transporte a utilizar para mantener la conexión, por lo que es viable utilizar tanto TCP como UDP.

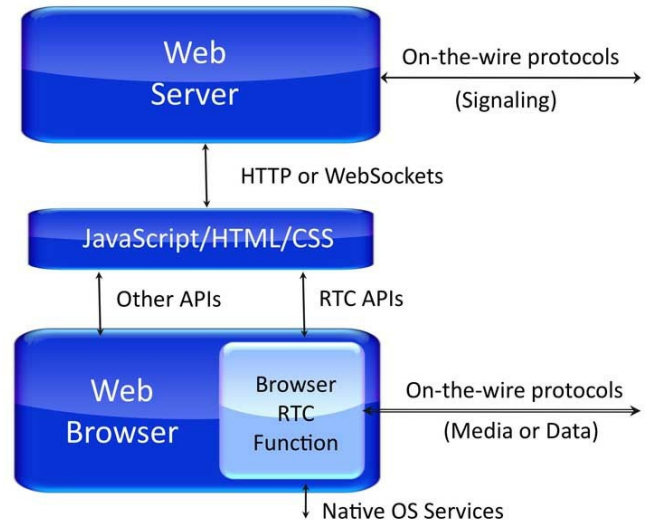

Figura 7. El modelo del navegador [25]

Sin embargo, una de las propiedades más interesantes de un comunicación RTC es la capacidad de interactuar, no sólo con dos, sino con múltiples clientes simultáneos. Por ejemplo, para realizar video-conferencias o para compartir flujos de audio y video. Se plantean dos tipos de disposiciones para solucionar este problema: la denominada *Full Mesh* y la llamada *Centrally Mixed*, que se encuentran explicadas en el *Anexo 9 WebRTC*.

2.3.5.2 Protocolos involucrados en un sistema WebRTC

Protocolo	**Uso**	**Especificación**
HTTP	Hyper-Text Transfer Protocol	RFC 2616
SRTP	Secure Real-time Transport Protocol	RFC 3711
SDP	Session Description Protocol	RFC 4566
ICE	Interactive Connectivity Establishment	RFC 5245
STUN	Session Traversal Utilities for NAT	RFC 5389
TURN	Traversal Using Relays around NAT	RFC 5766
TLS	Transport Layer Security	RFC 5246
TCP	Transmission Control Protocol	RFC 793
DTLS	Datagram Transport Layer Security	RFC 4347
UDP	User Datagram Protocol	RFC 768
SCTP	Stream Control Transport Protocol	RFC 2960
IP	Internet Protocol	RFC 791, RFC 2460

Tabla 2. Protocolos utilizados en un sistema WebRTC [25]

Hay un gran número de protocolos en los que WebRTC se apoya para llevar a cabo su misión. Los más importantes se encuentran listados en la Tabla 2.
La Figura 8 muestra un esquema de los niveles en los que interactúan cada uno de los protocolos listados, en relación a la pila de capas que operan en la *web*.

A lo largo de esta sección se discuten algunas propiedades de los protocolos anteriormente listados. En el *Anexo 9 WebRTC* se detalla muy brevemente algunas de las características de los mismos que pareció más importante destacar (estándares como HTTP, TCP, UDP e IP se asumen conocidos y no se discuten).

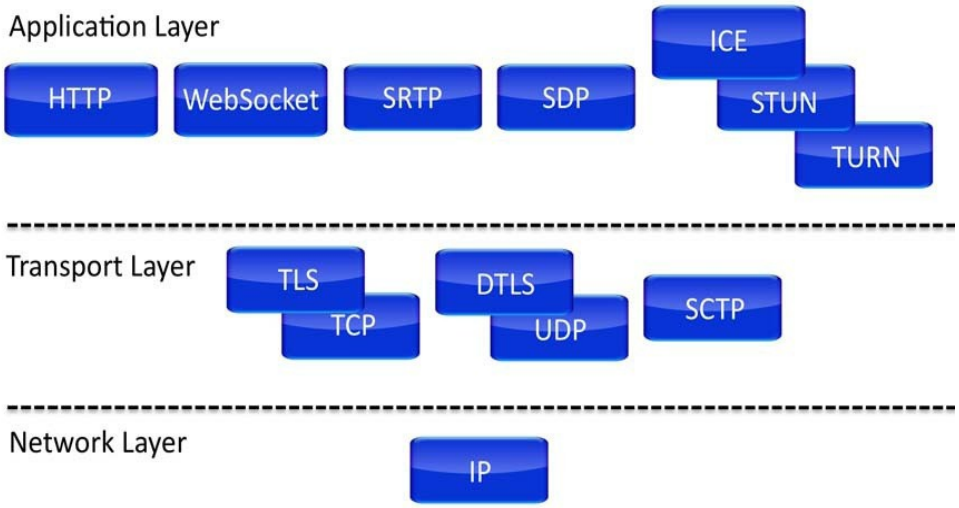

Figura 8. Protocolos involucrados en un ambiente WebRTC [25]

2.3.5.3 Sesión WebRTC

Para que dos nodos puedan comenzar a transmitir un flujo de datos, lo primero a realizar es configurar la sesión WebRTC. La idea consiste en prescindir del servidor *web* para realizar el intercambio de datos una vez configurada la sesión. Para esto se requiere que un número de mensajes se envíen entre el cliente y el servidor, pero también entre los *peers*.

Un desarrollador de aplicación WebRTC que desee realizar una comunicación *multimedia* entre *peers*, deberá seguir los siguientes pasos para configurar una sesión WebRTC:

1. Obtener datos locales *multimedia* (por ejemplo, es posible obtener un audio desde el micrófono de la computadora).
2. Generar una conexión entre el navegador y otro cliente (otro navegador o cualquier otro tipo de cliente).
3. Adjuntar datos *multimedia* y *data channels* a la conexión.
4. Una vez finalizado el flujo, cerrar la conexión.

En la Figura 9 se pueden apreciar los pasos anteriores. Para estos últimos, WebRTC provee la implementación de las siguientes tres APIs:

- MediaStream, también conocida como getUserMedia [20][22][26]: permite el acceso a datos de *streaming* locales, tales como el audio del micrófono y/o el video de la cámara.
- RTCPeerConnection [20]: permite establecer una conexión *multimedia* entre *peers*.
- RTCDataChannel [20]: permite crear conexiones de datos genéricos entre *peers*.

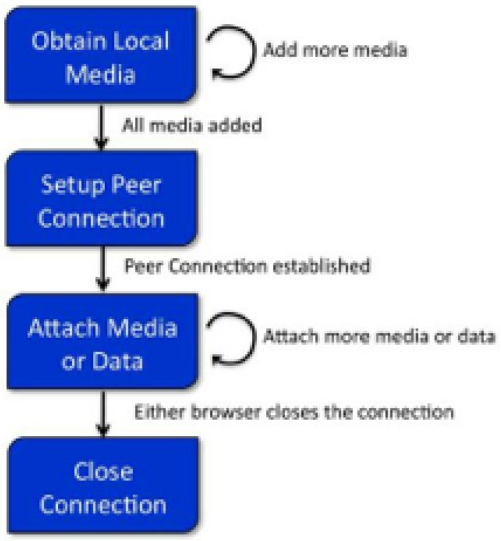

Figura 9. Establecimiento de una sesión WebRTC vista desde la API [25]

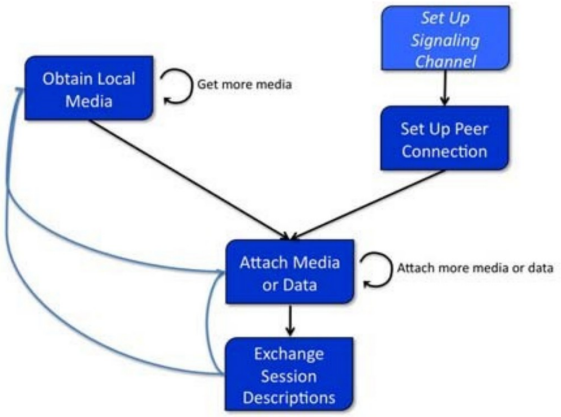

Figura 10. Establecimiento de una sesión WebRTC vista desde la señalización [25]

Estas APIs se describen a continuación, a medida que se expliquen los pasos involucrados en el establecimiento de una sesión WebRTC (a excepción de RTCDataChannel que se detalla en la Sección *2.3.6 Data Channels*).

2.3.5.3.1 Obtención de datos *multimedia* locales

La API MediaStream de WebRTC permite la captura de flujos de audio y de video desde el propio navegador. Éstos se pueden obtener desde el micrófono y la cámara de la computadora respectivamente.

A lo largo del tiempo se han implementado variadas soluciones para este problema que requerían la instalación de *plugins* tales como Microsoft Silverlight[16] o Adobe Flash. Sin embargo, esta API facilita el acceso a estos recursos sin el requerimiento de instalar nada adicional, se provee acceso a los flujos únicamente mediante código JavaScript.

MediaStream se basa en la manipulación de objetos MediaStream de JavaScript que representan una sola fuente de video o audio (o ambas) sincronizada (ver Figura 11). Cada uno de estos objetos contiene uno o más objetos de tipo MediaStreamTrack. Cada uno de estos objetos puede tener a su vez uno o más canales. Un canal representa la menor unidad posible de un flujo de datos *multimedia*, como por ejemplo, una señal de audio asociada con un parlante.

Cada objeto MediaStream tiene una única fuente de entrada y una única fuente de salida. Estos objetos son creados a partir del método getUserMedia, que es el que permite el acceso a los dispositivos de audio y video de una computadora desde el navegador.

[16] *Microsoft Silverlight*. http://www.microsoft.com/silverlight/. Junio de 2013.

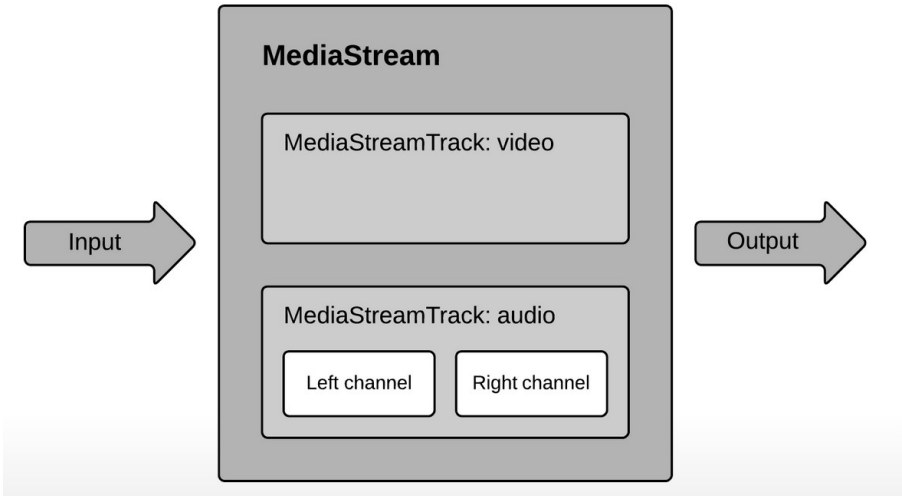

Figura 11. Objeto MediaStream [25]

Los objetos MediaStream catalogados como “locales”, son aquellos que tienen su origen a partir de un micrófono o de una cámara de video. Por otro lado, los MediaStream “no locales” son aquellos que representan a un elemento *multimedia*, tales como los *tags* <video> y <audio>, o un flujo obtenido a través de una Peer Connection de WebRTC, por ejemplo.

Para que el método getUserMedia pueda acceder a los recursos *multimedia* tales como el micrófono y/o la cámara de video, se requiere que el navegador pida permiso al usuario para acceder a dichos dispositivos. Una vez accedido a los mismos, se procede a la creación de los objetos MediaStream. La API es enteramente orientada a eventos y muy sencilla de utilizar.

Por información más detallada de getUserMedia, incluida la creación de los objetos MediaStream y la comunicación de los mismos a través de una Peer Connection, se sugiere recurrir a [25], capítulo 3.

2.3.5.3.2 Conexión entre dos *peers*

WebRTC provee de la API RTCPeerConnection para crear la Peer Connection entre dos *peers*. En este contexto, dos *peers* refieren a dos nodos finales conectados a la *web*. En lugar de realizar la comunicación a través de un servidor intermedio, es posible realizarla directamente entre los *peers*.

RTCPeerConnection

RTCPeerConnection consiste en la API de WebRTC que permite una comunicación estable y eficiente de flujo entre dos o más *peers*. En la Figura 12 se muestra un diagrama de la arquitectura de WebRTC y el rol que cumple en la misma.

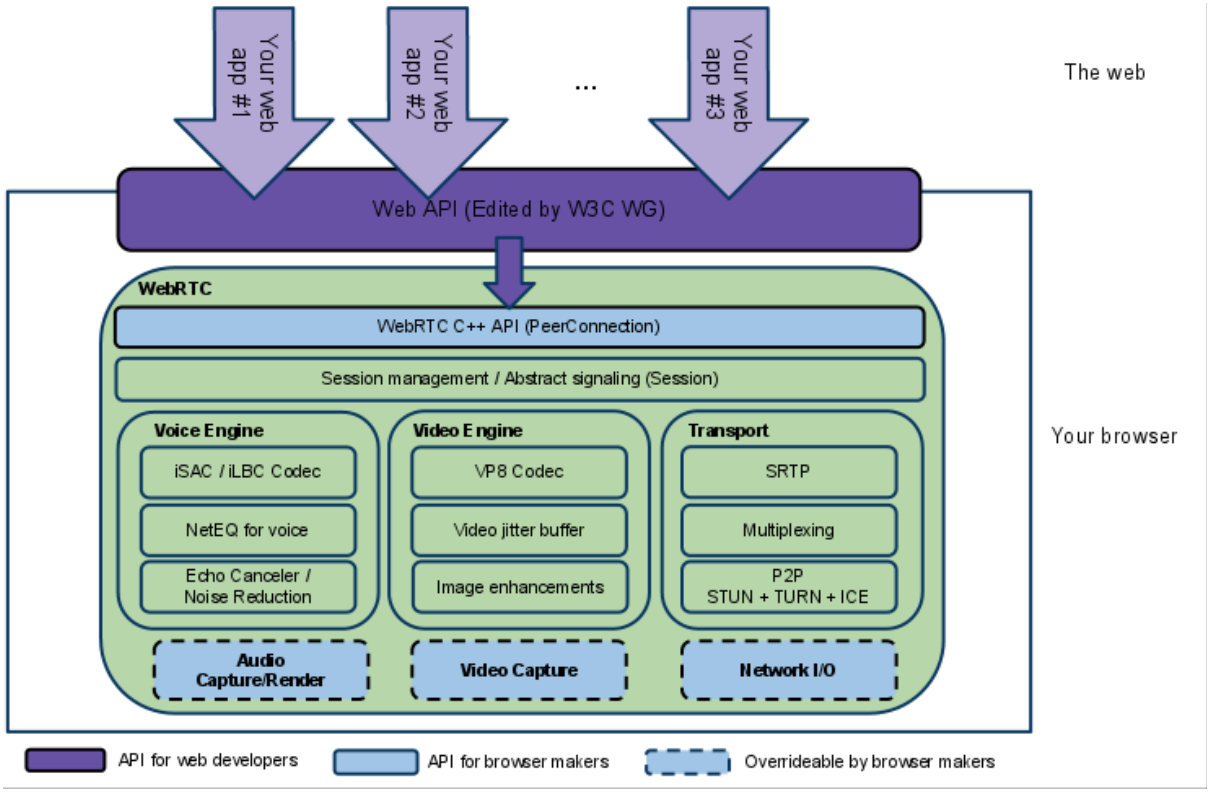

Figura 12. Arquitectura de WebRTC [25]

Desde la perspectiva de la API JavaScript de Peer Connection, lo fundamental de la Figura 12 consiste en que la misma protege al desarrollador de las innumerables complejidades que se encuentran por debajo de la misma (ver parte WebRTC de la Figura 12). Los *codecs* y protocolos utilizados por WebRTC realizan mucho trabajo para permitir la comunicación en tiempo real, incluso sobre redes no confiables:

- Encubrimiento de pérdidas de paquetes.
- Cancelación de eco.
- Adaptabilidad del ancho de banda.
- *Buffering* dinámico del *jitter*.
- Reducción y supresión del ruido.

ICE para lidiar con NAT y *Firewalls*

A pesar de que a la tecnología WebRTC se la califica por varios como una tecnología *server-less*, en la realidad precisa de servidores que provean las siguientes funcionalidades:

- Descubrimiento de *peers*.
- Señalización (se explica detalladamente en el *Anexo 9 WebRTC*).
- NAT/FireWall Traversal.
- Servidores de *relay* que permitan la comunicación, en caso de que la comunicación P2P falle.

En la realidad, la mayor parte de los dispositivos se encuentran por detrás de una NAT, o poseen antivirus o *firewalls* que bloquean ciertos puertos o protocolos (ver Figura 13).

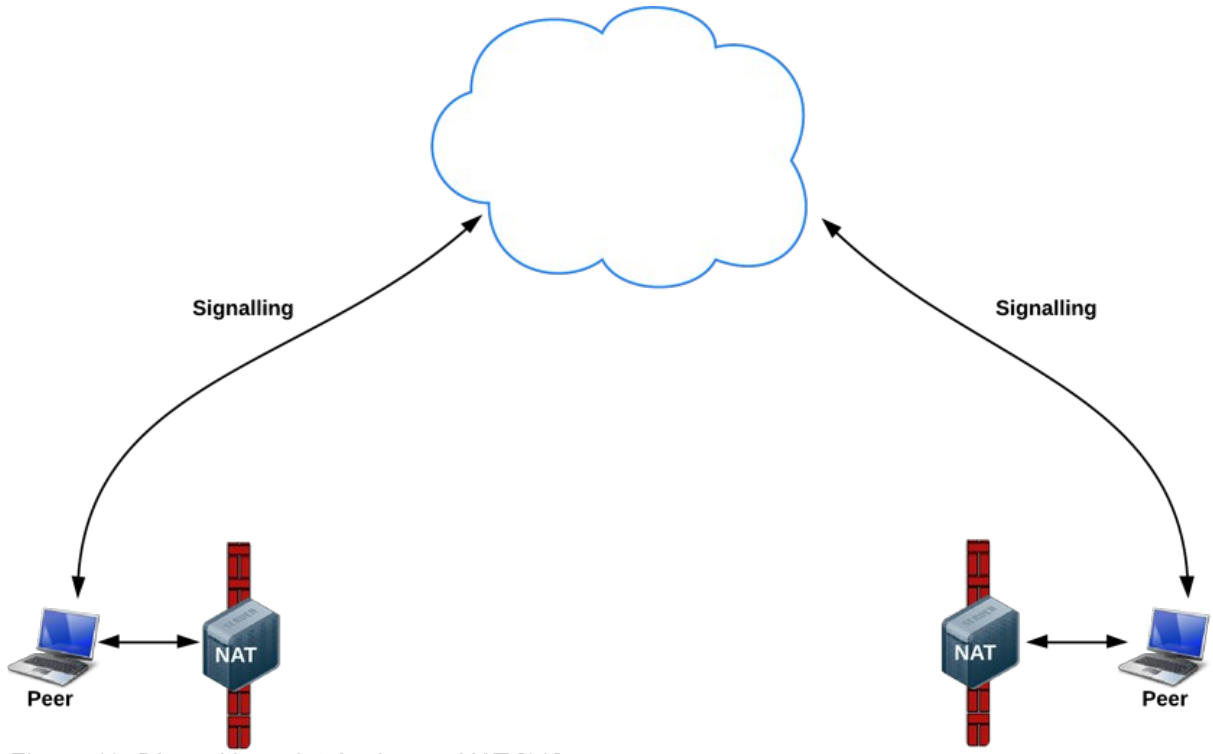

Figura 13. Dispositivos detrás de una NAT [22]

Como fue mencionado en la Sección *2.3.5.2 Protocolos involucrados en un sistema WebRTC*, WebRTC reutiliza protocolos ya definidos para solucionar este problema. El protocolo STUN y su extensión TURN son utilizados por el protocolo ICE para establecer la Peer Connection que enfrente el problema del NAT Traversal.

NAT provee a un dispositivo de una dirección IP privada para ser utilizada dentro de la red local privada, pero la misma no puede ser utilizada externamente. Sin una dirección pública, WebRTC se encuentra impedido de realizar la comunicación directa entre *peers*.

ICE es un protocolo que implementa la técnica Hole Punching (ver el *Anexo 9 WebRTC*). Se utiliza para traspasar las tablas NAT y lograr generar conexiones con nodos que pertenezcan a una red privada. Utiliza los protocolos STUN y TURN para lograr dicho cometido [27]. En el *Anexo 9 WebRTC* se explica más detalladamente cómo ICE gestiona las direcciones candidatas para establecer la conexión entre navegadores.

El procedimiento de ICE consiste básicamente en encontrar la mejor vía para comunicar a los *peers* que desean establecer una conexión entre ellos. ICE intenta todas las opciones en forma paralela y elige aquella que sea más óptima. En primer lugar intenta crear la conexión utilizando la dirección del *host* obtenida a partir del sistema operativo del dispositivo y de la tarjeta de red del mismo. En caso de fallar (el dispositivo se encuentra detrás de una NAT), ICE obtiene la dirección IP pública y puerto a través de un servidor STUN, y en caso de fallar nuevamente, el tráfico es ruteado finalmente a través de un servidor TURN.

En resumen:

- Se utiliza un servidor STUN para obtener la dirección IP pública y puerto del *peer*.

- Se utiliza un servidor TURN como *relay* para dirigir el tráfico, en caso de que la comunicación P2P no sea posible.

Por lo tanto, el desarrollador puede agregar en la configuración de la Peer Connection la dirección de los servidores STUN o TURN que necesite para realizar la conexión. Por ejemplo, Google provee de un servidor STUN público (stun:stun.l.google.com:19302) que fue utilizado para realizar pruebas en la fase de implementación.

STUN

A modo atravesar la NAT, WebRTC utiliza el protocolo STUN como se puede ver en la Figura 14.

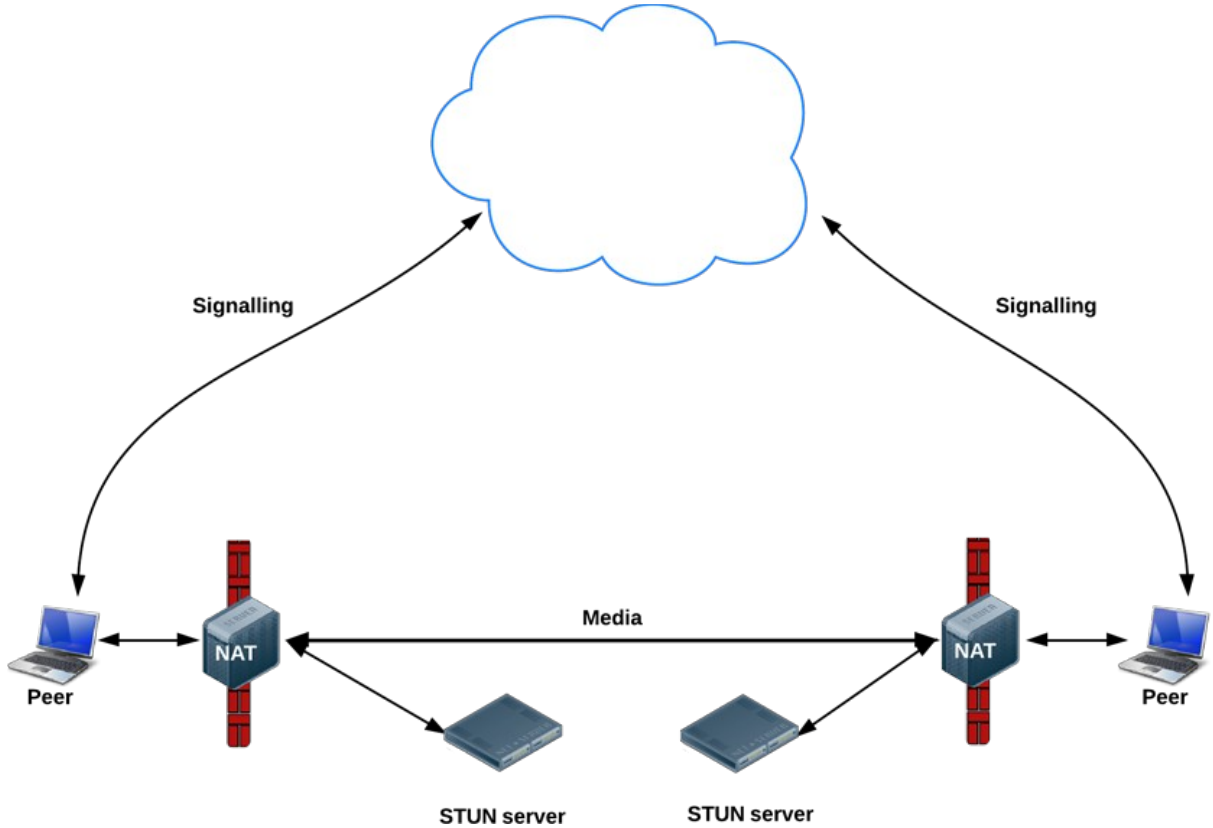

Figura 14. Uso de servidores STUN para obtener direcciones IP:puerto públicas [22]

Los servidores STUN se encuentran en forma pública en Internet y cumplen un simple cometido: chequear la dirección IP pública y puerto de un pedido entrante (de una aplicación que corre detrás de una NAT), y enviar esa dirección y puerto como respuesta. En otras palabras, la aplicación que corre en el *peer* detrás de la NAT utiliza el servidor STUN para conocer su dirección IP y puerto públicos.

Este proceso le permite conocer a un *peer* en WebRTC su dirección pública y pasarla a otros *peers* a través de un mecanismo de señalización, a modo de contar con una comunicación directa. Cabe destacar que en la práctica pueden existir múltiples capas de NATs, y que no todas las NATs funcionan de la misma forma, pero el principio sigue siendo el mismo.

La mayoría de las llamadas que se realizan utilizando WebRTC resultan exitosas utilizando STUN: 86% según [28], aunque este porcentaje puede ser menor para aplicaciones que se encuentran detrás de *firewalls* y NATs más complejos.

TURN

La API RTCPeerConnection intenta establecer una comunicación directa entre *peers* sobre UDP. En caso de fallar, la API intenta establecer nuevamente la comunicación, pero esta vez sobre TCP (HTTP en primer lugar, luego HTTPS). En caso de fallar nuevamente, se utilizan los servidores TURN como última opción para mantener la comunicación entre *peers*. Cabe destacar que los servidores TURN son utilizados como *relay* de flujo de video, audio y/o datos entre *peers*, pero no para señalización entre los mismos.

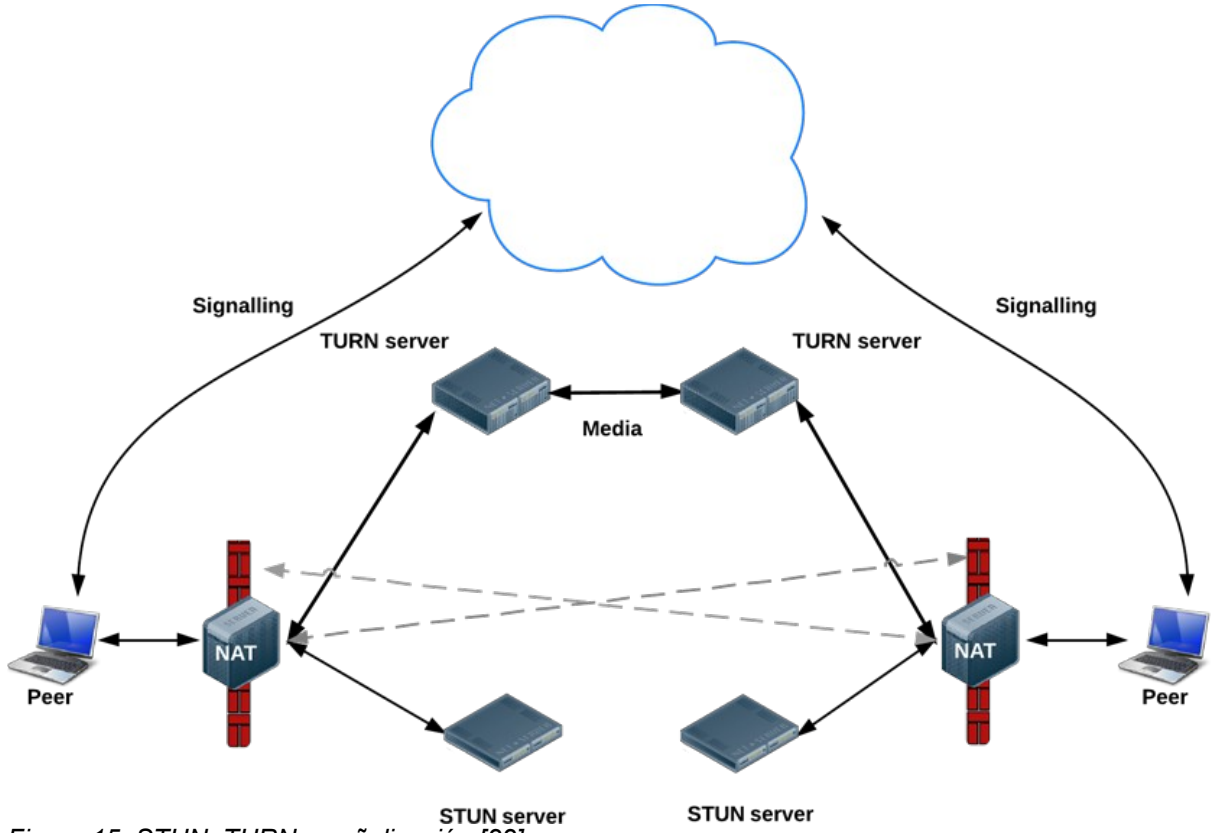

Figura 15. STUN, TURN y señalización [22]

Los servidores TURN tienen direcciones IP públicas, lo que permite que los mismos sean contactados por *peers*, incluso si los mismos se encuentran detrás de *Firewalls* o *proxies*. Los servidores TURN tienen el simple cometido de ser un servidor de *relay* de *streaming*, pero a diferencia de los servidores STUN, consumen mucho ancho de banda. En consecuencia, estos servidores requieren ser potentes.

La Figura 15 muestra cómo, según el procedimiento de ICE, un *peer* intenta comunicarse con otro. Dado que el uso de servidores STUN no fue exitoso, los *peers* terminan utilizando servidores TURN para transmitir el *streaming*.

2.3.5.3.3 Concatenación de datos *multimedia*

Una vez que la conexión entre *peers* se encuentra establecida, el *peer* local puede adjuntar cualquier número de flujos de datos *multimedia* locales a la Peer Connection para ser enviados a través de la conexión al navegador remoto. Similarmente, cualquier número de

flujos de datos *multimedia* remotos pueden ser recibidos por el navegador local, resultando en nuevos flujos que pueden ser manipulados por el *peer* local.

Es importante notar que cada intercambio de requerimientos de datos *multimedia* requiere de una negociación (o renegociación) entre navegadores acerca de cómo los datos *multimedia* serán representados en el canal. Cuando se realiza un pedido de un navegador a otro, ya sea para agregar o remover datos *multimedia*, el navegador debe generar un objeto SessionDescription de JavaScript apropiado (un contenedor para una *session description* - información de cómo establecer una sesión *multimedia*) para representar el conjunto completo de datos *multimedia* que fluye por la Peer Connection. La API de RTCPeerConnection provee de formas para que el desarrollador pueda ver y editar el SessionDescription antes de ser enviada al otro navegador.

Una vez que los navegadores intercambiaron los objetos SessionDescription, la sesión *multimedia* o datos de sesión pueden ser establecidos. Ambos navegadores comienzan con el Hole Punching. Una vez finalizado puede comenzar la negociación de la clave para la sesión segura de datos *multimedia*. Finalmente, la sesión *multimedia* o de datos puede comenzar.

2.3.5.3.4 Cierre de conexión

Cualquiera de los dos navegadores que se encuentran conectados pueden cerrar la conexión. La aplicación que corre en el navegador simplemente llama al método *close* de la API RTCPeerConnection para indicar que ha terminado de usar la conexión (ya sea porque el usuario hizo *click* sobre un botón o cambió de pestaña, por ejemplo). Esto ocasiona que el procesamiento de ICE y el flujo de datos *multimedia* se vean detenidos.

En forma similar, cuando un navegador pierde la conexión a Internet o *crashea*, los mensajes de tipo *keep-alive* enviados por el canal de datos *multimedia* o por el *data channel* fallarán. En consecuencia, el otro navegador intentará recomenzar el Hole Punching, y cuando el mismo falle cerrará la conexión. Una vez que la sesión termina, el navegador remueve los permisos obtenidos sobre los dispositivos de micrófono y cámara, por lo que una nueva sesión requerirá de nuevos permisos para su acceso por parte del usuario.

2.3.6 *Data Channels*

Los *data channels* conforman una funcionalidad clave especificada por WebRTC a través de la API RTCDataChannel [29]. Esta API provee un canal de comunicación bidireccional de intercambio de datos que corre sobre una Peer Connection. A diferencia de los *web sockets* y AJAX, que son tecnologías diseñadas para la comunicación entre un navegador y un servidor (o entre dos servidores), la comunicación a través de *data channels* es directa entre navegadores.

Además, RTCDataChannel utiliza el protocolo SCTP visto en la Sección *2.3.6 Data Channels*, que permite la configuración de retransmisiones y envío fuera de orden de paquetes (ver Tabla 3).
RTCDataChannel es actualmente soportada por los navegadores Chrome y Firefox (este último en forma parcial). En su núcleo, la API se encuentra diseñada muy similarmente a la de los *web sockets*. Soporta un conjunto flexible de tipos de datos para ser transmitidos por el canal: *strings*, y algunos tipos binarios del lenguaje JavaScript como Blob, ArrayBuffer y

	TCP	UDP	SCTP
Confiabilidad	confiable	no confiable	configurable
Envío	confiable	no confiable	configurable
Transmisión	orientada a byte	orientada a mensaje	orientada a mensaje
Control de flujo	si	no	si
Control de congestión	si	no	si

Tabla 3. Comparación de protocolos TCP, UDP y SCTP [29]

ArrayBufferView. Los tipos binarios resultan muy útiles al momento de lidiar con aplicaciones del tipo de intercambio de datos y juegos multi-jugador.

La especificación de RTCDataChannel aclara que los *data channels* pueden ser de tipo confiable o no confiable:

- *Data Channels* confiables: garantizan la transmisión de mensajes y el orden en que son enviados. Esto conlleva una sobrecarga adicional, además de un enlentecimiento de su funcionamiento.
- *Data Channels* no confiables: no garantizan la transmisión de mensajes ni el orden en que arriben a destino. Consecuentemente, este tipo de canal es más rápido que el confiable.

Un aspecto importante de la especificación de RTCDataChannel es que la API utiliza encriptación DTLS para asegurar que los paquetes que se envían a través del *data channel* se encuentren completamente encriptados en su camino a su destino.

2.4 Trabajos relacionados

2.4.1 Peer5

Peer5[17] es una compañía que surge con el fin de proveer a los navegadores *web* de las capacidades de transmisión de datos P2P. Desde los comienzos de la empresa en 2011, el grupo fundador encabezado por Hadar Weiss, actual CTO de la misma, optó por utilizar la tecnología WebRTC.

Ésta aún se encontraba lejos de la posibilidad de convertirse en un estándar. Los navegadores Chrome y Firefox no habían implementado aún la API Peer Connection de la tecnología (ver la Sección *2.3.5.3.2 Conexión entre dos peers*). La decisión consistió en utilizar una implementación de la API desarrollada por Ericsson[18] que personalizaba el navegador Epiphany e incluía una API propietaria para la transferencia de datos.

El resultado consistió en un primer producto de transferencia de datos P2P que demostró ser eficiente en su uso a través de los navegadores, rompiendo el paradigma *web* de la época de que todo cliente debía siempre comunicarse con un servidor.

[17] *Peer5*. http://peer5.com/. Junio de 2014.
[18] *Ericsson*. http://www.ericsson.com/. Noviembre de 2013.

Peer5 siguió incursionando en el área y optó posteriormente por volcarse a la tecnología de *data channels* de WebRTC. Si bien WebRTC es popularmente conocida por su API principal getUserMedia que permite enviar y recibir video y audio en forma P2P, también provee de la API RTCDataChannel que permite el intercambio de datos binarios en forma eficiente.

Esta API permite que los datos enviados sean además codificados a preferencia del desarrollador. Consecuentemente, se torna muy adecuada para el envío de contenido y Peer5 la utiliza en su producto más conocido ShareFest[19] para enviar archivos planos, imágenes, audio, video y datos de aplicación en tiempo real.

Peer5 utiliza un protocolo privado de señalización con requerimientos diferentes a los de la mayoría de las aplicaciones WebRTC según Weiss. Esto parecería un retroceso según los borradores de WebRTC, sin embargo para el grupo de Peer5 se torna provechoso según sus propios intereses.

Sus productos utilizan el lenguaje JavaScript tanto del lado del cliente como del lado del servidor. Para sus servidores utilizan Node.js como plataforma y *web sockets* como canal de comunicación con los clientes. La arquitectura desarrollada por Peer5 se puede apreciar en la Figura 16.

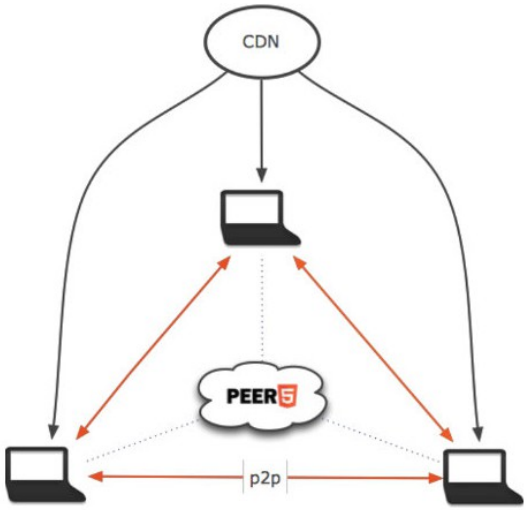

Figura 16. Arquitectura Básica de Peer5 [8]

Cuando un cliente decide, por ejemplo, cargar una página que tiene un video en su contenido, se reconoce al mismo y se le agrega automáticamente a un *swarm*. Un *swarm* consiste en una red de *peers* optimizada que permite el intercambio de partes del video entre sus participantes en forma directa. Según ingeniería reversa, la solución propuesta por Peer5 parece constar básicamente de cuatro componentes: los clientes o *peers*, los *trackers*, los CDNs y los servidores de señalización.

Los *trackers* consisten en los servidores Node.js, cuya función principal consiste en adherir *peers* a los *swarms* según varios factores tales como su localización, ancho de banda y congestión de red, entre otros. La comunicación con los clientes se realiza a través de *web sockets*.

[19] *ShareFest.Me*. https://www.sharefest.me/. Mayo de 2013.

Los clientes consisten en un código JavaScript que son descargados de un servidor de aplicación. Éstos envían y reciben partes de contenido de un CDN y en forma P2P a través de otros *peers*. En caso de que el cliente quiera descargar un video y reproducirlo al mismo tiempo, las partes de video que son recibidas se agregan al *buffer* de video en forma dinámica.

Todas las conexiones P2P son controladas por el propio cliente, lo que permite que el manejo de recursos no sea abusivo. El cliente también cuenta con un componente que monitorea y detecta constantemente las partes del video que son necesarias para continuar reproduciendo. Para esto se considera el tiempo de reproducción, la tasa de *bits* del video y el ancho de banda disponible del cliente.

En el *Capítulo 3 Parte Central del Trabajo* se verá una arquitectura similar a la propuesta por Peer5, pero fiel a los principios de funcionamiento de BitTorrent y GoalBit. Por ello, fue de sumo interés el estudio del código abierto de la aplicación ShareFest de Peer5, del cual se tomaron varias ideas que resultaron útiles al momento de optar por determinadas tecnologías para utilizar en el prototipo.

2.4.2 PeerCDN

PeerCDN[20] es una *startup* dedicada a crear tecnologías que aceleren el funcionamiento de aplicaciones *web* a costos menores de ancho de banda según se detalla en su página *web*. Fue fundada por tres estudiantes ingeniería de la Universidad de Stanford fanáticos del lenguaje JavaScript y de las tecnologías *web* tales como Node.js y *web sockets*.

El producto PeerCDN permite a las aplicaciones *web* servir su contenido estático (imágenes, video, etc) sobre una red P2P creada a partir de clientes que visitan la aplicación, utilizando por detrás la tecnología WebRTC. Más específicamente, utiliza los *data channels* provistos por la tecnología para transferir archivos entre los navegadores en forma directa.

El código de la aplicación es propietario y no existe mucha documentación acerca del mismo por tratarse de un producto muy reciente. Consecuentemente, se imposibilitó investigar su arquitectura. Debido a que únicamente los navegadores Chrome y Firefox implementan los *data channels* y *web sockets* (Firefox en forma parcial), se puede deducir que PeerCDN utiliza mecanismos de *fallback* para que los clientes que visiten la aplicación con otro navegador puedan acceder a un CDN para obtener el contenido estático.

Cabe destacar que la *startup* fue recientemente adquirida por Yahoo!, cuando PeerCDN aún se encontraba en su versión privada *beta*.

[20] *PeerCDN*. https://peercdn.com/. Mayo de 2014.

3 Parte Central del Trabajo

En el *Capítulo 2 Estado Del Arte* se detallan varias tecnologías que proponen una solución al problema del *streaming multimedia* P2P en la *web*. La gran mayoría de las arquitecturas proponen la instalación y uso de librerías externas para su funcionamiento. A su vez, algunas de ellas plantean al cliente como una unidad tonta, distribuyendo gran parte de la lógica de procesamiento del *streaming* del lado del servidor.

En este capítulo se pretende explicar los aportes del prototipo creado, como también estudiar sus requerimientos y decisiones tomadas en función de los mismos.

3.1 Aportes de la solución planteada

Habiendo estudiado el mercado disponible de tecnologías, se puede notar que hasta hace poco tiempo, el *streaming* de datos *multimedia* a través de la *web* requería del uso de servidores inteligentes y CDNs para la distribución de contenido, y de la instalación de librerías externas y *plugins* del lado del cliente para invocarlos.

Skype[21] es un claro ejemplo de un servicio propietario que requiere que un cliente instale una aplicación de escritorio, la cual se comunica con servidores que se encargan de mantener activa una comunicación *multimedia* entre clientes.

El caso más reciente de Netflix[22], que es un servicio para ver series y películas *online*, propone una arquitectura que permite a los clientes comunicarse con un servidor *multimedia* para traer los datos en varias calidades de video y audio según el ancho de banda disponible. Netflix permite visualizar las películas a través del navegador o a través de una aplicación móvil. Su limitación consiste en tratarse de una arquitectura cliente/servidor que no aprovechas las ventajas de un sistema P2P.

Con el surgimiento de HTML5 y tecnologías como WebRTC, se encuentra una solución al problema del uso de librerías externas, como también acabar con el paradigma de cliente/servidor para obtener datos *multimedia*.

Por estas razones, se plantea como objetivo la creación de un prototipo de arquitectura de *streaming* P2P que cumpla con los puntos anteriores: encontrarse conformada por clientes más inteligentes, liberando al servidor de gran parte de lógica de procesamiento.

Se optó por encarar un problema sencillo que consiste en la reproducción de un video con formato de contenedor .webm alojado en varios CDNs desde el navegador. El prototipo implementado permite también extender la arquitectura para soportar de forma sencilla la reproducción de elementos de audio. No brinda soporte para la reproducción de *streaming* en tiempo real, pero se consideró este punto al momento del diseño, a modo de permitir extender el prototipo para su soporte.

Se encontró muy interesante el enfoque propuesto por WebRTC. Esta tecnología aún se encuentra en proceso de borrador por parte de la IETF, pero es muy atractiva en su planteo acerca de poder aprovechar las potencialidades provistas por HTML5 que permiten la

[21] *Skype*. http://www.skype.com/en/. Mayo de 2013.
[22] *Netflix*. https://www.netflix.com/Netflix. Junio de 2013.

comunicación P2P entre navegadores en forma directa, y solucionando los impedimentos de comunicación tales como *firewalls* o NAT Traversal.

El caso de PeerCDN es el más similar al prototipo creado (si bien no comparte video en vivo), ver la Sección *2.4.2 PeerCDN*. Éste permite a un cliente descargar el contenido estático de un sitio *web* (imágenes, videos, etc) a través de P2P con otros clientes que se encuentren visitando el mismo sitio en el mismo momento. Utiliza WebRTC para realizar la comunicación P2P y es uno de las primeros casos que utiliza esta tecnología para atacar el problema P2P en la *web*.

El hecho de que PeerCDN haya logrado probar que se puede lograr un nivel de P2P eficiente entre navegadores, motivó aún más al desarrollo del prototipo y a la investigación de tecnologías que sirvan para desarrollarlo.

Recientemente, Netflix demostró encontrarse interesado en cambiar su arquitectura cliente/servidor a una arquitectura P2P de *streaming* asistido[23] para resolver conflictos existentes entre *peers* y sus ISPs. Se espera que el prototipo pueda resultar de utilidad para el estudio de la factibilidad de un servicio de *streaming* de video P2P entre navegadores mediante el simple uso de HTML5 y WebRTC, tal como peerCDN ha demostrado ser un avance en el área P2P de la *web*.

3.2 Requerimientos del prototipo

A continuación se plantea una lista de requerimientos que debe cumplir el prototipo desarrollado:

1. La aplicación debe basar su comportamiento en el prototipo GoalBit (similar a BitTorrent para *streaming*). Esto es, contar con una arquitectura P2P que mantenga las mismas entidades de BitTorrent: *peers* representados por navegadores *web*, *swarms*, *trackers* y servidores de contenido de datos.
   A su vez, basarse en los algoritmos de selección de *chunks* y *peers* implementados por el mismo. Por otro lado, se deben utilizar conceptos del protocolo GoalBit en lo que refiere a restricciones temporales y de compartimiento de los datos por tratarse de datos *multimedia* (se deben utilizar ventanas de tiempo para controlar este aspecto).
2. El prototipo debe utilizar las tecnologías WebRTC y HTML5 para su implementación, las cuales son nativas del navegador. Por lo tanto, no debe utilizarse ningún tipo de *plugin* o librería externa que deba instalarse sobre el navegador para el funcionamiento de la aplicación.
3. Los *peers* deben ser entidades inteligentes que puedan procesar la mayor cantidad de lógica posible sin afectar negativamente el rendimiento. Esto permite que los servidores no procesen mucha lógica y puedan atender más pedidos y de forma más veloz.
4. Debe ahorrarse ancho de banda, compartir los datos *multimedia* por P2P en un porcentaje global cercano al 50% y reducir la latencia lo más posible.

[23] *Netflix considers P2P-powered streaming technology.* https://torrentfreak.com. Abril de 2014.

3.3 Decisiones tomadas para el prototipo

En base a los requerimientos presentados en la sección anterior se creó la siguiente lista de decisiones cumple el prototipo para asemejarse lo más posible a un escenario real (ver Figura 17).

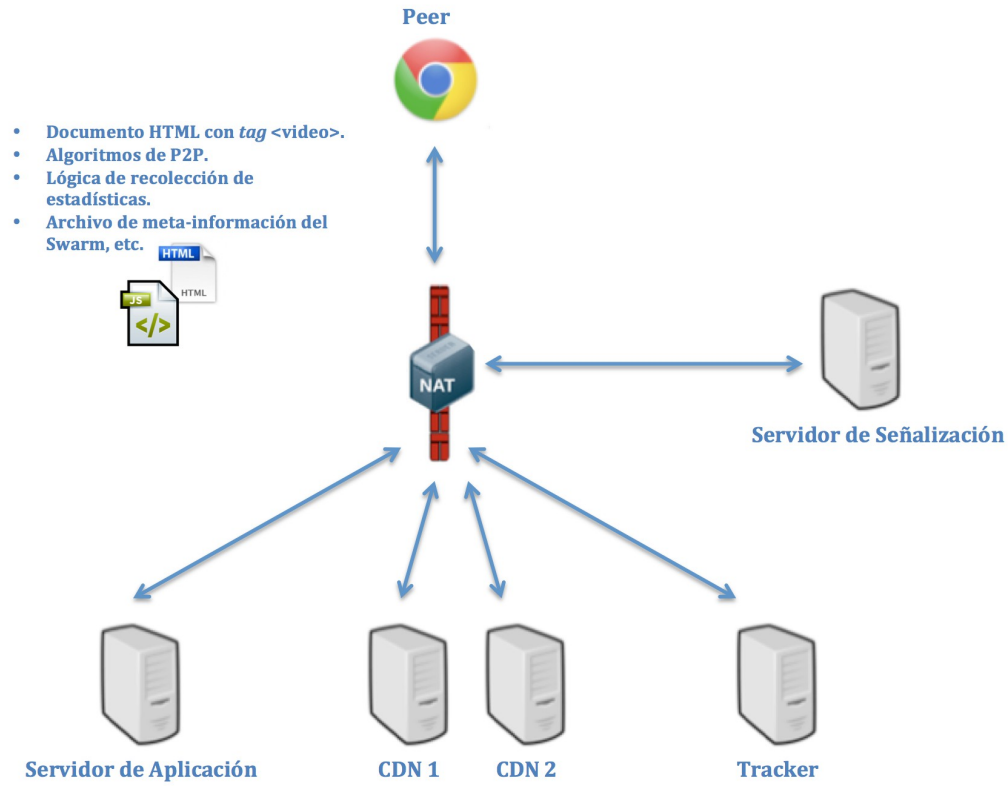

Figura 17. Componentes del prototipo.

1. Se cuenta con un servidor de aplicación del cual cada cliente descarga el código JavaScript a correr en el navegador y que implementa toda la lógica de un *peer*.
2. Se cuenta con dos servidores de contenido (CDN) que almacenan el video a reproducir.
3. La cantidad de *peers* (clientes) del sistema no tiene número fijo, varía dinámicamente.
4. Los *peers* contienen guardado en su lógica un archivo de meta-datos que cumple misma función que un archivo *torrent* de BitTorrent. Esto es, el archivo contiene información de los *trackers* y meta-datos del video a consumir.
   Además, cuentan con este archivo en su lógica desde que son inicializados, sin tener que pedírselo al *tracker* ni pedir su actualización posteriormente. El cliente obtiene el archivo junto con la lógica de cliente desde el servidor de aplicación. Esto surge como una simplificación de la realidad.
5. Los *peers* contienen la mayor cantidad de lógica de procesamiento que puedan, de tal forma de que se comporten como entidades lo más inteligente posible, liberando al *tracker* de tareas. Por lo tanto, los clientes implementan lógica de procesamiento del video a reproducir, algoritmos P2P de selección de piezas y peers, procesamiento de datos de *performance* tales como consumo desde los CDNs y de P2P, entre otros.

6. Se cuenta con un solo *tracker* en el sistema que mantiene lógica de gestión de un único *swarm*. Este servidor se comunica con los *peers* en forma periódica como colaborador para el funcionamiento de los algoritmos P2P, y como receptor de datos de *performance* del sistema.
7. E l *tracker* brinda también un servicio de estadísticas del sistema que puede accederse a través de la *web*.
8. Se cuenta con un único servidor de señalización requerido para establecer las comunicaciones por *data channels* entre *peers*.

4 Implementación

En este capítulo se explica más detalladamente el diseño y arquitectura del prototipo que fue implementado y que fue parcialmente presentado en el *Capítulo 3 Parte Central del Trabajo*. También se detallan y fundamentan las tecnologías y librerías utilizadas.

4.1 Introducción

La solución propuesta consiste en un sistema de *streaming multimedia* P2P entre navegadores *web*. Cuenta con los conceptos básicos de un sistema P2P tales como *peers*, *swarms*, *trackers* y servidores de contenido. A su vez, cuenta con un servidor de aplicación que provee la lógica de un cliente.

La idea consiste en brindar un servicio por el cual un usuario accede a una página *web*, y a través de la misma reproduce un video que se compone por *chunks* traídos o bien de un servidor de contenido, o bien por P2P a partir de otros usuarios que acceden al mismo sitio *web*.

Por otro lado, el *tracker* del sistema almacena estadísticas que permiten monitorear el estado del *swarm*, medidas de ahorro y de *performance* del mismo. Esto permite analizar el comportamiento del sistema y obtener conclusiones.

Para el desarrollo del prototipo se utilizaron las tecnologías de punta provistas por HTML5, tales como los *data channels* de WebRTC, *web sockets*, la API Media Source, los lenguajes HTML y JavaScript y algunas librerías de código abierto que se detallan en la Sección *4.6 Tecnologías utilizadas*.

La factibilidad de un sistema como el propuesto permitiría lograr un mejor aprovechamiento de los recursos al momento de diseñar un sitio *web*. Por ejemplo, una página o aplicación *web* de un periódico u otro medio se vería altamente beneficiada, ya que sus recursos estáticos tales como imágenes, audio y video podrían compartirse por P2P entre los clientes que se conecten al sitio, reduciendo costos de CDN y de ancho de banda. Como se explicó en el *Capítulo 3 Parte Central del Trabajo,* tal fue el enfoque encarado por PeerCDN, empresa que demostró la viabilidad de un servicio similar que fue desarrollado con tecnologías similares.

4.2 Diseño y arquitectura

El diseño de un prototipo que se asemeje a un sistema de *streaming* real fue una actividad que requirió de un tiempo largo de investigación y estudio de las tecnologías disponibles, como también del entendimiento y reproducción de los algoritmos P2P que debieron codificarse.

El diseño y arquitectura de la solución fue planteándose en forma gradual. En primer lugar, se debió evaluar la factibilidad y consecuentes restricciones del uso de las herramientas propuestas por HTML5 para lograr una comunicación por *data channels* entre navegadores, como también el establecimiento y comunicación por *web socket* entre un navegador y un servidor. Acorde a los requerimientos y decisiones tomadas a partir de ellos vistos en las Secciones *3.2 Requerimientos del prototipo* y *3.3 Decisiones tomadas para el prototipo* respectivamente, el prototipo cuenta con las siguientes entidades:

1. Número dinámico de *peers* (más específicamente, navegadores Chrome de Google).
2. Un servidor de aplicación que provee a los clientes del código JavaScript de un *peer*.
3. Un *tracker* que mantiene la lógica de un *swarm* y almacena estadísticas del sistema.
4. Dos servidores de contenido (CDNs) que almacenan el video a compartir.
5. Un servidor de señalización que permite establecer las comunicaciones por *data channels* entre *peers*.
6. Un servidor STUN para realizar el NAT traversal.
7. Un servidor TURN que realiza el *relay* de los mensajes en caso de que el Hole Punching no sea exitoso.

En la Figura 18 se aprecia un diagrama global de la arquitectura de la solución, en la cual participan todas las entidades anteriormente mencionadas.

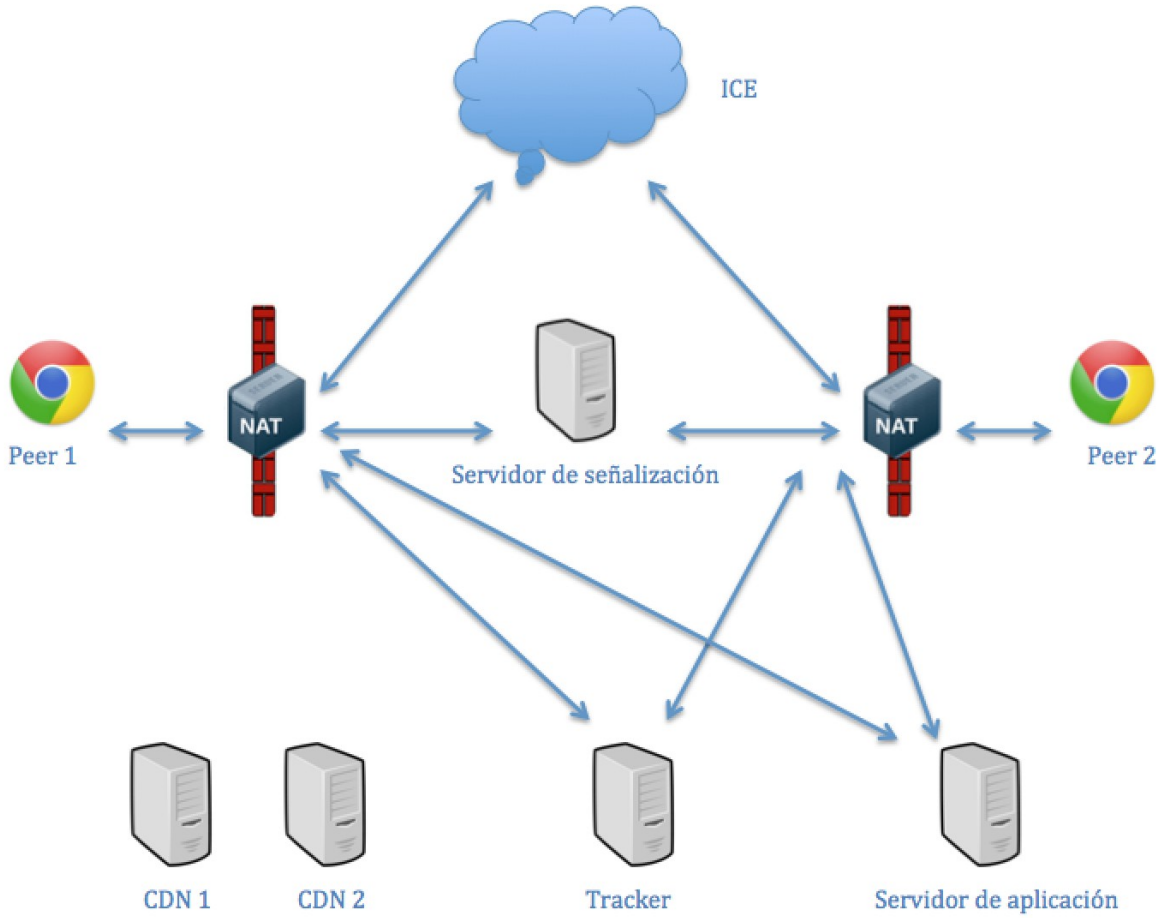

Figura 18. Arquitectura global de la solución propuesta.

4.2.1 Servidor de Aplicación

El servidor de aplicación es el encargado de alojar el sitio *web* principal al que accederá cada *peer* para visualizar un video. Consta de un servidor NodeJS que escucha por peticiones HTTP. El mismo está instalado en Heroku[24] y es accesible a través de un navegador, a través de la URL http://stream-io.herokuapp.com.

[24] *Heroku – Cloud Application Platform.* https://www.heroku.com/. Diciembre de 2013.

El servidor devolverá en una página HTML el video seleccionado para el prototipo y además el código JavaScript que implementa la lógica del cliente, que se encargará de orquestar las interacciones entre el servidor de señalización, los *CDNs*, el *tracker* y el resto de los *peers*.

Dicha lógica es la responsable de implementar los algoritmos utilizados para descargar, almacenar y reproducir los *chunks* del video, que pueden ser obtenidos desde uno de los dos *CDNs*, o directamente desde otro *peer*. La decisión de optar por dónde descargar un *chunk* también forma parte de las responsabilidades del cliente.

Para minimizar la carga en los *CDNs* y maximizar el ahorro, sin perder calidad en la continuidad de la reproducción, es necesario definir un conjunto de políticas que estos algoritmos deben satisfacer. Los detalles de implementación y definición de estas políticas se explican en la Sección *4.5 Estrategias del protocolo*.

4.2.2 *Tracker*

El *tracker* cumple básicamente dos responsabilidades:

- Mantener la lógica del *swarm.*
- *C*alcular y servir las estadísticas de ahorro y calidad del sistema.

Corre en un servidor NodeJS que escucha peticiones HTTP y también posibles conexiones por *web socket*. Se encuentra instalado en Heroku y provee una interfaz de estadísticas accesible a través de un navegador utilizando la URL http://stream-io-tracker.herokuapp.com/stats.

La conexión por *web socket* se expone para la interacción con el cliente JavaScript retornado por el Servidor de Aplicación como canal para el envío de los datos de señalización requeridos por los algoritmos P2P. El *tracker* solamente interactúa con el cliente JavaScript, no recibe ni envía información a ningún otro componente de la arquitectura.

Que sea el encargado de mantener el *swarm* implica que debe dar de alta (y baja) los *peers* que ingresan (o salen) del sistema. Al mismo tiempo, debe utilizar esa información para satisfacer las peticiones de los *peers,* que cada tanto solicitan saber con qué subconjunto de otros *peers* podrían eventualmente hacer P2P. Parte del trabajo del *tracker* es devolver "la mejor" lista posible, de manera de favorecer el P2P lo más posible.

En lo que respecta a las estadísticas, es el encargado de asimilar la información enviada por los *peers* para generar métricas de ahorro y de calidad, como pueden ser: qué porcentaje de la descarga se realizó a través de un *CDN*, qué tanto se descargó a través de P2P, cuánto tiempo demoró el video en empezar reproducir, etc.

4.2.3 CDNs

Se cuenta con dos servidores de contenido, denominados CDN1 y CDN2 que tienen la responsabilidad de servir los archivos de video. Para el CDN1 se utilizó el servicio SoftLayer[25] y para el CDN2 se utilizó GoDaddy[26]. El video puede ser accedido también de manera íntegra desde http://trial.goalbit-solutions.com/goalbit-

[25] *SoftLayer*. http://www.softlayer.com/. Agosto de 2014.
[26] *GoDaddy*. http://www.godaddy.com/. Agosto de 2014

misc/videos/remuxed_2sec.webm (CDN1) o desde http://goalbit-solutions.com/www2011/images/videos/remuxed_2sec.webm (CDN2).

Los CDNs son componentes esenciales en el sistema, porque los algoritmos asumen que son entidades confiables y eficientes. Las peticiones a los CDNs deben ser performantes para que el sistema funcione.

En la arquitectura planteada, el sevidor de Softlayer representa al CDN1, como más potente y costoso. Mientras tanto el servidor de GoDaddy toma el rol del CDN2, siendo menos potente y más barato.

4.2.4 Servidor de Señalización y servidores ICE

El Servidor de Señalización tiene la responsabilidad de ser el canal de comunicación durante la conexión inicial de dos o más *peers* cuando necesitan negociar la Peer Connection. El resultado de una negociación exitosa será la posibilidad de comunicarse directamente a través de Internet, sin intermediarios, utilizando sus direcciones IPs públicas.

Se utilizó un servidor NodeJS que corre una implementación realizada por el equipo de *PeerJS* (ver la Sección *4.6.6 Peerjs*). La misma escucha conexiones por *web socket* y se encuentra alojada en OpenShift[27] bajo el dominio http://peerjs2-streamio.rhcloud.com.

Cuando dos *peers* tienen intención de realizar P2P, deben primero generar una Peer Connection exitosa. En el proceso, los *peers* deberán primero interactuar con el Servidor de Señalización, quien actuará de *pivot* en la negociación. Dicha negociación constará del intercambio de una serie de paquetes SDP (*Offers* y *Answers*) para acordar las condiciones iniciales de la sesión.

Es en ese momento, como parte del acuerdo de la sesión, en el que se realiza el *NAT Traversal*. Se utiliza el servidor STUN para descubrir la dirección IP pública local, que luego es enviada al *peer* remoto a través del Servidor de Señalización en un paquete SDP (ver el *Anexo 9.3.2 Session Description Protocol*). Una vez finalizado el intercambio, los *peers* deberán ser capaces de enviarse mensajes directamente. Por otra parte, si el *NAT Traversal* no resulta exitoso se utiliza un servidor TURN que hará el *relay* de los mensajes de los *peers*.

Se decidió utilizar servicios públicos tanto para el servicio STUN como para el servicio TURN. El servidor STUN es un servicio de Google gratuito, alojado en stun.l.google.com:19302, mientras que el servidor TURN es un servicio de Viagenie[28], accesible a través del dominio numb.viagenie.ca:3478.

4.2.4.1 Uso Del Sistema

Como puede verse en la Figura 18, los clientes que desean acceder al sitio *web* para reproducir el video proceden de la siguiente manera:

1. El usuario abre el navegador Google Chrome y accede al sitio *web* http://stream-io.herokuapp.com para descargar el código JavaScript del *peer* provisto por el servidor de aplicación.

[27] *OpenShift by Redhat*. https://www.openshift.com/. Agosto de 2014.
[28] *Viagénie.* http://www.viagenie.ca/. Agosto de 2014.

2. Junto con el código del *peer*, se descarga el archivo *torrent* que contiene los datos del *tracker* del sistema y los meta-datos del video a reproducir. El cliente lee este archivo e inicia una comunicación por *web socket* con el *tracker*.
3. El *tracker* agrega el nuevo *peer* al *swarm* y le envía meta-información de los otros *peers* conectados al *swarm* y de los CDNs.
4. El *peer* obtiene los datos del *swarm*, y a través del servidor de señalización se utilizan los protocolos SDP y ICE para establecer una comunicación en forma directa con los otros *peers* a través de *data channels*, resolviendo el problema del NAT traversal y *firewalls* (ver la Sección *2.3.5.3.2 Conexión entre dos peers*).
5. Paralelamente, el *peer* comienza a correr los algoritmos P2P e inicializa las ventanas de tiempo para obtener *chunks* del video a reproducir, pidiendo los primeros a los CDNs que almacenan el video.
6. Una vez establecida la comunicación por *data channel* con los otros *peers*, el cliente puede compartir *chunks* con los mismos en la medida que los protocolos de intercambio lo permitan (ver Secciones *4.4.4.1 Intercambio de información contextual* y *4.4.2.2 Mensaje Swarm-Request*).
7. El video comienza a reproducirse en el navegador tan pronto como sea posible.

Cabe destacar que, el procedimiento ICE es realizado en forma transparente al prototipo ya que el *framework* WebRTC lo provee de manera nativa. Únicamente se debieron configurar los servidores STUN y TURN con los cuales debe comunicarse el *peer* para resolver el NAT Traversal.

4.3 Restricciones del prototipo

Debido al uso de tecnologías tan incipientes como los *data channels* de WebRTC, la API Media Source de HTML5 y los *web sockets*, fueron surgiendo varios obstáculos y restricciones durante el desarrollo de la solución. Algunas fueron fáciles de solucionar, otras requirieron de un cambio en el diseño de la aplicación.

Comenzando por los *data channels* de WebRTC, un primer obstáculo que surgió consiste en que su implementación se encuentra desarrollada en forma completa únicamente en el navegador Chrome, y parcialmente en Firefox. Por tal razón, el prototipo debió ser desarrollado para que funcionara en el navegador Chrome de Google, siendo fácilmente extensible agregar el soporte para Firefox.

A su vez, debieron realizarse varias pruebas para verificar que el desempeño de un *peer* no se viera afectado notoriamente por la cantidad de conexiones por *data channels* que tuviera establecidas. Este experimento no dio consecuencias significativas, aunque si demostró consumir mucho ancho de banda.

Por otro lado, la implementación actual de los *data channels* de Chrome soporta el intercambio de datos de tipo String de JavaScript y binarios. Consecuentemente, se tuvo que tener especial cuidado al momento de codificar y decodificar los datos enviados por este medio. La librería peerJS utilizada (ver la Sección *4.6.6 Peerjs*) se encarga de codificar y decodificar los mensajes enviados en formato UTF, en función del tamaño del mensaje.

El uso de la API Media Source permite construir en forma dinámica flujos de datos *multimedia* para los *tags* <video> y <audio> de HTML5, tal como fue explicado en la Sección *2.3.4 Media Source*. Consecuentemente, esta API permitió aprovechar el *tag*

<video> y todas sus funcionalidades para reproducir el video sin la necesidad de instalar ningún *plugin* sobre el navegador. Afortunadamente, el único navegador que implementa la API Media Source y también todas funcionalidades del *tag* <video> en su totalidad es Chrome. Por lo tanto, el uso de esta API generó la restricción de utilizar el mismo navegador que fue restringido por el uso de los *data channels*.

Como se puede concluir de lo anterior, la solución propuesta debió ser implementada para el navegador Chrome de Google. Esto consiste en una restricción bastante fuerte, dado que actualmente Chrome es utilizado por un 34.1% de la población mundial[29]. Igualmente, no resulta complicado su extensión y compatibilidad para otros navegadores como Firefox y Safari, los cuales tienen en sus planes futuros una implementación robusta de las mismas tecnologías.

Por otro lado, el uso del navegador Chrome tiene una fuerte restricción sobre el formato de contenedor de video soportado nativamente para el *tag* <video>. Chrome soporta únicamente el formato WebM de video, el cual es abierto y fue específicamente diseñado para su reproducción en un <video>. Esto se debe a que Google es el *sponsor* de WebM, queriendo instaurar un estándar de formato de video en la *web*.

El formato WebM se especifica en el *Anexo 1 Formatos de contenedor y codecs* y en el *Anexo 8 WebM Byte Streams*. Básicamente un archivo WebM se compone de flujos de video comprimidos mediante el *códec* V8 de video y el *códec* Vorbis de audio. Su especificación es técnicamente similar al contenedor Matrosk, en cual se basó gran parte del formato del contenedor.

A diferencia de HLS, WebM no cuenta con archivos .ts bien definidos, sino que su estructura se compone de segmentos. La estructura de un archivo WebM se compone por un segmento de inicialización y por varios segmentos de datos *multimedia* o *clusters*. Cada uno de estos segmentos contiene un cabezal con meta-información del contenido del segmento.

Esto fue una limitante al momento de implementar la forma de intentar reproducir una lógica similar a la de HLS. El archivo de video que se encuentra en los servidores de contenido debe ser leído por rango a pedido del *peer*, en lugar de poder contar con archivos de formato .ts que puedan descargarse más rápidamente.

Finalmente, la última restricción enfrentada se encuentra ligada al lenguaje JavaScript que se utilizó para la implementación de la solución. Dado que este lenguaje es implementado en los navegadores como una sola hebra de ejecución, no existe paralelismo de tareas.

Esto fue un factor determinante al momento de decidir qué lógica mover del servidor al cliente de forma de hacerlo lo más inteligente posible. La sobrecarga de lógica en un *peer* puede ocurrir al momento de contar con varias conexiones por *data channel* con otros *peers*, comunicación por *web socket* con el *tracker*, reproducción de video y llamados AJAX a los CDNs, como también con la toma de estadísticas que se toman periódicamente.

[29] *Estadísticas globales del uso de navegadores*. http://www.w3counter.com/, Junio de 2014.

4.4 Protocolo de Transporte

4.4.1 Descripción del flujo de un *peer* de alto nivel

Varios protocolos y múltiples tecnologías fueron investigadas para implementar el prototipo. Se han adoptado comportamientos de varios de ellos y esta sección explica los algoritmos y las decisiones tomadas en torno a los mismos.

Para desarrollarlos de manera que se entienda bien el porqué de la elección de los mismos, se describen paso a paso todos los estados por los que va pasando un *peer* a lo largo de su vida. De esta manera se podrá comprender sin problemas cómo la combinación de las partes lleva al funcionamiento del todo.

Inicialmente, lo primero que trata de hacer un *peer* cuando carga, es hacer contacto con el *tracker*. Éste le dará información sobre el video a descargar, los CDNs a los que debe contactar para descargar el video y un identificador para que los demás *peers* lo reconozcan como único.

Una vez que cuente con información sobre el video, se podrá comenzar a cargar; y para ello, antes se deben inicializar los algoritmos de petición de *chunks*. Se decidió apegarse bastante al protocolo de GoalBit (ver el *Anexo 4 GoalBit*) para las peticiones P2P.

Se establecen tres ventanas de ejecución a lo largo de la reproducción del video. Estas están definidas por la ventana de tiempo existente entre el tiempo de reproducción del video y el tiempo máximo de video almacenado en el *buffer*. La cantidad de tiempo en esta ventana, define entonces la urgencia con la cuál la aplicación debe pedir nuevos *chunks*.

Segundos de *buffereo* extra (x)	**Ventana**
X < 30s	Urgente
30s < x < 50s	Normal
50s < x	Futura

Tabla 4: Tipos de ventana de reproducción.

Un *peer* siempre comienza en la ventana urgente, porque al principio su *buffer* está vacío. El comportamiento en ese momento consta en pedirle al mejor CDN para tratar que la reproducción no se frene.

Si se consigue *bufferear* más allá del punto por el cuál se va reproduciendo el video, eventualmente se pasa a la ventana normal. Ésta tiene varios objetivos:

- Obtener partes del video que sean interesantes para intercambiar con otros *peers*.
- Llegar a un estado en el cuál se pueda depender únicamente del P2P y a la vez tratar de reducir los costos lo más posible, sin deteriorar la continuidad en la reproducción.

A medida que el video continúe *buffereando*, se llegará a un estado de comodidad suficiente para ejecutar un comportamiento más arriesgado y comenzar a prescindir de los CDNs. Se entrará en la ventana futura y el *peer* sólo podrá conseguir nuevas partes del video, si los *peers* con los que habla se los pueden proporcionar.

Es importante en esta etapa explicar a fondo las comunicaciones y algoritmos utilizados para lograr una comunicación P2P exitosa. Las siguientes secciones detallan estos puntos.

4.4.2 Comunicación *Peer - Tracker*

En el proceso de inicialización de un *peer*, éste se anuncia al *tracker*, quien luego de generar una instancia asociada al mismo, le devuelve su *id*, y un conjunto de CDNs. Para cada uno de ellos se retorna nombre, URL y costo, siendo este último un entero positivo menor o igual a diez que representa el costo económico asociado al uso del recurso y su desempeño. Un número cercano a cero significa que es caro y asegura un buen rendimiento, mientras que cercano a diez refiere a un bajo costo y menos confiabilidad.

Luego de la inicialización, cada *peer* mantiene tres tipos de comunicaciones periódicas, las cuales se pasarán a describir a continuación. Cabe destacar que todas las comunicaciones entre *peers* y el *tracker* se mantienen a través de *web sockets* (ver el *Anexo 7 Web Sockets*).

4.4.2.1 Mensaje *Keep-Alive*

Para que el *tracker* mantenga una visibilidad actualizada del estado de la red, debe dar por muerto a aquellos *peers* con los que no mantenga ningún tipo de comunicación por un tiempo determinado. En la implementación se decidió utilizar para ello nueve segundos. Se opta por un mecanismo de este tipo, en contraposición a que los *peers* notifiquen su retiro de la red de manera explícita, por la simple razón de que si un *peer* se retira de manera abrupta, no logrará tal notificación.

En otras palabras, cada *peer* es responsable de mantener comunicaciones periódicas con el fin de que el *tracker* no lo dé de baja. En la implementación del prototipo, los mensajes *keep-alive* se envían cada tres segundos y, además de cumplir con el objetivo antes descrito, también son usados para llevar al *tracker* información de desempeño del *peer*, tales como cantidad de *chunks* obtenidos por medio de P2P, cantidad de *chunks* obtenidos de cada CDN. También se usa el mensaje de *keep-alive* para enviar al *tracker* el tiempo que tomó el inicio de la reproducción, la cantidad de veces que entra en *buffering* (se detiene la reproducción de video) y el tiempo total que la reproducción del video estuvo detenida.

4.4.2.2 Mensaje *Swarm-Request*

Las comunicaciones entre dos *peers* pueden eventualmente terminarse por distintos motivos, como puede ser la falta de interés mutuo, o simplemente el abandono de la red por parte de alguno de los participantes. Por tal motivo y el carácter dinámico de la red donde entran y salen *peers* continuamente, cada *peer* puede necesitar periódicamente solicitar al *tracker* un nuevo conjunto de *peers* para añadir a su *swarm* ya existente.

Esto se hace por medio de un mensaje *swarm-request* en el cual un *peer* envía su *id* y el número de *chunk* por el cual va reproduciendo. A partir de esta información el *tracker* le responde con un conjunto de *peers* con los cuales el *peer* solicitante intentará establecer conexiones en caso que sean nuevos *peers* que ingresan a su *swarm*.

4.4.2.3 Mensaje *File-Completed*

Cada *peer* al terminar de cargar todo el archivo de video, se lo comunica al *tracker* por medio de un mensaje llamado *file-completed*, que incluye como información únicamente el *id* del *peer* en cuestión. El *tracker*, en base a este mensaje, cambia el estado del *peer*

identificado a *seeder*, pasando éste solamente a distribuir contenido pero sin solicitar *chunks* a sus pares.

4.4.3 Comunicación *Peer* - CDN

La única comunicación que mantienen los *peers* con los CDNs se da para la solicitud de *chunks*. Para el caso de la implementación presentada, esto se realiza por medio de pedidos AJAX (ver el *Anexo 6.3 Lenguaje JavaScript*) que hacen uso del *header Range*[30] de HTTP. Esto le permite al *peer*, sabiendo el comienzo y el fin de cada *chunk*, solicitarle el mismo al CDN, quien distribuye estáticamente los contenidos desde el servidor *web* sin tener que hacer un mayor procesamiento a nivel del servidor de aplicación. La utilización de HTTP Range habilita el uso de la mayoría de los CDNs disponibles en el mercado.

4.4.4 Comunicación P2P

Como ya se mencionó en la Sección *4.4.2.2 Mensaje Swarm-Request*, cada *peer* recibe del *tracker* información de otros *peers* con los que se dispone a intercambiar *chunks*. Tal como se expresa en la Sección *3.3 Decisiones tomadas para el prototipo* cualquier comunicación entre *peers* de acuerdo con la implementación realizada se da por medio de *data channels*.

Esta comunicación no implica solamente los mensajes de intercambio de *chunks*, sino que también se utilizan mensajes para compartir información contextual como ser, que un *peer* tiene un nuevo *chunk*, notificar que sigue activo, etc.

Cabe destacar que, el conjunto de mensajes utilizados tanto para el intercambio de información contextual como para el intercambio de *chunks*, se basa en los utilizados por GoalBit en su implementación. Para las necesidades de esta implementación se realizó una adaptación de varios de ellos. Esto responde a varias causas, como pueden ser simplificaciones prácticas, mejoras de desempeño del prototipo (uniendo información de varios mensajes en uno para economizar la comunicación) o simplemente diferencias en los requerimientos del prototipo realizado con respecto al problema que resuelve GoalBit.

En la Sección *4.4.4.1 Intercambio de información contextual* se describe con mayor detalle cada uno de las categorías y los tipos de mensaje que se intercambian.

4.4.4.1 Intercambio de información contextual

Cuando se habla de información contextual de un *peer*, se habla de información como por ejemplo su índice de reproducción (el *id* o número del *chunk* que se encuentra reproduciendo) o su *bitfield*. Este último es una representación en forma de mapa de *bits*, donde para el contenido en reproducción se indica cuáles *chunks* ha descargado el *peer* y cuáles no ha obtenido aún, tal cual se muestra en la Figura 19.

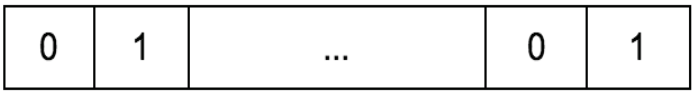

Figura 19: Arreglo cuyo tamaño es igual al número de chunks del contenido y donde 0 representa que no tiene el chunk correspondiente a la posición del arreglo y 1 que ya lo ha obtenido.

[30] *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. https://www.ietf.org/rfc/rfc2616.txt. Agosto de 2014.

En el proceso de comunicación entre dos *peers* se dan distintas etapas. En la etapa inicial, cuando un *peer* recibe información de otro en su *swarm* desde el *tracker*, comienza una etapa de establecimiento de la conexión donde se utilizan los mensajes HANDSHAKE_RESPONSE y HANDSHAKE_ESTABLISHED, tal cual se explica en la Sección *4.5.3 Proceso de Handshake*.

Después de establecida la conexión mediante el proceso de *handshake*, en el cual se intercambian sus respectivos *bitfields*, cada *peer* debe mantener actualizado el *bitfield* de los *peers* con los cuales se encuentra dialogando, a medida que estos van recibiendo nuevos *chunks*. Para ello se utiliza el mensaje HAVE, donde cada vez que un *peer* recibe un nuevo *chunk*, notifica de tal evento a todos aquellos *peers* con lo cuales tenga una conexión establecida, cuyo índice de reproducción sea menor al *chunk* que motiva el mensaje HAVE y si ese *peer* ya no posee el *chunk*. En otras palabras, un *peer* recibirá un HAVE para un *chunk*, en caso que aún le falte.

Finalmente, se debe explicar cómo se maneja el problema de la caída de *peers* dentro de la red P2P. Como ya se vió en la Sección *4.4.2 Comunicación Peer - Tracker*, cuando un *peer* se retira de la red, el *tracker* detecta su ausencia luego de diez segundos de no recibir ningún mensaje *keep-alive* hacia él. Consecuentemente, el *tracker* lo marca como inactivo y no lo retorna a otros *peers* a la hora de generar *swarms*.

Análogamente entonces, los *peers* deben ser capaces de detectar cuando un *peer* con el que tenían una conexión activa abandona dicha conexión (ya sea por cualquiera de los posibles motivos: desinterés en la contraparte, o la finalización voluntaria o involuntaria de la ejecución del programa). Por ende, de manera similar a la interacción entre los *peers* y el *tracker*, los *peers* se envían mensajes *keep-alive* cada diez segundos. En caso que un *peer* no reciba ningún mensaje de otro durante un lapso mayor a ese tiempo, este último es declarado inactivo o muerto y removido del conjunto de *peers* que mantiene el primero.

4.4.4.2 Intercambio de *chunks*

El intercambio de *chunks* en la implementación presentada es regulado por medio de varios mensajes entre los *peers* participantes. Para poder explicar los mensajes, primero se debe explicar la existencia de cuatro propiedades que un *peer* mantiene para con los *peers* con los que tiene una conexión establecida. Suponiendo que el *peer* A tiene dichas propiedades respecto del *peer* B, las cuales, en el contexto del *peer* A, se definen de la siguiente manera:

- ***Local Choked*:** determina si A está dispuesto a responder a pedidos de B.
- ***Remote Choked*:** determina si B está dispuesto a responder a pedidos de A.
- ***Local Interested*:** determina si A está interesado en al menos un *chunk* que posee B.
- ***Remote Interested*:** determina si B está interesado en al menos un *chunk* que posee A.

Para cambiar el estado de dichas propiedades, los *peers* intercambian los siguientes mensajes, cuyos cuerpos constan únicamente del tipo de mensaje:

- ***Choke*:** el receptor de este mensaje pasa a no estar habilitado a descargar *chunks* del emisor.

- ***Unchoke*:** el receptor de este mensaje pasa a estar habilitado a descargar *chunks* del emisor.
- ***Interested*:** el receptor es notificado que el emisor está interesado en él.
- ***Not-Interested*:** el receptor es notificado que el emisor dejó de estar interesado en él.

Para que se habilite el intercambio de *chunks* desde un peer A a un peer B, el *peer* A debe *unchokear* al *peer* B y el *peer* B debe haber mostrado interés en A.

Para explicar en base a los estados previamente listados, se puede analizar el escenario anterior, meramente desde el punto de vista del *peer* A. Que A haya *unchokeado* a B significa que deshabilitó *local choked* para B en A. Que B haya mostrado interés en A significa que se habilitó *remote interested* para B en A.

Tanto los mensajes descritos anteriormente, como la ejecución de determinados algoritmos como los vistos en la Sección *Error: Reference source not found*, son los responsables de que estos estados cambien.

Toda esta información permite a los *peers* controlar con quiénes ellos desean compartir sus *chunks* y saber a quiénes pueden pedírselos, utilizando el interés mutuo entre pares. Esto tiene el objetivo de mantener conexiones duraderas y efectivas, con un subconjunto de todos los *peers* de la red P2P.

4.5 Estrategias del protocolo

4.5.1 Selección de *chunks*

Las peticiones a los diferentes recursos y la obtención de piezas del video son cruciales en el sistema. Son las encargadas de conseguir tener un video para visualizar, lo cual es básicamente el caso de uso a satisfacer.

A continuación se explican una a uno los algoritmos para cada una de las tres ventanas de ejecución. Se pretende lograr una explicación a más bajo nivel de la que se dio anteriormente, incluyendo información sobre las estructuras y lógicas utilizadas.

Se cuenta con un módulo que se encarga exactamente de la ejecución de los distintos algoritmos dependiendo de las ventanas, llamado *broker*. JavaScript es un lenguaje bloqueante y por lo tanto no permite ejecutar bucles infinitos. Se decidió entonces utilizar algoritmos que se desencadenan a través de *timeouts* periódicos, dejando que el resto de la aplicación ejecute en los interines correspondientes. El tiempo de *timeout* es configurable y en base a las experiencia y los tiempos de respuesta de las peticiones, se decidió pedir *chunks* cada un segundo.

4.5.1.1 Ventana Urgente

Un *peer* siempre comienza su ejecución en la ventana urgente, porque al principio su *buffer* está vacío. El comportamiento en ese momento consta de pedirle al mejor CDN para tratar de que la reproducción no se frene. Para saber cuál es el mejor, se guarda un promedio de los últimos tres RTT de las peticiones a cada uno de ellos. Si una respuesta no llega en determinado tiempo definido, se da como perdida y puede ser pedido a otro CDN. De lo contrario, un *chunk* no será pedido más de una vez, dado que se asume que los recursos son confiables.

Otra característica muy importante de la ventana urgente, es que se piden *chunks* consecutivos. El *buffer* del video solo puede almacenar nuevas partes del video, si son continuas a la información que ya contenía. Por lo tanto, conseguir *chunks* contiguos a los que ya fueron almacenados en el *buffer*, asegurará la continuidad de la reproducción de video. De esta manera se logrará el objetivo más importante de la ventana: asegurar que la reproducción no se detenga.

Aunque los *chunks* se piden de manera consecutiva, es posible que las respuestas lleguen en otro orden. Como se debe esperar a tener datos consecutivos para concatenar en el *buffer* del video, es necesario mantener una estructura auxiliar. Ésta sirve para almacenar cada pieza y poder recurrir a ella, tanto para ir a buscarla cuando finalmente pueda ser adherida al *buffer*, como para que luego sea compartida con otros *peers*. Por lo tanto, jugará un papel más importante en las siguientes dos ventanas.

4.5.1.2 Ventana Normal

Aquí la forma en la que se piden los *chunks* ya es más compleja. Los CDNs tienen un costo asociado para quienes pagan por sus servicios. Por lo general esos costos luego se traducen a gastos en base al ancho de banda consumido por cada uno de ellos. Es razonable pensar que un CDN más rápido y confiable es más caro que uno más lento y con una menor efectividad de respuesta.

Para ello es importante seguir utilizando a los CDNs como recurso confiable, pero a su vez hacer uso únicamente del que sea menos costoso.

Ya se sabe que el *tracker*, cuando provee información sobre los CDNs, también envía información sobre sus costos. Gracias a eso, cada cliente tiene información suficiente para restringir las peticiones al CDN más barato en los algoritmos de la ventana normal.

Esta ventana significa una transición entre las ventanas urgente y futura. Si bien se entrará en profundidad más adelante (ver la Sección *4.5.1.3 Ventana Futura*), se puede adelantar que esta última tratará de conseguir *chunks* únicamente a través de P2P. Para que ello ocurra, se debe generar un interés entre ellos. Eso significa que unos tienen que tener partes del video que les interesen a los otros y viceversa.

Es razonable pensar que si todos los *chunks* se pidiesen de manera consecutiva, los que corran primero nunca podrían pedirle nada a los que comiencen más tarde que ellos. Por lo tanto se llegó a la conclusión de que es muy importante utilizar un algoritmo que consiga partes del video que estén más adelante en la reproducción. Sin embargo, debe tenerse cuidado de no descuidar los *chunks* consecutivos que también hacen falta, porque de lo contrario la reproducción se acercaría mucho y se volvería a la ventana urgente.

Para satisfacer todas esos requerimientos, fue necesario implementar una distribución entre peticiones consecutivas y aleatorias dentro del resto de *chunks* faltantes. Esta distribución se determinó en base a serias pruebas manuales y se optó por un encare simple y efectivo. De cada tres pedidos, se pide dos veces la pieza faltante más cercana y una vez cualquiera de todas las que faltan.

A su vez, un *peer* ya conoce a esta altura los *peers* con los que puede hablar y por lo tanto ya cuenta con información sobre qué *chunks* poseen. Se decidió entonces implementar el

algoritmo de forma aún más inteligente y, a la hora de realizar peticiones, se piden únicamente *chunks* que otros no tengan. Es importante notar que esto contribuye de manera muy importante a nutrir los *swarms* con nuevas piezas de video.

Para lograr conocer qué *chunks* ya fueron obtenidos por otros *peers*, se utilizan los *bitfields* recibidos mediante el intercambio de información contextual (ver la Sección *4.4.4.1 Intercambio de información contextual*). Se implementó un módulo que se encarga de usar todos los *bitfields* de los *peers* con los que uno está hablando y calcular cuáles son los *chunks* que más rinde obtener a partir de CDN. La idea consiste en minimizar las peticiones a CDN. Para ello, se asume que si dos *peers* están hablando, potencialmente llegarán a intercambiar piezas entre ellos. Entonces, mejor es no tratar de conseguir piezas que se pueden obtener luego por P2P, a no ser que se precise más urgentemente.

Se decidió agregar en paralelo la posibilidad de conseguir *chunks* a través de P2P, aún en la ventana normal. Esta decisión es una de las grandes responsables de los altos porcentajes de P2P en las pruebas realizadas. De todas formas la explicación del P2P en sí, se explica en la sección siguiente.

4.5.1.3 Ventana Futura

En este caso se implementó un comportamiento muy similar al algoritmo *rarest-first* que emplea BitTorrent (ver la Sección *2.2.1.1 BitTorrent*). Es importante recalcar que en esta ventana se está bajo un contexto meramente P2P. Por lo tanto, el *chunk* óptimo a ser pedido se calcula en base a todos los *peers* que están dispuestos a colaborar enviando piezas.

Nuevamente haciendo uso de los *bitfields* obtenidos a través del intercambio de mensajes contextuales, se logra saber cuál es el *chunk* más raro en el *swarm* de un *peer*. El algoritmo básicamente toma todos los *bitfields* que indican qué piezas posee cada *peer* y suma cuántas veces se repite cada pieza en el *swarm*. El *chunk* con menos cantidad de repeticiones es el que más rinde compartir, y así aumentar sus chances de esparcimiento en la red.

El algoritmo también fue optimizado para que si hay varios *chunks* que empatan en ser los más raros, se pide el que esté más cercano en la reproducción. Esto ayuda a tratar de que la reproducción no se acerque al *buffereo* y se siga en la ventana futura por más tiempo.

4.5.2 Armado del *swarm*

En primer lugar, es necesario puntualizar la diferencia entre un *peer* que ya terminó de descargar el contenido y se dedica luego a compartirlo y un *peer* que aún se encuentra descargando el mismo. El primero es llamando *seeder*, mientras que el segundo *leecher*.

Cuando un *peer* solicita un nuevo *swarm* al *tracker*, este le envía su *id* y el número de *chunk* por el que va su reproducción (su índice de reproducción). El *tracker*, disponiendo de esta información conjuntamente con el índice de reproducción de los otros *peers* que conoce, genera un *swarm* de máximo diez *peers* que se distribuyen de la siguiente manera:

- Un máximo de ocho *leechers*, seleccionando aquellos cuyo índice de reproducción sea más cercano al del *peer* solicitante.

- Un máximo de tres *seeders*, elegidos de manera aleatoria entre los *seeders* disponibles por el *tracker*. Si el *swarm* parcialmente generado por *leechers* ya contiene ocho *peers*, entonces sólo se agregarán dos *seeders* para completarlo.

Cabe acotar que la implementación presentada no tiene en cuenta ningún criterio de cercanía geográfica como se estila en sistemas similares puestos en producción, debido a motivos de alcance.

4.5.3 Proceso de *Handshake*

Cuando un *peer* recibe un conjunto de *peers* desde el *tracker*, intenta establecer una conexión con cada uno de ellos para los cuales no tiene una conexión previamente establecida (dada implementación presentada, puede suceder que al solicitar nuevos *peers* al *tracker*, éste retorne *peers* que el *peer* solicitante ya tiene en su *swarm*). El éxito o no de dicho establecimiento dependerá directamente de si ambos están *interested* entre sí (ver la Sección *4.4.4.2 Intercambio de chunks*).

El proceso de *handshake* para un par de *peers* A y B se puede describir de la siguiente manera: Suponiendo que es A quien recibe al *peer* B desde el *tracker*, en caso de que A ya tenga a B dentro de su *swarm*, no intenta conectarse nuevamente. Por el contrario, si A no tenía a B dentro de su *swarm*, A envía una petición de conexión por medio de peerJS para iniciar el *handshake*. En este mensaje se incluye información contextual del peer A, como ser su *id*, su índice de ejecución y su *bitfield* (ver la Sección *4.4.4.1 Intercambio de información contextual*). Por su parte, B recibe esta información y evalúa si está interesado en A. Si no lo está, simplemente no le contesta; si a B, A le resulta interesante, entonces B le envía un mensaje HANDSHAKE_RESPONSE (ver la Sección *4.4.4.1 Intercambio de información contextual*), que contiene la misma información que la petición de conexión inicial, pero del *peer* B. Entonces, A recibe este mensaje y evalúa si el *peer* B le es atractivo para intercambiar *chunks*. En caso que no lo sea, no le contesta a B; de lo contrario le envía un mensaje HANDSHAKE_ESTABLISHED carente de contenido y cuyo único objetivo es confirmar el *handshake* por parte de A. Luego de que se envía este mensaje la condición está establecida en lo que a A respecta. Cuando B recibe este mensaje, la conexión queda formalmente establecida para ambas partes.

Dado todo el proceso que se acaba de explicar, cabe destacar las siguientes acotaciones:

Para que el proceso de *handshake* sea efectivo, el protocolo implementado depende de que el canal sea confiable[31]. Los *data channels* por defecto son confiables, utilizando un pseudo-TCP sobre UDP (cuya implementación y estudio está fuera del alcance definido para este proyecto).

Se utilizaron a nivel de implementación *timers* para que un *peer* logre detectar que el proceso de *handshake* no fue exitoso. Estos *timers* suplen la necesidad de implementar mensajes para transmitir la falta de interés en establecer la conexión con un *peer* que envía o recibe una solicitud.

Se puede decir que el proceso de *handshake* de la implementación está basado conceptualmente en el conocido *three-way handshake* que implementa TCP. En este caso

[31] *WebRTC Data Channels – draft-jesup-rtcweb-data-02*. Draft IETF. http://tools.ietf.org/id/draft-jesup-rtcweb-data-protocol-04.txt. Agosto de 2014.

no tiene el objetivo de asegurar que el medio de transporte sea confiable, sino que trata que el proceso de *handshake* logre serlo.4.5.4 Selección de *Peers*
Una vez obtenida la información sobre un *swarm* por parte del *tracker*, un *peer* conoce a un conjunto de *peers* con los que potencialmente puede intercambiar piezas. Es importante destacar que el *tracker* solo puede calcular cuáles son los óptimos a entregar en base a su cercanía de reproducción, pero no puede asegurar que de hecho se comporten de manera esperada. Es responsabilidad de cada *peer*, asegurarse de tener *peers* con los cuales intercambiar piezas y actuar siempre para mejorar su situación.

Para ello existen dos algoritmos que son los encargados de reaccionar ante situaciones que requieran mejorar la comunicación P2P: *tit-for-tat* y *optimistic unchoking*.

4.5.3.1 *Tit-for-tat*
Se adoptó el comportamiento que BitTorrent utiliza para controlar los flujos de intercambio de piezas P2P (ver la Sección *2.2.1.1 BitTorrent*). Básicamente, cuando en la transferencia de *chunks* entre dos *peers* no está siendo pareja (intercambio en ambas direcciones), se busca cerrar la conexión con el *peer* que no está cooperando. Para ello se lo *chokea* y se deja lugar a un *peer* que potencialmente pueda rendir mejor.

Cada diez segundos se ejecuta este algoritmo y en base al comportamiento anterior se encarga de recalcular cuáles son los mejores *peers* con quiénes hablar.

Lo primero que se realiza es asegurarse de que el *swarm* local del *peer* contenga diez *peers*. Puede pasar que el *tracker* en instancias previas haya entregado menos de un total de diez, y por lo tanto se pide una y otra vez hasta lograr esa cantidad. A su vez, puede ocurrir que algunos se desconecten y tengan que ser sustituidos.

Suponiendo que todos ellos están presentes, u operando con algunos menos, el *peer* tiene que recalcular con quiénes está activamente intercambiando piezas. En realidad, cada *peer* debe calcular a cuáles tiene *unchokeados*. Se agregó una lógica para restringir esa cantidad a solo cuatro, con el motivo de no sobrecargar a los *peers*.

Para ello, cada *peer* busca quiénes están interesados y se queda con los que se hayan desempeñado mejor en los últimos mensajes enviados. Si hay alguno que ya no estaba *unchokeado*, se *unchokea*. Si estaba *unchokeado* y ya no va a formar parte de lo cuatro, se *chokea*. Para medir su desempeño, se decidió ir guardando la suma de *bytes* transferidos dentro de cada uno de los períodos de *tit-for-tat* y se le denominan *byterate*. Para utilizar como medida, se promedian de manera ponderada los últimos tres *byterates*, dándole más importancia al último.

4.5.3.2 *Optimistic Unchoking*
Se adoptó también el comportamiento de BitTorrent *optimistic unchoking* (ver la Sección *2.2.1.1 BitTorrent*). Trata concretamente de, a modo periódico, darle un lugar en el intercambio de piezas a un *peer* aleatorio que no estaba activo.

Cada treinta segundos (o cada tres períodos de *tit-for-tat*), se obtienen todos los *peers* del *swarm* que estén *chokeados* y se elige a alguno de ellos, sin tener en cuenta sus previos *byterates*. De esa forma se obtiene un *peer* aleatorio a ser *unchokeado* y se lo introduce al intercambio de piezas. Si logra desempeñarse mejor que alguno de los que ya estaban

activos, conservará su lugar. Si no llega a superar a ninguno, será quitado diez segundos después, gracias a la ejecución del siguiente control de *tit-for-tat*.

El objetivo de esto consiste en darle oportunidad a la red de balancearse e incorporar a nuevos actores. No es deseable que las piezas se compartan únicamente sobre un grupo selecto de *peers*. De esta forma se da la oportunidad de expandir y mezclar los *swarms* activos.

4.6 Tecnologías utilizadas

En esta sección se explica y fundamenta el uso de algunas tecnologías y librerías que se utilizaron para implementar la solución. A modo de clasificación, se pueden dividir las tecnologías utilizadas para los servidores y para los *peers* (clientes). En la tabla 7 se aprecia la lista de tecnologías y librerías utilizadas en ambos componentes.

Cliente	**Servidor**
Media Source API (nativa de Chrome)	Node.js
Librería PeerJS para *data channels* de WebRTC	Librería socket.io para *web sockets*
Librería socket.io para *web sockets*	Librería Underscore
Librería RequireJS	
Librería Underscore	

Tabla 5. Tecnologías utilizadas para clientes y servidores

4.6.1 Node.js

Node.js[32] es una plataforma para crear aplicaciones del lado del servidor en el lenguaje JavaScript. La misma permite que estas aplicaciones sean diseñadas para maximizar su rendimiento al implementar un sistema no bloqueante de entrada/salida y eventos asíncronos.

Las aplicaciones construidas con esta plataforma corren sobre una única hebra de ejecución, a excepción de los eventos sobre manipulación de archivos y sistema de red. Esto se puede ver en la Figura 20.

Internamente, Node.js utiliza el motor de JavaScript V8 de Google para ejecutar el código, agregando una nueva cantidad de módulos escritos en el lenguaje, y quitando muchos otros que manejan eventos asociados al DOM.

Node.js ha demostrado ser muy adecuado para la implementación de aplicaciones de tiempo real (juegos *online*, salones de *chat*, *streaming multimedia*). Se optó por utilizarlo debido a las siguientes razones:

1. En primer lugar, consiste en un sistema liviano y extensible dado que el programador de la aplicación la escribe desde los cimientos, sin tener que cargar con varios módulos innecesarios que incrementan el tiempo de procesamiento.

[32] *Node.js*. http://nodejs.org/. Junio de 2013.

El único módulo adicional que se instaló sobre los servidores Node.js es *express*[33] (versión 3.0), que consiste en un *framework* muy liviano y pequeño que provee una API más amigable y robusta de las librerías HTTP de Node.js.

2. Para la comunicación entre un *peer* y el *tracker* se utilizaron *web sockets*, los cuales ya se encuentran incluidos en forma nativa en V8, y por lo tanto en Node.js. A pesar de esto, se instaló la librería socket.io tanto en los *peers* como en los servidores. Ésta consiste en una librería de más alto nivel sobre los *web sockets* y más amigable en su uso. Además, el conjunto de herramientas de socket.io fue creado para ser utilizado específicamente en un servidor Node.js, por lo que su uso resulta óptimo en la solución.
3. La plataforma cuenta con una gran comunidad[34] de desarrolladores por detrás, la cual cuenta con detallada documentación del proyecto y muy actualizada, foros de discusión, correos de noticias semanales y organización de conferencias en varios países. A su vez, la plataforma se encuentra en un constante proceso de evolución y mejora[35].
4. Finalmente, la plataforma Node.js consiste en un proyecto de código abierto, lo que se apega a las preferencias personales de los autores.

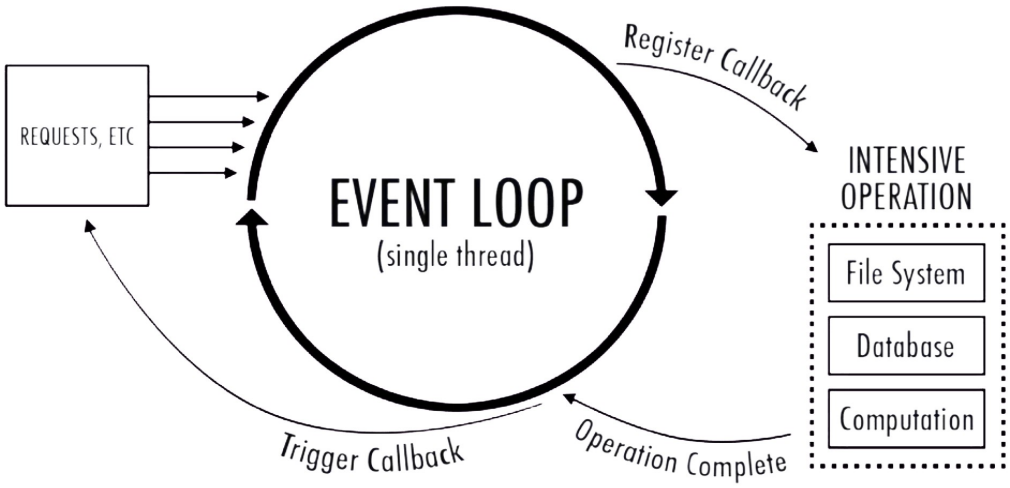

Figura 20. Arquitectura asíncrona de Node.js.

4.6.2 *Web Sockets*

Para la utilización de *web sockets*, tanto del lado del cliente como del lado del servidor, se utilizó la librería socket.io[36] (versión 0.9.16). La misma consiste en una librería JavaScript para aplicaciones *web* de tiempo real. Se conforma por dos partes: una librería que corre del lado del cliente[37], y una librería que corre del lado del servidor Node.js[38]. Al igual que Node.js su uso es orientado a eventos.

---

[33] *Express.js*. http://expressjs.com/. Junio de 2013.
[34] *Node.js Community*. http://nodejs.org/community/. Junio de 2013.
[35] *Node.js Releases*. https://github.com/joyent/node/releases. Junio de 2013.
[36] *Socket.io*. http://socket.io/. Junio de 2013.
[37] *Socket.io Client*. https://github.com/LearnBoost/socket.io-client. Junio de 2013.
[38] *Socket.io Server*. https://npmjs.org/package/socket.io. Junio de 2013.

socket.io utiliza el protocolo WebSocket como primer modo de comunicación. En caso de que el navegador no implemente el protocolo WebSocket, la librería utiliza como mecanismo alternativo otros métodos tales como JSONP *polling* o AJAX, proveyendo de la misma interfaz. A pesar de que puede ser utilizado como únicamente una librería de más alto nivel d e *web sockets*, puede ser utilizado también para realizar *broadcast* de información a múltiples *peers*, almacenar información asociada a cada cliente, y cuenta con un sistema de entrada/salida asíncrono.

La librería socket.io fue instalada en el servidor de aplicación y en el *tracker*. En el primero se instaló para que el *peer* que se conecte al servidor y obtenga la aplicación *web* del mismo ya cuente con acceso a la librería. En el caso del *tracker*, se instaló para que el cliente pueda establecer una comunicación por *web sockets* con el mismo para el intercambio de meta-información.

Se optó por utilizar esta librería en lugar de otras debido a su sencillez, y también por tratarse de la librería de *web sockets* más utilizada y mantenida por la comunidad de JavaScript en la actualidad (según el sitio *web* oficial de socket.io).

4.6.3 Underscore

Underscore[39] consiste en una librería JavaScript muy liviana que dota al lenguaje de métodos de la programación funcional. Esta librería facilitó enormemente la manipulación de objetos y arreglos de JavaScript, en particular en lo que refiere a la manipulación de los objetos que representan a los *chunks* de video. Para la solución propuesta fue utilizada la versión 1.6.0 de la librería.

4.6.4 RequireJS

RequireJS[40] consiste en un archivo JavaScript que compone un cargador de módulos del lenguaje. Fue desarrollado para desempeñarse de forma óptima en navegadores *web*, pero puede ser también utilizado en otros ambientes JavaScript, tales como los servidores Node.js (que lo traen incorporado por defecto).

El uso de este cargador de módulos JavaScript permitió estructurar la aplicación en módulos de forma más ordenada y entendible. Cada módulo contiene un bloque de código que declara sus propias dependencias y su funcionalidad.

Además, la librería mejora la velocidad y calidad de carga del código JavaScript de la aplicación en comparación con la utilización de *tags* <script> en el cógido HTML. Ésto se debe a que cada <script> en el código se corresponde con un pedido HTTP al servidor, los cuales no son paralelizables.

RequireJS contiene un optimizador de código que permite minimizar cada módulo de la aplicación y concatenarlos finalmente en un único archivo JavaScript. Esto también se traduce en que el desarrollador puede manejar las dependencias de un módulo en la forma que le resulte más amena, siendo que luego el optimizador elimine dependencias innecesarias y minimice el código. Esta herramienta permitió mejorar notoriamente el tiempo de descarga del código del *peer* desde el servidor de aplicación.

---

[39] *Underscore.js*. http://underscorejs.org/. Agosto de 2013.
[40] *RequireJS*. http://requirejs.org/. Agosto de 2013.

Se utilizó la versión 2.1.8 de la librería en la solución.

4.6.5 Media Source

Se utilió la API Media Source que se presentó en la Sección *2.3.4 Media Source* para poder concatenar *chunks* de video de formato .webm a un *tag* <video>. Esto fue realizado a modo de poder ir reproduciendo el video a medida que el *peer* recibe nuevos *chunks* por parte de sus *peers* compañeros y/o del CDN.

Se recuerda que esta API aún se encuentra en etapa de borrador por parte de la IETF, y se encuentra actualmente implementada únicamente por el navegador Chrome para el contenedor .webm (Chrome en su versión 36.0.1985.125).

4.6.6 Peerjs

Como se explicó en la Sección *2.4.2 PeerCDN*, los *data channels* de WebRTC son canales de comunicación bidireccionales de intercambio de datos que corren sobre una Peer Connection. Además, la comunicación a través de *data channels* es directa entre navegadores.

Si bien la API RTCDataChannel provista por WebRTC no es difícil de comprender, su uso puede resultar en varios errores si no se utiliza correctamente. En particular, requiere la declaración de varios *callbacks*, ya sea para la recepción de claves candidatas como para el pedido y recepción de mensajes SDP entre navegadores que tienen que estar definidos en el correcto orden. A modo de contar con una API más simple y compatible con otros navegadores se optó por utilizar la librería PeerJS[41] que es una de las más utilizadas por la comunidad de WebRTC.

La principal cualidad de esta librería es la de envolver la implementación de los *data channels* de WebRTC para proveer una API más sencilla de usar, pero sin perder todas las funcionalidades y configuraciones provistas por la original. Su modo de funcionamiento es en base a un identificador único, el cual es utilizado por un *peer* para crear conexiones P2P de datos o de *streaming* a un *peer* remoto.

Este modo de conexión entre dos *peers* es el más implementado por las librerías que proveen una funcionalidad similar. El procedimiento habitual consiste en que un servidor de aplicación sirva una página/aplicación *web* a dos o más clientes, la cual cuenta también con un identificador único que es utilizado por un servidor de señalización para establecer el enlace entre los navegadores. Este tipo de identificador de funcionamiento puede ser comparado con el de una aplicación de *chat* en la cual se establecen grupos de participantes. Aplicaciones *web* tales como ShareFest[42] o PubNub[43] se basan en este procedimiento para establecer la conexión entre navegadores.

El proyecto PeerJS aún se encuentra en una etapa inicial debido a la rapidez con que varía la especificación de los *data channels*, lo cual trajo complicaciones al momento de actualizar la versión del navegador Chrome. En la solución propuesta se cuenta con la versión 0.3.7 que corre en Chrome y Chrome Canary. PeerJS consiste en un archivo que se incluye en

[41] *PeerJS*. http://peerjs.com/. Diciembre de 2013.
[42] *ShareFest.me*. http://sharefest.me/. Diciembre de 2013.
[43] *PubNub*. https://github.com/pubnub/webrtc. Noviembre de 2013.

los clientes y de un servidor Node.js de señalización que puede descargarse de GitHub[44] para realizar el *test* local de la aplicación.

El proyecto PeerJS provee un servidor de señalización que puede correrse en forma local o referenciarse en la nube, el cual se encarga de implementar el intercambio de claves candidatas del procedimiento ICE entre los *peers* que se interconectan. El hecho de contar con este servidor ahorró el no tener que codificar un servidor que se encargue de estas funcionalidades de bajo nivel. en la Figura 18 se puede ver como el servidor peerJS interactúa con el resto de las entidades de la arquitectura.

[44] *PeerJS Server*. https://github.com/peers/peerjs-server. Enero de 2014.

5 Pruebas realizadas

5.1 Objetivos

El objetivo de la etapa de pruebas consiste en analizar el comportamiento del prototipo en base a varios escenarios realistas, y tomar conclusiones a partir de los resultados obtenidos

Las pruebas se corren de manera que sea posible aplicar un proceso que sea repetible, cuantificable y comparable entre los diferentes casos a analizar.

Se examina el sistema como una “caja negra”. Se define un conjunto de casos de prueba que abarcan la totalidad de las variantes interesantes que están dentro del alcance del proyecto. Los escenarios resultantes son considerados como las hipótesis de un contexto determinado, y al hacerlas variar debe ser posible cubrir suficientes escenarios que permitan analizar las características del sistema y al mismo tiempo comparar los resultados entre sí.

Se plantea como objetivo principal analizar dos atributos: calidad percibida por los usuarios finales y ahorro en el uso de los CDNs.

La calidad percibida del sistema refiere a una medición de la experiencia de usuario al utilizar el sitio. En el caso del prototipo, se traduce en el tiempo que el usuario debe esperar a que el video reproduzca. En otras palabras, lo que se mide es la cantidad de segundos y la cantidad de oportunidades en que el video está “trancado” sin que el usuario pueda hacer nada para destrancarlo.

Por otra parte, el ahorro refiere a la cantidad de *chunks* que, pudiendo ser descargados desde un CDN (lo cual representaría un costo monetario), terminan siendo descargados desde otros *peers* activos del sistema haciendo P2P durante la reproducción. Así mismo, también interesa medir la cantidad de *chunks* descargados desde el CDN1 (más potente y más costoso), contra la cantidad de *chunks* descargados desde el CDN2 (menos potente y más barato).

La estrategia consiste en utilizar las estadísticas almacenadas en el *tracker* como herramienta de medición, y un *framework* de *testing* desarrollado para las necesidades de esta etapa, como mecanismo para correr cada uno de los escenarios de prueba de una manera repetible y comparable (ver la Sección *5.2 Infraestructura*).

El resultado final de la etapa de pruebas explica en qué casos el sistema se desempeña de mejor o peor manera, nociones numéricas de ahorro, valores de calidad percibida y algunas gráficas comparativas de los resultados.

5.2 Infraestructura

Dado que el sistema construido tiene una componente P2P muy importante, el ambiente de pruebas considera que para obtener resultados que tengan sentido, se debe contar con un *swarm* de *peers* que estén utilizando el sistema de manera concurrente.

La métrica de ahorro no tendría sentido si no existieran suficientes *peers* en el sistema con quien poder realizar P2P. Esto ocurre, por ejemplo, para el caso en el que solo un *peer*

existe en el sistema. Bajo ese concepto, el 100% de los *chunks* serían descargados desde los CDNs. Por lo tanto, es necesario generar un escenario más realista, donde existan más usuarios en la red.

La infraestructura de pruebas consta de:

- Nueve computadoras portátiles.
- Cuatro computadoras virtualizadas.
- Una red WLAN (incluyendo las nueve portátiles).
- Una LAN (incluyendo las cuatro máquinas virtuales).
- La interfaz de estadísticas provista por el *tracker*.
- Un *framework* de *testing* que ayuda a orquestar las pruebas de manera remota.

Cada computadora se corresponde con un *peer* participando en el sistema. Esto significa que cada computadora mantiene un navegador *web* abierto (Google Chrome) y accediendo (en un momento dado) a la URL donde se encuentra alojado el sistema.

Las nueve computadoras reales son MacBooks Pro corriendo OSX como sistema operativo, con procesadores Intel Core i5 de 2.6Ghz, 8 GB de RAM y GPUs dedicadas. Por otra parte, las otras cuatro máquinas son *Ubuntu servers 12.04* virtualizados de 1Gb de RAM y un CPU de 3.4GHz, residentes en un *host* físico corriendo CentOS con 20GB de RAM y CPU cores de 3.4GHz.

Si bien es posible simular dos o más *peers* corriendo en una sola computadora (por ejemplo, abriendo dos o más pestañas del navegador), es deseable crear el escenario más realista posible. El caso ideal consiste en poder colocar cada una de las computadoras disponibles detrás de un NAT diferente, con direcciones IP públicas diferentes. Sin embargo, emular dicho caso resulta muy difícil de implementar, dado que la coordinación entre computadoras se torna compleja. Por lo tanto, se optó por utilizar sólo dos subredes diferentes, cada una detrás de una NAT diferente. Cada nodo simula un y solo un *peer*, representando el caso más realista posible dentro del alcance viable.

La red WLAN listada anteriormente es la que brinda conectividad a las nueve computadoras físicas y da acceso a Internet a través de dos *modems* empresariales (*modems* de 120 Mb/s de bajada y 20 Mb/s de subida cada uno, con un balanceador para distribuir la carga de manera equitativa). Por otra parte, la red LAN que aloja a las otras cuatro máquinas virtuales se encuentra en otro lugar físico y no comparte la misma subred que la WLAN previamente mencionada. El *modem* que brinda conectividad a Internet a la LAN tiene un plan de 6 Mb/s de bajada y 3 Mb/s de subida. La Figura 21 muestra un diagrama simplificado de la infraestructura descrita.

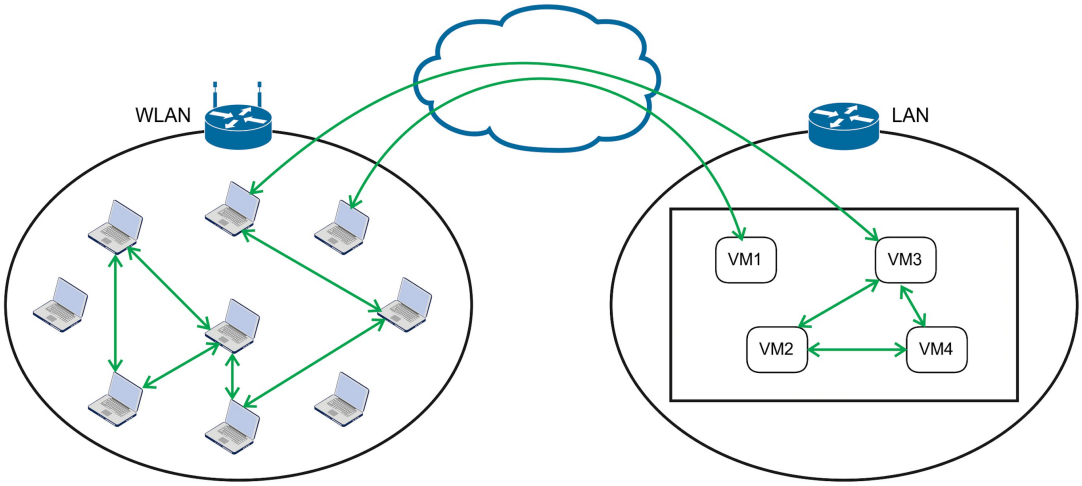

Figura 21: Estructuramiento físico del sistema implementado.

La salida de cada caso de prueba consta de un conjunto de estadísticas resumidos en el *tracker* (accedidos a través de una página *web*). Éstas se generan a medida que los diferentes *peers* ingresan al sistema. Como se explicó en la Sección *4.4.2.1 Mensaje Keep-Alive*, a medida que transcurre la ejecución del video, cada *peer* reporta una serie de datos al *tracker* con la finalidad de generar las estadísticas finales. Las Figura 22 y Figura 23 muestran un la interfaz del servicio *web* para un escenario de prueba.

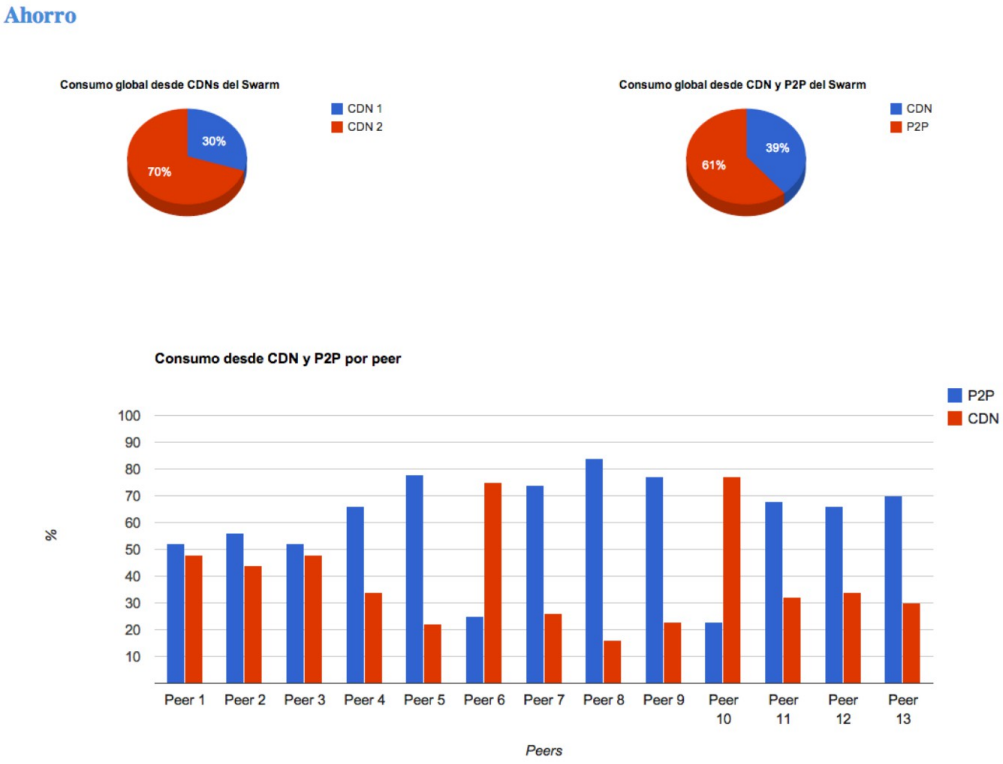

Figura 22: Ahorro del Tracker.

Calidad

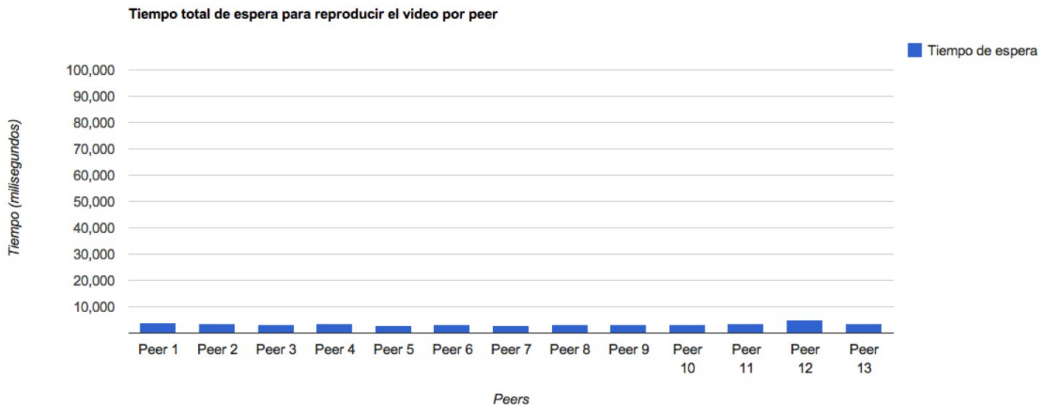

Figura 23: Calidad del Tracker. Tiempo total de espera para reproducir el video por peer.

Como se explica en la Sección *5.3 Casos de prueba*, el cubrimiento de los casos de prueba está dado en gran parte por el estado del *swarm* en el sistema en un momento dado. La definición de un caso de prueba depende del momento y la distribución en que los *peers* ingresan al sistema (accediendo a través del navegador a la URL previamente mencionada), por lo que es deseable poder controlar de manera automatizable la simulación de dichos escenarios. Además, es importante que cada caso de prueba sea repetible en el tiempo. Por lo que, para solucionar esta problemática se construyó un *framework* de *testing* que permite:

- Sincronizar de manera remota el ingreso al sistema de un conjunto de *peers* mediante el envío automatizado de paquetes HTTP entre los nodos de la infraestructura.
- Dar de baja y/o eliminar *peers* indeseados que potencialmente estuvieran activos en el sistema.
- Plasmar un escenario de prueba en un archivo de configuración, de manera de poder iniciar la ejecución del caso por medio de éste. Esto brinda la repetibilidad necesaria, ya que cada caso de prueba queda descrito en un documento que no se volvió a modificar.

Cada una de las computadoras de la infraestructura tiene instalado el *framework* de *testing*. Los detalles de arquitectura y tecnologías utilizados para construirlo se encuentran en el *Anexo 10 Framework de Testing*.

5.3 Casos de prueba

El objetivo de los casos de prueba es estudiar cómo se desempeña el sistema bajo diferentes condiciones a través de la observación sistemática de las métricas resultantes de ahorro y de calidad accesibles en el *tracker*.

Para realizar el estudio se definieron nueve escenarios de prueba. Se entiende que los escenarios seleccionados representan una cobertura aceptable dentro del alcance del prototipo. Cada uno de ellos está compuesto por trece *peers* que interactuarán en el sistema y la diferencia de cada escenario está dada por la distribución en la que dichos *peers* ingresan al sistema.

Por ejemplo, algunos escenarios se centran en distribuciones que hacen variar la cantidad de *seeders* en el sistema, mientras que otros se enfocan en el comportamiento del mismo cuando los *peers* ingresan como “ráfagas” de determinada cantidad de *peers*. Se definieron tres categorías de escenarios:

- Porcentaje de *seeders* en el sistema.
- Distribución de ingreso.
- Ráfagas de *peers*.

Cada una de ellas cuenta con tres casos de prueba distintos.

Vale la pena aclarar que cada escenario de prueba fue ejecutado tres veces y los resultados obtenidos documentados debidamente para cada ejecución (ver el *Anexo 11 Resultados de las pruebas realizadas*), de manera de poder promediar los valores finales de cada escenario.

El video utilizado en el prototipo de formato WebM tiene una duración de quince minutos y está codificado utilizando la herramienta ffmpeg[45]. Para dicha codificación se utilizaron las librerías libvpx (VP8) y libvorbis (Vorbis) para video y audio respectivamente. El comando ffmpeg utilizado fue el siguiente:

```
ffmpeg -i toTranscode.mp4 -codec:v libvpx -quality good -cpu-used 0 -b:v 500k -qmin 10 -qmax 42 -maxrate 500k -bufsize 1000k -threads 4 -g 120 -keyint_min 120 -codec:a libvorbis -b:a 128k transcoded_5sec.webm
```

La Tabla Tabla 6: Información del video utilizado por el prototipo. muestra en detalle las propiedades del video.

Video		**Audio**	
Codec	Google/On2's VP8 Video (VP80)	**Codec**	Vorbis Audio (vorb)
Resolución	640x360 pixels	**Canales**	Estéreo
Frame Rate	29.970628 fps	**Sample rate**	44100Hz
		Bitrate	112kb/s

Tabla 6: Información del video utilizado por el prototipo.

5.3.1 Porcentaje de *seeders* en el sistema

Se le dio énfasis en cómo impactan la cantidad de *seeders* en el sistema. Se construye la aplicación de manera que cuando un *peer* termina de descargar todos los *chunks*, el mismo pasa a estado de *seeder*, donde abandona las políticas restrictivas del P2P y pasa a ser absolutamente benevolente con todo aquel que le pida un *chunk*.

Es decir que los *seeders* aportan todos los *chunks* que le son pedidos, sin importar si quien quiere conectarse con él tiene *chunks* para ofrecer o no, lo cual tiene sentido ya que a un *seeder* no le interesa recibir más *chunks*. Un *seeder* tiene como única restricción la cantidad de *peers* con los que puede hablar al mismo tiempo (que suma un total de cuatro).

[45] *ffmpeg*. http://www.ffmpeg.org/. Agosto de 2013.

Por lo tanto, se tiende a pensar que con cierto porcentaje de *seeders* presentes en el sistema, el ahorro en CDNs aumente, o dicho de otra manera, que se realice más P2P.

Se analiza esta situación haciendo variar el porcentaje de *seeders*. Para ello, se definen tres casos de prueba dentro de esta categoría:

- **Caso A.** 0% de *seeders* en el sistema.
- **Caso B.** 15% de *seeders* en el sistema.
- **Caso C.** 30% de *seeders* en el sistema.

Cada escenario cuenta con trece *peers* en el sistema, los cuales ingresan al mismo con una distribución de tiempo constante. En el Caso A, los *peers* irán ingresando, uno cada cinco segundos, del primero al décimo tercero.

A medida que los *peers* van ingresando y que el video avanza en la reproducción, los mismos comenzarán a intercambiar *chunks*. Cada uno de esos eventos se reporta al *tracker*, quien construye las estadísticas de ahorro y de calidad. Al finalizar la reproducción, se utiliza dicha información para documentar los valores finales de ese escenario. El mismo caso de prueba se ejecuta tres veces en total y se promedian los valores finales de las tres corridas.

Para los casos B y C, el procedimiento es equivalente, con la salvedad de que al iniciar la ejecución ya se cuenta con *seeders*. Por ejemplo, para el caso B se cuenta con dos *seeders* además de los trece *peers* al iniciar la ejecución. En el caso C, se utilizan cuatro *seeders* además de los trece *peers*, que también ingresan cada cinco segundos en el sistema de manera paulatina.

Luego de haber recabado toda la información, se puede realizar un análisis de cómo impacta el cantidad de *seeders* en el sistema. Los resultados son estudiados en la Sección *5.4 Resultados y conclusiones*.

5.3.2 Distribución de ingreso

Se plantea el objetivo de estudiar el comportamiento del sistema al variar la distribución de entrada de los *peers*. En particular, interesa analizar qué sucede cuando los *peers* ingresan simultáneamente, de manera constante, o aleatoria al sistema.

Una de las políticas de P2P consiste en dar *chunks* únicamente a aquellos *peers* que tienen *chunks* que puedan interesar. Dependiendo de la implementación, esta política puede presentar trabas indeseables en la realización de P2P. Por ejemplo, en una implementación donde todos los *chunks* se piden de manera secuencial, es difícil que dos *peers* que ingresan al sistema de manera lejana en el tiempo se interesen entre sí y puedan realizar P2P.

El prototipo posee optimizaciones relacionadas a esto último (ver la Sección *4.5.1.2 Ventana Normal*), por lo que interesa estudiar cómo afecta al sistema cuando los *peers* ingresan en alguno de los tres escenarios siguientes:

- **Caso A.** Todos al mismo tiempo.
- **Caso B.** Tiempo de ingreso constante (Uno cada cinco segundos).
- **Caso C.** En tiempo aleatorio (de cero a dos minutos).

Luego de ejecutados los casos de prueba se podrá distinguir si hay diferencias en las estadísticas de ahorro y calidad entre los escenarios. Lo deseable sería no identificar ningún caso donde alguna de las dos métricas se vea perjudicada.

5.3.3 Ráfagas de *peers*

Como se explicó en la Sección *4.5.3.1 Tit-for-tat*, la cantidad máxima de *peers* con las que otro *peer* puede estar intercambiando *chunks* de manera concurrente está limitada a un máximo de cuatro.

Al mismo tiempo, por las características de las políticas P2P, es posible que en ciertos casos se formen subconjuntos de *peers* que solo hablan entre ellos y dejen fuera de la conversación a *peers* más nuevos. Es deseable que estos casos no afecten el consumo general del CDN y que la media de P2P se mantenga siempre que sea posible. Se espera que los nuevos *peers* que ingresen en el sistema sean capaces de compartir *chunks* entre ellos y no entrar a un estado indeseable como puede ser el de intentar conectarse continuamente con *peers* más avanzados en la descarga que difícilmente se interesen en *peers* más “nuevos”, dificultando el intercambio P2P.

Para un sistema como el implementado, resulta interesante analizar cómo impactan distribuciones de *peers* que entran al sistema en forma de “ráfagas”. Por ejemplo, qué pasa cuando cinco *peers* se encuentran descargando o reproduciendo el video en un estado en el que ya se encuentran realizando P2P entre ellos y diez minutos después, cuatro *peers* más ingresan en un intervalo de veinte segundos.

Importa verificar que los nuevos *peers* son también capaces de realizar P2P, que su ingreso no afecta el rendimiento de los *peers* que ya se encontraban en el sistema y que la media total de P2P no disminuya demasiado.

Para analizar estos casos se definieron tres escenarios más de prueba:

- **Caso A**. Ráfagas de dos *peers*: cinco ráfagas de dos *peers* y una ráfaga de tres *peers*, lanzadas una vez cada tres minutos.
- **Caso B**. Ráfagas de cuatro *peers*: dos ráfagas de cuatro *peers* y una ráfaga de cinco *peers*, lanzadas una vez cada cuatro minutos.
- **Caso C**. Una ráfaga de seis *peers* y otra ráfaga de siete *peers*, una cada cinco minutos.

Como para el resto de los casos, la idea es que cada escenario sea ejecutado tres veces y los valores finales sean promediados.

5.4 Resultados y conclusiones

La etapa de pruebas generó como resultado un conjunto de estadísticas sobre el sistema que se encuentran en el *Anexo 11 Resultados de las pruebas realizadas*. La presente sección realiza un estudio resumido de los valores obtenidos de los casos de prueba y hace

un análisis enteramente basado en dichas estadísticas y en el conocimiento del funcionamiento del sistema.

La infraestructura utilizada y los escenarios probados se encuentran definidos en las Secciones *5.2 Infraestructura* y *5.3 Casos de prueba*, respectivamente. Se analizan las estadísticas de ahorro y calidad.

Los valores de calidad son las medidas del tiempo en el que el usuario está esperando a que el video se reproduzca. Hay dos medidas a tomar en cuenta:

- **Tiempo de espera inicial**: el tiempo que pasa desde que el usuario ingresa al sitio y el video comienza a reproducir.
- **Tiempo de espera por *buffereo***: el tiempo que pasa cuando el video se detiene en la mitad de la reproducción debido a que el *chunk* que se debe reproducir en ese momento todavía no fue descargado.

Para medir esos tiempos se utilizaron eventos provistos por la API del *tag* de video de HTML5. Lamentablemente, dichos eventos no funcionan de la manera esperada en algunos casos dado que la API provista por el elemento <video> es muy incipiente. Consecuentemente, resultó imposible medir el Tiempo de espera por *buffereo*. La métrica que se utilizó para medir la calidad es el Tiempo de espera inicial y durante las pruebas se observaron las reproducciones para ver, a grandes rasgos, el Tiempo de espera por *buffereo*, que resultó ser despreciable en la gran mayoría de los casos.

La métrica de ahorro cuenta también con dos componentes:

- **Ahorro de CDN1 vs CDN2**: el CDN2 es el servidor de contenido más económico. Por lo tanto, se busca minimizar la utilización del CDN1 sin comprometer la calidad percibida.
- **Ahorro de CDN vs P2P**: se busca maximizar el P2P sin comprometer la calidad percibida.

Para medir el ahorro simplemente se llevan contadores que almacenan desde dónde se descargó un *chunk* y luego se reportan al *tracker* para que arme sus estadísticas globales. Vale la pena aclarar que la infraestructura planteada presenta varios nodos compartiendo una misma red local. Por lo tanto, el ancho de banda existente entre estos nodos es alto. Es un buen escenario para analizar el sistema en condiciones casi ideales, donde hay pocas restricciones de ancho de banda para realizar P2P con nodos que pertenezcan a la misma red.

Se comprobó a través de pruebas puntuales que el prototipo funciona también en escenarios donde los *peers* se encuentran detrás de NATs. En ellos se pudo ver el intercambio de mensajes tanto a través de servidores de *relay* como directamente entre *peers*. En un trabajo futuro tiene sentido estudiar el rendimiento del prototipo bajo estos escenarios.

5.4.1 Estadísticas de ahorro

5.4.1.1 Porcentaje de *seeders* en el sistema

Este escenario plantea estudiar el comportamiento del sistema cuando incrementa o disminuye la cantidad de *seeders* al inicio de la ejecución. La descripción detallada del mismo se encuentra en la Sección *5.3.1 Porcentaje de seeders en el sistema*.

En la Figura 24, Figura 25 y Figura 26 pueden verse los resultados de las gráficas de ahorro generadas en el *tracker* para cada uno los escenarios planteados. Notar que cada escenario se corrió tres veces, por lo que cada gráfica representa solo una de las tres ejecuciones de cada caso. La lista completa de resultados puede verse en el *Anexo 11 Resultados de las pruebas realizadas*, pero como los resultados de cada una de las tres corridas son muy similares, es posible utilizar las figuras mencionadas para desarrollar el análisis.

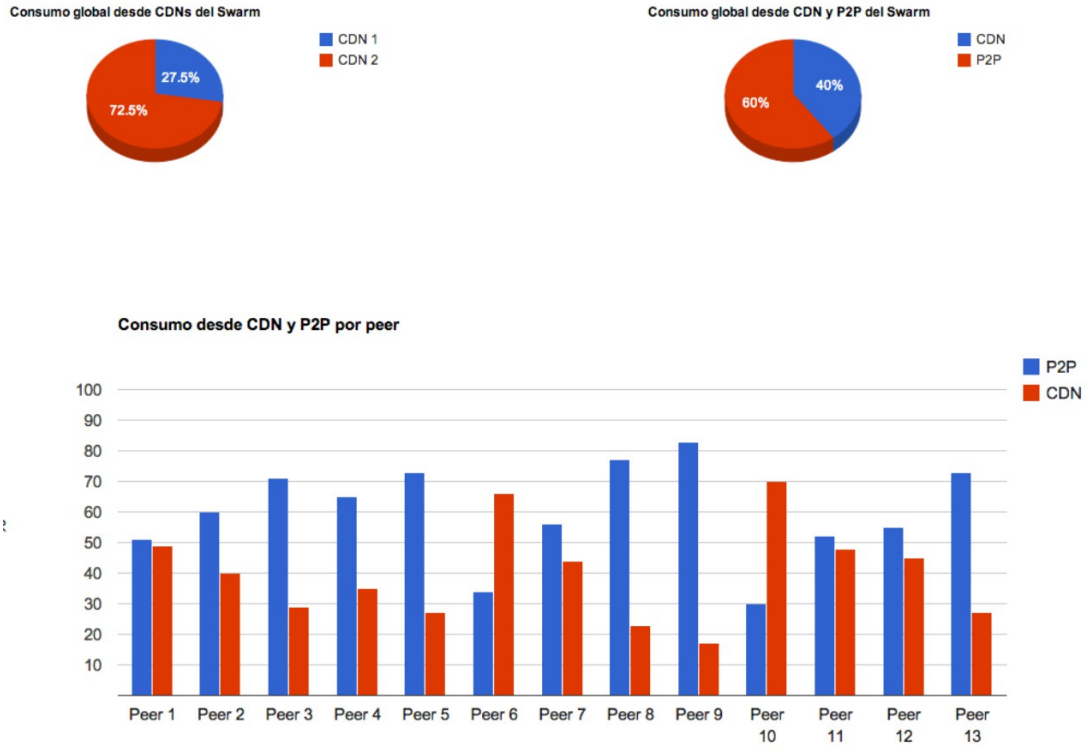

Figura 24: Caso A. Estadísticas de ahorro, 0% seeders.

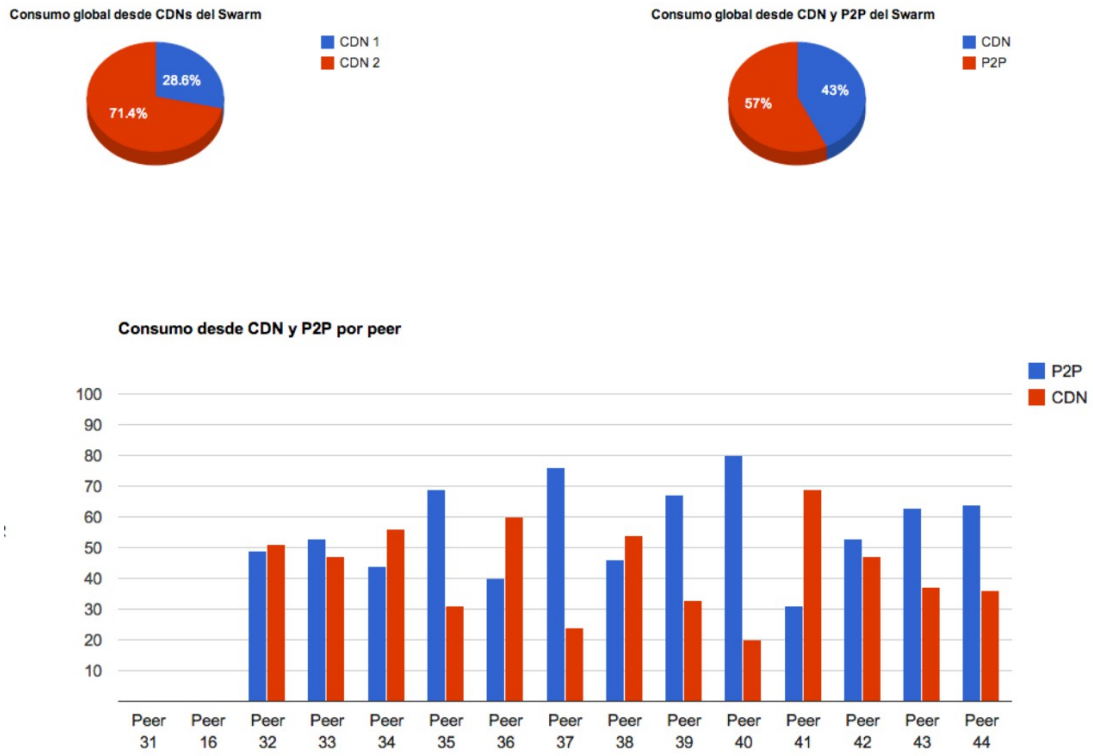

Figura 25: Caso B. Estadísticas de ahorro, 15% seeders.

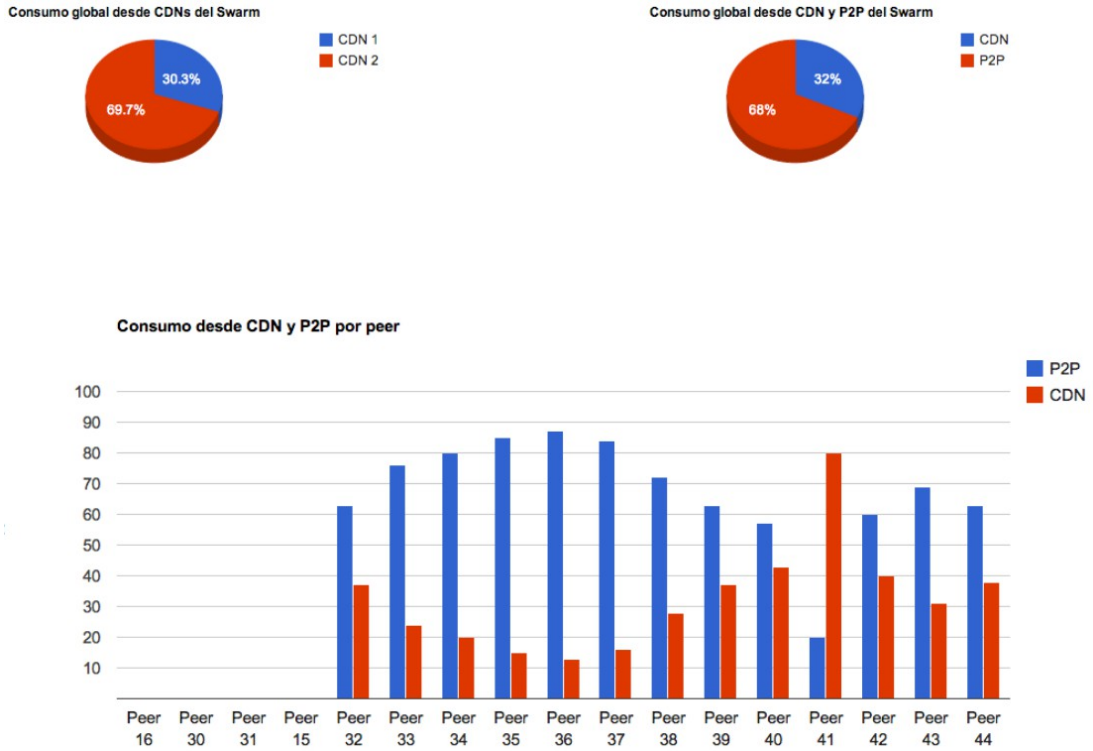

Figura 26: Caso C. Estadísticas de ahorro, 30% seeders.

Los tres escenarios de prueba dieron como resultado valores alentadores, inclusive sin *seeders* en el sistema. Los resultados muestran que el sistema soporta un ahorro mayor al 57% (de utilización de CDN). Es decir que al menos el 57% de los *chunks* intercambiados durante la reproducción del video fueron realizando P2P, sin utilizar el CDN.

Algo peculiar es que el caso A reporta un mejor ahorro que el B, el error es pequeño (menor al 3%) por lo que se concluye que un 15% de *seeders* en el sistema realmente no impacta de manera trascendente. Tomando en cuenta que cada *peer* tiene un tope máximo de cuatro *peers* con quien intercambiar *chunks* concurrentemente y dado que el sistema comienza a hacer P2P de manera temprana, aparecen *peers* que rápidamente aportan *chunks* tan bien como unos pocos *seeders*.

Por otra parte, cuando la cantidad de *seeders* aumenta, el resto de los *peers* no compiten de la misma forma y los *seeders* logran incrementar el P2P global. El caso C es un ejemplo donde se demuestra un incremento considerable en el ahorro, logrando un 68% de utilización de P2P.

Otro hecho interesante es que en todos los casos el porcentaje de utilización del CDN1 (el más caro y más potente) es mucho menor que del CDN2 (el peor caso es un 30%). La razón es estrictamente por la implementación de los algoritmos, que prefieren al CDN2 siempre que sea posible (ver la Sección *4.5.1 Selección de chunks*). Las distribuciones de los *seeders* realmente no afectan este aspecto, dado que la decisión de optar por un CDN se toma en las ventanas urgente y normal y no dependen del P2P.

También puede verse en la Figura 24, Figura 25 y Figura 26 el consumo discriminando por *peer* (notar que la Figura 25 y la Figura 26 tienen algunas entradas libres en el eje de los *peers*, dichas entradas representan a los *seeders*). Lo que se destaca de estas gráficas es que no hay saltos importantes de consumo por *peer*. Es decir, no hay *peers* que hayan quedado absolutamente por fuera de la interacción P2P. Esto demuestra que la red P2P construida es una red justa, que es una propiedad importante en este tipo de sistemas.

5.4.1.2 Distribución de ingreso

El detalle de la definición de este escenario puede verse en la Sección *5.3.2 Distribución de ingreso*. Se estudia el comportamiento del sistema cuando se hacen variar las distribuciones de ingreso de los *peers*, por lo que se definen tres casos: ingreso todos al mismo tiempo, ingreso en intervalos de tiempo constante e ingreso en intervalos de tiempo aleatorio. En la Figura 27, la Figura 28 y la Figura 29 se ven los resultados gráficos de los tres.

Tal como sucede en el caso de 0% de *seeders* en el sistema, se utiliza sólo una de las tres gráficas generadas para cada caso como herramienta de análisis para esta sección, dado que los valores resultantes son muy similares entre sí (ver en el *Anexo 11 Resultados de las pruebas realizadas*).

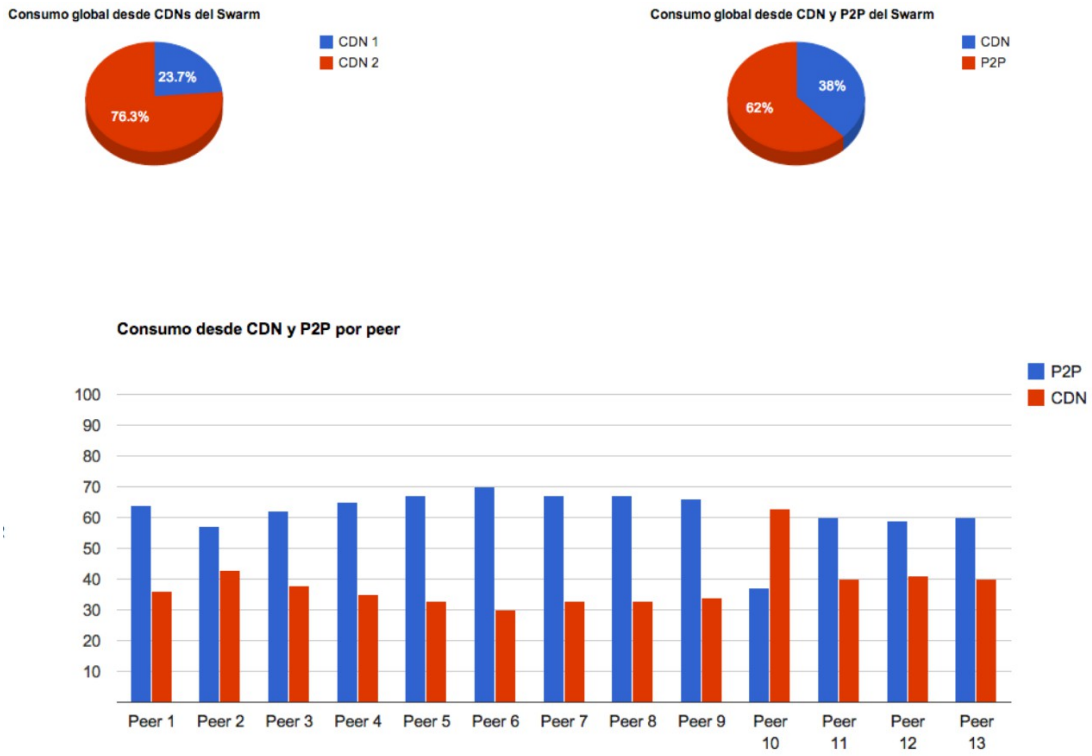

Figura 27: Caso A. Ingreso todos al mismo tiempo.

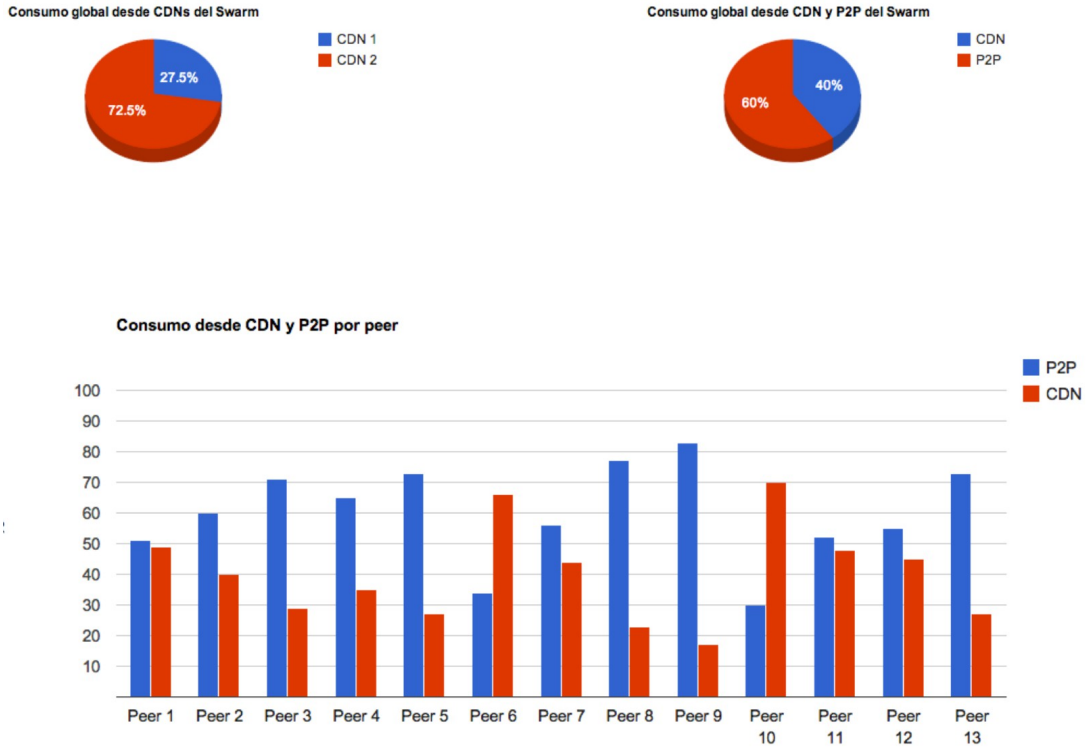

Figura 28: Caso B. Ingreso en tiempo constante, uno cada cinco segundos.

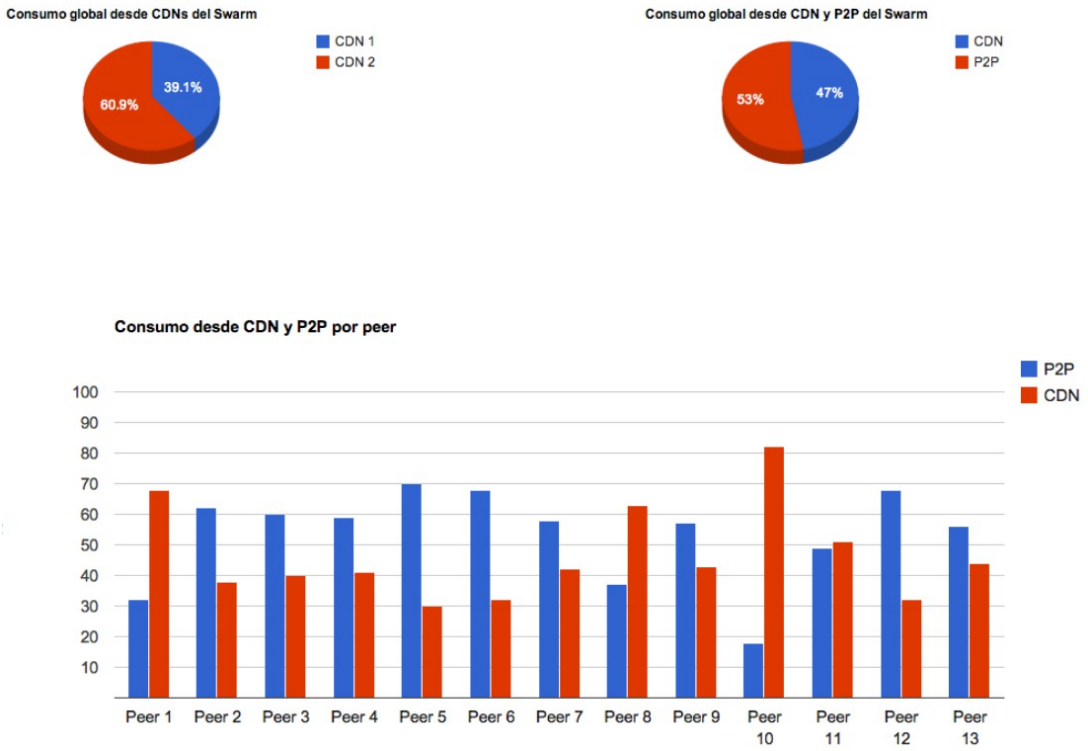

Figura 29: Caso C. Ingreso en tiempos aleatorios (de cero a dos minutos).

Como puede verse en la Figura 27, la Figura 28 y la Figura 29, todos los casos de esta sección se corren sin *seeders* presentes en el sistema al inicio de la ejecución. El Caso A muestra los resultados de probar el sistema cuando todos los *peers* entran exactamente al mismo tiempo, el caso B muestra lo propio para la situación en que los *peers* ingresan en tiempo constante la sistema (intervalos de cinco segundos) y el caso C brinda los resultados cuando los *peers* ingresan de manera aleatoria.

El caso A y el caso B presentan resultados muy similares, puede verse en la Figura 27 y Figura 28 que la distribución de P2P por *peer* es relativamente pareja y que el porcentaje de utilización P2P varía de 60% a 62%. Por otra parte, en el caso C, el porcentaje de utilización P2P (si bien es importante) disminuye a 53%.

Como se explica en la Sección *4.5.2 Armado del swarm*, el *tracker* devuelve un subconjunto de *peers* con los que potencialmente intercambiar *chunks*. El algoritmo del *tracker* selecciona los *peers* más cercanos en la reproducción. Para el caso A y el caso B, en poco tiempo el sistema incluye una buena cantidad de *peers* cercanos entre sí, mientras que para el caso C los *peers* ingresan al sistema de manera aleatoria, pero probablemente mucho más lento que en los otros dos escenarios, dado que la función utilizada devuelve un valor entre cero y dos minutos para el intervalo de tiempo de ingreso entre un *peer* y otro. Esta variación de tiempo es fundamental para explicar la disminución del porcentaje de ahorro.

El sistema saca más provecho del P2P cuando el intervalo de ingreso es menor. Cuando la frecuencia de entrada disminuye, el tiempo que debe pasar para que los *peers* puedan conseguir *chunks* que interesen al resto es más largo. Esto se traduce en que se acorta el tiempo en el que se intercambian *chunks* por P2P, y como la cantidad de *chunks* del video es finita, termina impactando en un porcentaje de intercambio menor al de la media. Tomar

en cuenta que las políticas de descarga de *chunks* “raros” (o sea, *chunks* que puedan terminar siendo interesantes para el resto) solo se da en ventanas normal y futura.

De todas formas 53% de ahorro es un valor aún muy bueno, incluso para este caso menos amigable. Se considera que si el video utilizado fuese más largo se estimaría el efecto transitorio y se alcanzaría el mismo ahorro de los casos A y B.

Con respecto al porcentaje de utilización de CDN1 y CDN2, se ve cómo el mismo se mantiene de manera similar al escenario de Porcentajes de *seeders*. La utilización del CDN2 sigue siendo mayor que la del CDN1. Para el caso A y B, se mantuvo por encima del 70% de utilización del CDN2 (el más barato), mientras que en el caso C la utilización del CDN1 fue más importante (casi un 40%). Esto es coherente con la explicación anterior sobre la variación del porcentaje de P2P en el caso C, dado que se pasó más tiempo en la ventana urgente donde normalmente se utiliza el CDN1 para descargar los *chunks*.

5.4.1.3 Rafagas de *peers*

La definición de este escenario está definido en la Sección *5.3.3 Ráfagas de peers* y se utilizan los casos A, B y C descritos en esa sección. Se analiza cómo responde el sistema frente a diferentes ráfagas de *peers* ingresando al mismo tiempo.

Los resultados obtenidos tras las ejecuciones de los casos de prueba definidos para este escenario no son tan constantes como en los escenarios anteriores. Para el caso A el porcentaje de uso varía desde un 45% a un 61% de uso de P2P, el caso B es más estable, con una variación de 57% a 59% y el caso C varía de 43% a 60%. Los resultados completos pueden verse en el *Anexo 11 Resultados de las pruebas realizadas* y la explicación de los casos en que el P2P disminuye se da en que siempre algún *peer* no logra “subirse” a ningún *swarm*, las gráficas de distribución por *peer* lo dejan en claro. Cuando la distribución de ingreso se da por ráfagas, es más normal que algún peer quede por fuera de la comunicación y no logre conectarse con otros *peers* por más de unos pocos segundos.

Como la petición de *chunks* “raros” solo se da en ventana normal y ventana futura, si un *peer* no logra salir de la ventana urgente por mucho tiempo, entonces la probabilidad de tener *chunks* que interesen al resto disminuye cuando se es uno de los últimos *peers* en ingresar al sistema. Al mismo tiempo, si la frecuencia de entrada es baja, entonces sucede que existen *peers* que quedan más avanzados en la reproducción y difícilmente puedan interesarse en los nuevos *peers* con *chunks* poco raros. Se genera un círculo vicioso: los *peers* que no logran hacer P2P de la mejor manera tampoco logran salir de la ventana urgente con facilidad porque no logran ser interesantes al resto de sus vecinos, probablemente porque no tiene *chunks* raros (que se piden fuera de la ventana urgente).

La Figura 30, Figura 31 y Figura 32 muestran un ejemplo de resultado de ejecución para cada uno de los tres casos de prueba definidos.

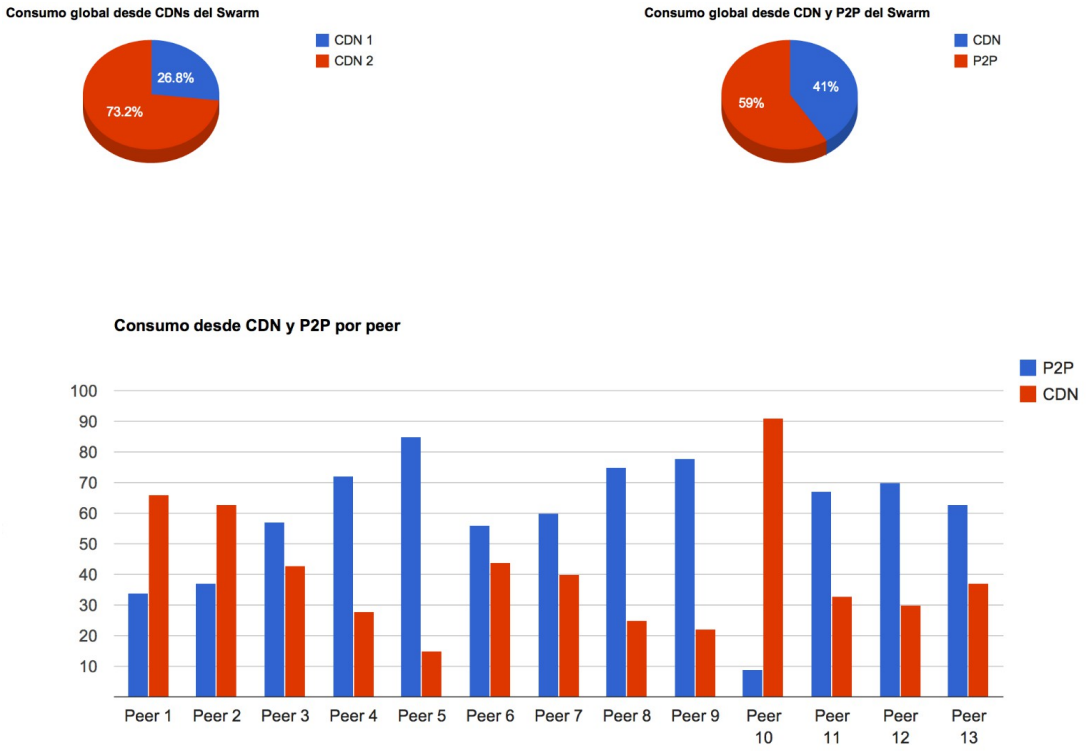

Figura 30: Caso A. Ráfagas de dos peers. Cinco ráfagas de dos peers y una ráfaga de tres peers, lanzadas una vez cada tres minutos.

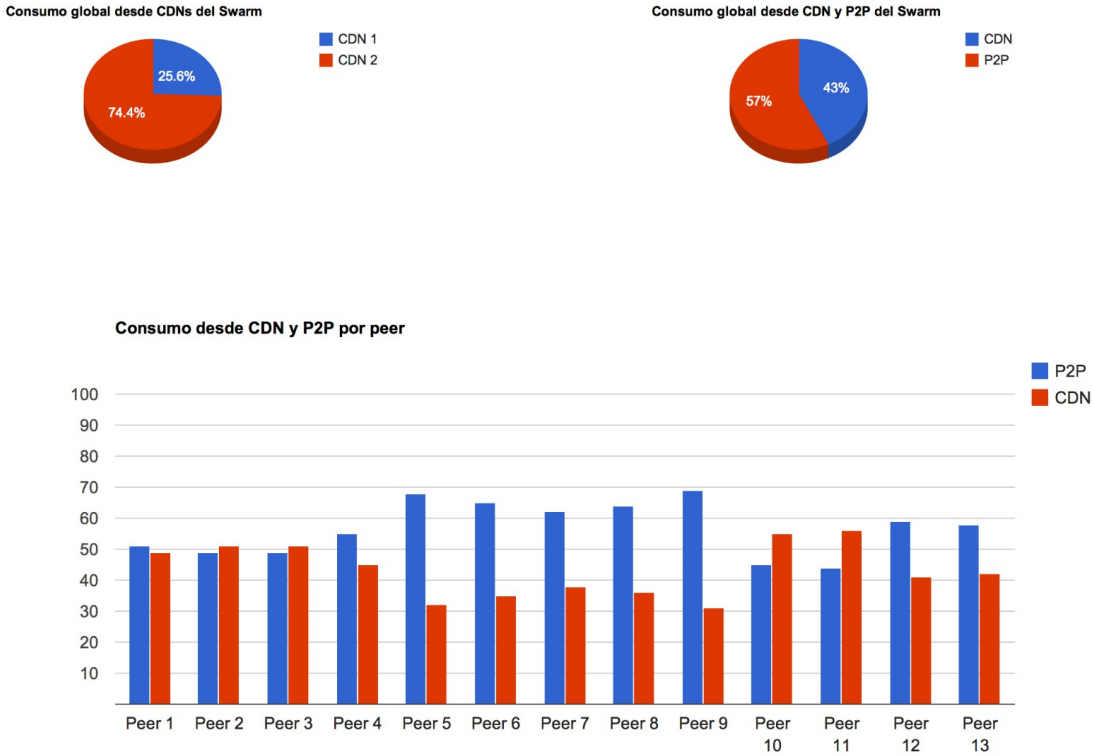

Figura 31: Caso B. Ráfagas de cuatro peers. Dos ráfagas de cuatro peers y una ráfaga de cinco peers, lanzadas una vez cada cuatro minutos.

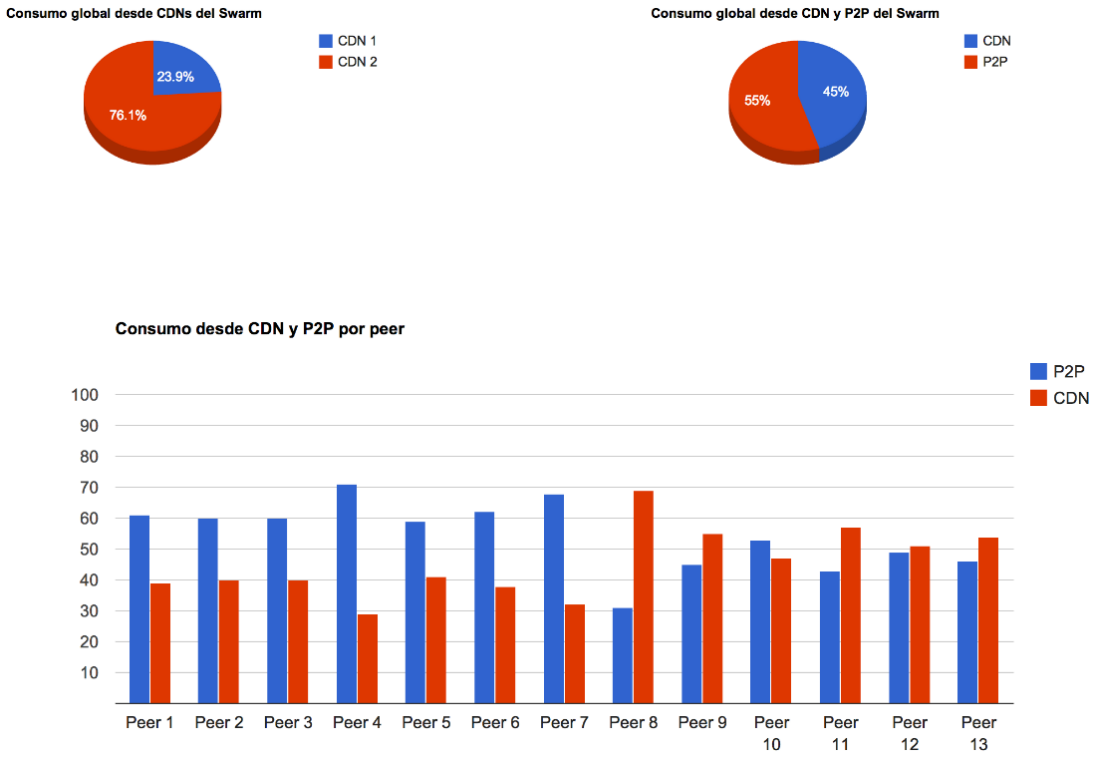

Figura 32: Caso C. Una ráfaga de seis peers y otra ráfaga de siete peers, una cada cinco minutos.

Si bien en algunos casos el porcentaje de P2P realizado disminuye, nunca baja del 43%, que sigue siendo un valor aceptable. Así mismo, también siguen apareciendo casos en que el porcentaje de P2P se eleva hasta un 60%.

Para realizar el análisis también se debe tomar en cuenta el número máximo con el que los *peers* pueden hablar simultáneamente, que es cuatro. Al ingresar en ráfagas, pueden existir casos no deseables, en los que se deje afuera del conjunto de interacción a unos pocos *peers*. Por ejemplo, si ingresan cinco *peers* al mismo tiempo, podría llegar a pasar que sólo cuatro formen un canal de intercambio, dejando al quinto *peer* de lado. En las gráficas listadas no parece haber ningún caso de ese estilo, probablemente por las técnicas implementadas para mitigar esos casos, como la utilización de *optimistic unchoking* o las técnicas de petición de *rarest-first* explicadas en la Sección *Error: Reference source not found*.

5.4.2 Estadísticas de calidad

Como se explicó al comienzo del capítulo, la medida de calidad percibida por el usuario que es posible de medir es el Tiempo de espera inicial (tiempo que pasa desde que el usuario ingresa al sitio y el video comienza a reproducir). Otras medidas de espera son imposibles de capturar por problemas de compatibilidad con la API del *tag* de video de HTML5 (el prototipo las implementa pero desafortunadamente el navegador no dispara los eventos relacionados).

Como ya fue mencionado, el tiempo de espera durante la reproducción se determinó como despreciable. Se utilizó como método la observación mientras se ejecutan los casos de prueba.

Las gráficas resultantes referentes a las medidas del Tiempo de espera inicial discriminado por peer pueden verse en el *Anexo 11 Resultados de las pruebas realizadas*. Se comprueba que los valores de los mismos se mantienen prácticamente constantes a lo largo de todas las pruebas, sin mostrar grandes diferencias a través de los diferentes escenarios.

La Figura 33 muestra una gráfica que es el resultado de uno de los casos de prueba, a modo de ejemplo.

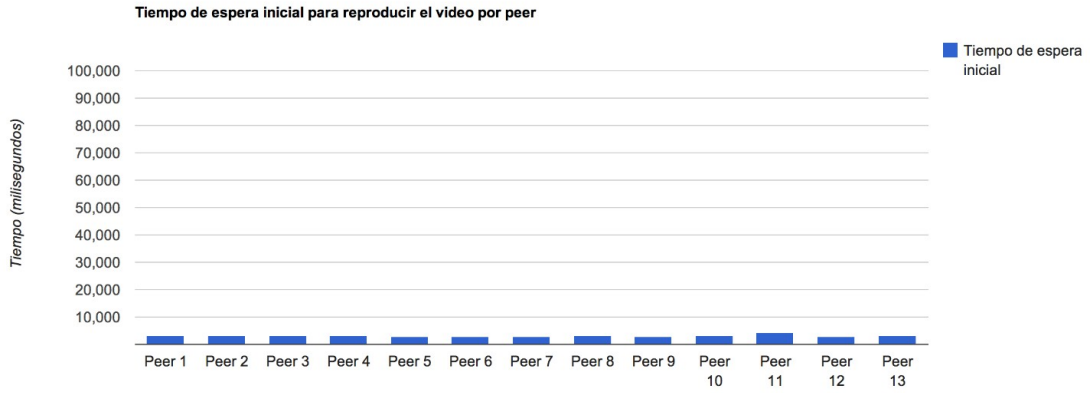

Figura 33: Gráfica de Tiempo de espera inicial por peer.

Según los promedios globales, el Tiempo de espera inicial del sistema para la infraestructura de *testing* utilizada es aproximadamente de 3,5 segundos.

Otro atributo de calidad a destacar es la necesidad de utilizar el sistema con una computadora potente para poder tener una buena experiencia de usuario. Computadoras con poca memoria RAM, poca capacidad de CPU y sin GPU dedicada normalmente se ven saturadas luego de un tiempo utilizando el sistema. Esto se plantea como una de las mejoras posibles a la aplicación en el futuro, ver el *Capítulo 6 Conclusiones*.

5.4.3 Resumen final

Dado que la gran mayoría de los escenarios de prueba ejecutados arrojan resultados muy similares, se realiza un promedio final en base a los promedios de cada uno de las tres ejecuciones de cada caso de prueba.

La Tabla 7 muestra el resultado para el escenario *Porcentaje de seeders en el sistema*. Los valores de la tabla son los resultados de promediar los valores capturados en cada una de las tres ejecuciones de cada caso. De la misma forma, se promediaron los valores del resto de los escenarios de prueba, como se muestra en la Tabla 8 y Tabla 9.

	Porcentaje de *seeders* en el sistema		
	Caso A	**Caso B**	**Caso C**
P2P (%)	60	56	67
CDN (%)	40	44	34
CDN 1 (%)	28	27	30
CDN 2 (%)	72	73	70
Espera Inicial (s)	3.2	3.2	3.4

Tabla 7: Porcentaje de seeders en el sistema. Promedio de las tres ejecuciones para cada caso.

	Distribución de ingreso		
	Caso A	**Caso B**	**Caso C**
P2P (%)	61	60	52
CDN (%)	39	40	48
CDN 1 (%)	24	28	33
CDN 2 (%)	76	72	67
Espera Inicial (s)	3.5	3.2	4.3

Tabla 8: Distribución de ingreso. Promedio de las tres ejecuciones para cada caso.

	Ráfagas de *peers*		
	Caso A	**Caso B**	**Caso C**
P2P (%)	55	57	53
CDN (%)	45	43	47
CDN 1 (%)	24	27	21
CDN 2 (%)	76	73	65
Espera Inicial (s)	3.4	4.3	3.3

Tabla 9: Ráfagas de peers. Promedio de las tres ejecuciones para cada caso.

En base a los valores de las tres tablas anteriores, se realiza un promedio global de las características del sistema, como se ve en la Tabla 10.

Por lo tanto, se concluye que el prototipo logra alcanzar un 58% de P2P contra un 42% de utilización de CDNs, superando así el objetivo inicial de 50% de P2P (ver la Sección *3.2 Requerimientos del prototipo*). Por su parte, de la utilización de ambos CDNs, el CDN1 sólo se utiliza en un 27% con respecto al CDN2, que es utilizado en el restante 72% de las ocasiones. En lo que respecta a la espera inicial, se concluye que un usuario tiene promedialmente unos 3,5 segundos de espera desde que accede al sitio hasta que el video empieza a reproducirse.

	Promedio Total
P2P (%)	58
CDN (%)	42
CDN 1 (%)	27
CDN 2 (%)	72
Espera Inicial (s)	3.5

Tabla 10: Promedios globales de desempeño del sistema.

6 Conclusiones

El consumo de contenido *multimedia* a través de Internet ha crecido enormemente en los últimos años, por lo que existe una gran cantidad de contenido disponible desde diversos proveedores. Entre las modalidades de transmisión de video, el *broadcasting* y el VoD son los servicios con mayor proyección de crecimiento.

El *streaming multimedia* en Internet ofrece importantes desafíos tecnológicos. Uno de los enfoques más utilizados para afrontar estos desafíos consiste en el uso de arquitecturas cliente/servidor. Sin embargo, existen variantes orientadas al P2P que ayudan a realizar soluciones más escalables con infraestructura menos potente y menos costosa.

La IETF ha mostrado interés en el desarrollo de estos protocolos y se encuentra elaborando un estándar de PPSP. Varios proveedores de tecnología se encuentran actualmente desarrollando productos que lo implementan. A su vez, los fabricantes de navegadores *web* encabezados por Google, se encuentran desarrollando la tecnología WebRTC, que permite implementar una red P2P de contenido *multimedia*.

El presente trabajo pretendió brindar un pantallazo general respecto al estado del arte del *streaming multimedia* distribuido en la *web* para redes P2P. Se realizó un amplio estudio de los desarrollos tecnológicos presentes hoy en día, tales como GoalBit, PeerCDN, Peer5, pero enfocándose particularmente en el uso de la tecnología WebRTC (ver el *Capítulo 2 Estado Del Arte*). Esta tecnología aún está en proceso de maduración, ya que se encuentra en etapa avanzada de borrador, pero que aún no es implementada ni soportada por varios navegadores *web*.

Durante el transcurso de esta tesis, se ha investigado y analizado también la viabilidad de elaborar un prototipo que permitiera, utilizando la tecnología WebRTC, implementar un sistema P2P de *streaming* de video que basara sus algoritmos en los de GoalBit y BitTorrent.

Según el *Capítulo 5 Pruebas realizadas*, se puede concluir que el prototipo desarrollado cumplió con sus objetivos principales y requerimientos vistos en la Sección *3.2 Requerimientos del prototipo*:

- Se logró simular un sistema P2P realista en la *web*, sin requerir de la instalación de ningún tipo de *plugin* ni de librerías externas. Demostrando de esta manera, la viabilidad de utilizar WebRTC como tecnología base para las comunicaciones P2P en un sistema de estas características.
- Se utilizaron los medios de comunicación de datos *data channels* y *web sockets*, como también la API Media Source para realizar *streaming* en forma dinámica hacia el *tag* <video>.
- Se llegó a un porcentaje mayor de 50% de P2P en la generalidad, reduciendo lo más posible el ancho de banda y la latencia.
- Se logró construir el sistema de tal forma que la red P2P es una red justa, donde las descargas entre *peers* se dan de manera equitativa.
- Se minimizó el uso del CDN más caro, dirigiendo en promedio sólo un 27% de las peticiones a CDN, hacia él, logrando reducir los costos asociados a su utilización.

- Se consiguió una calidad aceptable a lo largo de la reproducción, con tiempos de espera por *buffering* casi despreciables y un tiempo de espera inicial menor a 3 segundos y medio, lo que genera una experiencia de usuario aceptable.

La implementación del prototipo aporta mucho valor al área de *streaming* en la *web*. Son pocos productos los que evolucionaron (o planean evolucionar) de una arquitectura cliente/servidor a una arquitectura P2P. Tal es el caso de Netflix que pretende comenzar a incursionar en el área P2P[46]. Además, no se encuentra ningún tipo de proyecto como el planteado cuyo código sea totalmente abierto.

6.1 Dificultades y obstáculos

A lo largo del proyecto se encontraron varios obstáculos que fueron cruciales al momento de tomar determinadas decisiones. En una primera etapa del trabajo, fue necesaria la elaboración de varios prototipos de prueba que permitieran evaluar la factibilidad de desarrollar el prototipo utilizando las tecnologías de punta de HTML5. Entre varios ejemplos, se debió implementar pequeñas aplicaciones que permitieran:

- Comunicar *peers* entre sí en forma directa a partir del uso de *data channels* de WebRTC.
- Comunicar *peers* con los servidores a partir del uso de *web sockets*.
- Reproducir un video en un *tag* <video> de HTML5 a partir de contenido multimedia pedido en forma dinámica a otra fuente, utilizando la API Media Source.
- Cargar a los *peers* con la mayor parte de la lógica de procesamiento de forma tal de contar con servidores más tontos y clientes más inteligentes.

Los ejemplos anteriores se desempeñaron de buena forma para un número razonable de *peers*. A pesar de esto, a partir del uso de estas tecnologías, surgieron restricciones tales como:

- Uso exclusivo del navegador Chrome en su versión 36.0.1985.125. Esto se debe a que los *data channels* de WebRTC se encuentran implementados casi en su totalidad únicamente para este navegador. A su vez, la API Media Source solamente se encuentra implementada en su totalidad en Chrome.
- Chrome tiene una fuerte restricción sobre el formato de contenedor de video soportado nativamente para el *tag* <video>. Este navegador soporta únicamente el formato WebM de video, el cual es abierto y fue específicamente diseñado para su reproducción en un *tag* <video>. Esto se debe a que Google es el *sponsor* de WebM, queriendo instaurar un estándar de formato de video en la *web*.
- JavaScript consiste en un lenguaje que se sustenta en la ejecución asíncrona de su código mediante el uso de funciones de *callback*, y se caracteriza por ser mono-hebra. Consecuentemente, no existe concurrencia, pero si puede simularse mediante el uso de *timeouts*. Por ello, se tuvo especial cuidado al programar rutinas que pudieran causar condiciones de carrera, como también en no dejar eventos colgados.
- La implementación actual de los *data channels* de Chrome soporta el intercambio de datos de tipo *String* de JavaScript y binarios. Consecuentemente, se tuvo que tener especial atención al momento de codificar y decodificar los datos enviados por este

[46] *Netflix considers P2P-powered streaming technology*, https://torrentfreak.com. Abril de 2014.

medio. Por esta razón se optó por utilizar la librería peerJS (ver la Sección *4.6.6 Peerjs*) que se encarga de codificar y decodificar los mensajes enviados en formato UTF, en función del tamaño del mensaje.

- Las aplicaciones *web* pueden requerir del uso de mucha memoria RAM, razón por la cual se debió tener especial atención en no dejar memoria colgada, ser eficientes en la codificación de los algoritmos P2P, e intentar utilizar lo menos posible las operaciones de acceso al DOM que son las más costosas.

6.2 Posibles extensiones y mejoras

Si bien fue implementado un prototipo que soluciona el problema VoD utilizando tecnologías *web* de punta, resultaría de gran interés implementar un sistema similar que soporte *live streaming*. Lamentablemente, no se contó con el tiempo ni la infraestructura necesaria como para poder encarar este problema. No hay lugar a dudas de que sería un aporte aún mayor poder incursionar en esta área.

Por otro lado, se notó que el desempeño del prototipo no fue el ideal en cuanto a consumo de recursos. Las máquinas MacBook Pro utilizadas componen parte del conjunto de computadoras personales más potentes del mercado. Sin embargo, debieron dedicarse completamente a correr el prototipo en la etapa de *testing*, dado que el consumo de CPU fue superior al esperado. Sería ideal dedicar tiempo y recursos al estudio de las estructuras de datos utilizados y a la implementación de los algoritmos de P2P, de forma tal de poder optimizar el código y explotar las potencialidades brindadas por el lenguaje JavaScript. Claramente, el consumo de recursos del prototipo consiste en uno de los puntos más débiles, y debería ser estudiado en mayor detalle realizando pruebas de *performance* sobre el mismo.

Finalmente, la seguridad y robustez del sistema debiera ser revisada con más detalle. Si bien los *data channels* de WebRTC envían y reciben la información encriptada, no se le brindó la suficiente atención a algunos aspectos de seguridad relacionados tales como:

- Detectar cuándo un *peer* malicioso intenta distribuir contenido falso en el *streaming*.
- Potenciales ataques a los *data channels* y *web sockets*, debido a tratarse de tecnologías aún en etapa de desarrollo.
- Autenticación de los peer con el *tracker*, tal como es realizado por el protocolo PPSP (ver el *Anexo 2.3 Protocolo del Tracker*).

Bibliografía

1. Mark Pilgrim. *Dive Into Html5*. First Edition. O'Reilly. Agosto de 2010.
2. Bram Cohen. *BitTorrent Protocol Specification versión 1.0*. Marzo de 2013.
3. GoalBit Solutions. http://goalbit-solutions.com/. Marzo de 2013.
4. María Elisa Bertinat, Franco Robledo Amoza, Daniel De Vera, Pablo Rodríguez Bocca, Gerardo Rubino, Darío Padula, Pablo Romero. *GoalBit: The First Free and Peer-to-Peer Streaming Protocol*. Francia, 17 de Junio de 2009.
5. *Survey Of P2P Streaming Applications - draft-ietf-ppsp-survey-06*. Draft IETF, 8 de Enero de 2014.
6. *HTTP Live Streaming - draft-pantos-http-live-streaming-11*. Draft IETF, 16 de Abril de 2013.
7. Apple Inc. *HTTP Live Streaming Overview*. https://developer.apple.com/library/ios/documentation/networkinginternet/conceptual/streamingmediaguide/StreamingMediaGuide.pdf. Mayo de 2013.
8. Tsahi Levent-Lev. *Peer5 and WebRTC: Talking Data Channel With Hadar Weiss*. Foro BlogGeek.Me, 12 de Setiembre de 2013.
9. Daniel Osvaldo De Vera Rodríguez. *Goalbit: la primer red P2P de distribución de video en vivo de código abierto y gratuita*. Tesis para Magister en Informática. FIng UDeLaR, Montevideo, Uruguay, 18 de Octubre de 2010.
10. *HTML5 Specification*. http://www.w3.org/TR/html5/. W3C Candidate Recommendation, Mayo de 2014.
11. *HTML <audio> tag*. http://www.w3.org/wiki/HTML/Elements/audio. W3C, Abril de 2013.
12. *HTML <video> tag*. http://www.w3.org/wiki/HTML/Elements/video. W3C, Setiembre de 2011.
13. *Can I Use the HTML5 video element?*. http://caniuse.com/video. Mayo de 2013.
14. How to enconde and implement for HTML5 Video. http://www.encoding.com/HTML5_Video_Format_Conversion_Support/. Diciembre de 2013.
15. *The WebSocket API*. http://dev.w3.org/html5/websockets/. W3C Draft, 28 de Febrero de 2014.
16. Vanessa Wang, Frank Salim, Peter Moskovits. *The Definitive Guide To HTML5 WebSocket*. Appress, Febrero de 2013.
17. WebM Media Container. http://www.webmproject.org/. Mayo de 2013.
18. HTML5 WebSockets, http://www.websocket.org/. Mayo de 2013.
19. *Media Source Extensions*. W3C Draft. https://dvcs.w3.org/hg/html-media/raw-file/tip/media-source/media-source.html. 1 de Abril de 2014.
20. *WebRTC 1.0, Real-Time Communication Between Browsers*. W3C Draft. http://dev.w3.org/2011/webrtc/editor/archives/20140410/webrtc.html. 10 de Abril de 2014.
21. *WebRTC official page*. http://www.webrtc.org/. Mayo de 2013.
22. Sam Dutton. *Getting Started with WebRTC*. Foro HTML5Rocks, 23 de Julio de 2012.
23. Sam Dutton. *WebRTC in the real world: STUN, TURN and Signaling*. Foro HTML5Rocks, 4 de Noviembre de 2013.
24. Sam Dutton, Justin Uberti. *Real-time communication with WebRTC: Google I/O 2013*. Conferencia Google I/O San Francisco, Estados Unidos, Mayo de 2013.

25. Daniel C. Burnett, Alan B. Johnston. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web - Third Edition*. Digital Codex LLC, Marzo de 2014.
26. Eric Bidelman. *Capturing Audio & Video in HTML5*. Foro HTML5 Rocks, 22 de Febrero de 2012.
27. *Interactive Connectivity Establishment*. RFC 5245. http://tools.ietf.org/rfc/rfc5245.txt. Abril de 2010.
28. *WebRTC Statistics and Metrics*. http://webrtcstats.com/. Enero de 2014.
29. *WebRTC Data Channel Protocol - draft-jesup-rtcweb-data-protocol-03*. Draft IETF. http://tools.ietf.org/html/draft-jesup-rtcweb-data-protocol-03.txt. 26 de Marzo de 2013.
30. *PPSP Tracker Protocol-Base Protocol (PPSP-TP/1.0) - draft-ietf-ppsp-base-tracker-protocol-03.* Draft IETF. http://tools.ietf.org/id/draft-ietf-ppsp-base-tracker-protocol-03.txt. Julio de 2014.
31. *Peer-to-Peer Streaming Peer Protocol (PPSP) - draft-ietf-ppsp-peer-protocol-06*. Draft IETF. http://tools.ietf.org/id/draft-ietf-ppsp-survey-06.txt. Julio de 2013.
32. *Problem Statement and Requirements of Peer-to-Peer Streaming Protocol (PPSP) - draft-ietf-ppsp-problem-statement-13*. Draft IETF. http://tools.ietf.org/html/draft-ietf-ppsp-problem-statement-13.txt. 20 de Febrero de 2013.
33. *PPSP Tracker Protocol--Extended Protocol - draft-huang-ppsp-extended-tracker-protocol-02*. Draft IETF. http://tools.ietf.org/html/draft-huang-ppsp-extended-tracker-protocol-02.txt. Agosto de 2013.
34. *PPSP Streaming Protocol (PPSP) Requirements - draft-zong-ppsp-reqs-04*. Draft IETF. http://tools.ietf.org/id/draft-ietf-ppsp-survey-04.txt. 7 de Julio de 2010.
35. CISCO. *Cisco Visual Networking Index: Forecast and Methodology, 2013–2018*. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html. 10 de Junio de 2014.
36. *Javascript Session Establishment Protocol – draft-ietf-rtcweb-jsep-06*. Draft IETF. http://tools.ietf.org/id/draft-ietf-rtcweb-jsep-06.txt. 13 de Febrero de 2014.
37. Chin Yong Goh, Hui Shyong Yeo, Hyotaek Lim. *A Comparative Study of Tree-based and Mesh-based Overlay P2P Media Streaming*. Faculty of Computing and Informatics, Multimedia University, Cyberjaya, Malaysia.
38. *WebM Byte Stream Format*. https://dvcs.w3.org/hg/html-media/raw-file/tip/media-source/webm-byte-stream-format.html. W3C Draft. Diciembre de 2013.

Anexos

1 Formatos de contenedor y *codecs*

A continuación se menciona brevemente una lista de los formatos de contenedor más conocidos y utilizados hoy en día.

MPEG-4

Contenedor con extensión .mp4 o .m4v. Los contenedores MPEG-4 se basan en el viejo contenedor .mov de QuickTime de Apple[47].

Flash Video

Contenedor con extensión .flv[48]. Este tipo de contenedores son utilizados por Adobe Flash Player[49], aunque actualmente Flash también reconoce contenedores de tipo MPEG-4.

Ogg

Contenedor con extensión .ogv. Ogg, también conocido como Theora, es un estándar abierto, y libre de patentes. Es soportado por Firefox 3.5+, Chrome 4+ y Opera 10.5+ (ver Tabla 12), navegadores que ofrecen soporte nativo para el mismo y también para el audio Ogg, conocido como Vorbis[50].

WebM

Contenedor con extensión .webm. Consiste en un nuevo formato de contenedor anunciado por Google I/O en 2010, diseñado para su uso exclusivo con el *codec* de video VP8 y el *codec* de audio Vorbis. Según [16] será soportado en forma nativa, sin *plugins* específicos de plataforma en las próximas versiones de Chromium, Firefox y Opera. Adobe también ha anunciado que su próxima versión de Flash soportará este formato de contenedor. Actualmente es soportado por la última versión estable de Chrome y Chrome Canary.

Audio Video InterLeave

Contenedor con extensión .avi. Este formato de contenedor fue creado por Microsoft en 1992. Oficialmente no soporta muchas funcionalidades de los formatos de contenedores de video más recientes; no soporta ningún tipo de meta-datos de video, ni soporta la gran mayoría de *codecs* de video ni de audio. Sin embargo, existen extensiones a este formato que lo hacen muy usado.

A continuación se describe brevemente una lista de los *codecs* de video más relevantes [1]. La gran mayoría de éstos se basan en estándares propuestos por el Moving Pictures Expert Group (MPEG) y otros propietarios.

H.264

Se conoce también como “MPEG-4 Advanced Video Coding” y fue desarrollado por el MPEG Group y estandarizado en 2003. Su objetivo es proveer de un solo *codec* para dispositivos de bajo ancho de banda y bajo CPU (dispositivos móviles) y dispositivos de alto ancho de banda y alto CPU (computadores personales).

[47] *Apple QuickTime Player.* https://www.apple.com/quicktime/. Mayo de 2013
[48] *Adobe Flash Video File Format Specification – Version 10.1.* http://download.macromedia.com/f4v/video_file_format_spec_v10_1.pdf, Mayo de 2013.
[49] *Adobe Flash Player.* http://www.adobe.com/products/flashplayer.html. Mayo de 2013.
[50] Vorbis. http://www.vorbis.com/. Mayo de 2013.

Para lograr esto, el estándar H.264 se divide en perfiles, cada uno de los cuales define un conjunto opcional de características que comercian la complejidad del tamaño del archivo. Los mayores perfiles utilizan más características opcionales, que ofrecen una mejor calidad visual a tamaños de archivos menores, toman más tiempo en codificar, y requieren más poder de CPU para decodificar en tiempo real.

YouTube utiliza H.264 para codificar videos en alta definición para reproducirse con Adobe Flash, como también utiliza H.264 para codificar videos para dispositivos móviles Apple y Android. H.264 es también uno de los *codecs* de video mandatorios por la especificación de Blu-ray. El estándar H.264 se encuentra patentado y su licencia pertenece actualmente a MPEG LA group.

Theora

Theora es un *codec* de video que evolucionó desde el *codec* VP3 desarrollado por la compañía On2, y posteriormente se continuó su desarrollo por parte de Xiph.org Foundation. Theora es un *codec royalty-free* y se encuentra libre de patentes[51].

Los videos codificados con Theora pueden ser embebidos en cualquier formato de contenedor, aunque generalmente se les encuentra en contenedores Ogg. La mayoría de las distribuciones de Linux soportan Theora *out of the box*, y Firefox 3.5 incluye soporte nativo para el mismo.

VP8

VP8 es otro *codec* de video originado por On2[52]. Técnicamente es similar en calidad al H.264, con mucho potencial para futuras mejoras.

En 2010 Google adquirió On2 y publicó la especificación de este *codec* de video y también ejemplos del codificador y decodificador como código abierto. Como parte de esto, Google también hizo públicas las patentes que poseía On2 de VP8 y les dió licencia de *royalty-free*.

[51] *Theora.* http://www.theora.org/. Mayo de 2013.
[52] *On2 Techonologies*. http://www.on2.com/. Mayo de 2013.

2 Protocolo PPSP

2.1 Motivación

Esta sección se basa fuertemente en [5][32][34]. Como se mencionó en el *Capítulo 1 Introducción*, el tráfico de *streaming* se encuentra dentro del tipo de tráfico más grande y rápido de Internet, donde en particular, el *streaming* P2P es el que contribuye sustancialmente. La arquitectura P2P trae como ventaja una mayor escalabilidad y tolerancia a fallos de un solo punto (del servidor). Además, las aplicaciones actuales de *streaming* P2P permiten distribuir programas de *streaming* de video bajo demanda y video en vivo a gran escala a una gran audiencia con un número reducido de servidores. En conjunto con la incorporación de CDNs, el sistema se torna aún más eficiente.

Dado el incremento de la integración del *streaming* P2P en Internet, uno de los grandes problemas surgidos es la falta de un protocolo abierto y estándar de *streaming*. La gran mayoría de los sistemas actuales utilizan protocolos propietarios. Existen varios ejemplos de implementaciones cerradas que han demostrado ser eficientes en su objetivo, pero resultan finalmente en esfuerzos de desarrollo repetitivos y grandes problemas de integración con otros elementos como los CDNs, al intentar posicionar al *streaming* P2P como una solución global.

El borrador del PPSP WG [32] surge como una primera aproximación a encarar este problema. Allí se discute el desarrollo de un protocolo de *streaming* P2P abierto, con el objetivo de estandarizar operaciones de señalización en sistemas de *streaming* P2P para resolver los problemas anteriormente mencionados.

2.2 Requerimientos

El PPSP WG se encuentra diseñando dos protocolos de señalización y control para sistemas de *streaming* P2P para transmitir contenido *multimedia* en vivo con requerimientos de envío que se acoplen en tiempo real.

Existen dos tipos de nodos en este sistema P2P: los *peers* y los *trackers*. Los *peers* son nodos que se encuentran enviando y recibiendo contenido *multimedia* en forma activa, e incluyen *hosts* conectados en forma estática tales como computadoras personales y dispositivos móviles con direcciones IP variables en el tiempo. El conjunto de *peers* que se encuentran participando en una sesión de *streaming* varía con el paso del tiempo, y a dicha agrupación se le conoce como *swarm* (ver glosario en [3]). Los *trackers* son nodos bien conocidos con una conexión estable que mantienen meta-información sobre el contenido de *streaming* y de los *swarms*. Los *trackers* pueden ser organizados en forma centralizada o en forma distribuida.

El protocolo para señalización y control entre *trackers* y *peers* se conoce como PPSP Tracker Protocol (PPSP-TP), y el protocolo para señalización y control para la comunicación entre *peers* se conoce como PPSP Peer Protocol (PPSP-PP). Ambos protocolos permiten a los *peers* recibir datos de *streaming* dentro de las restricciones de tiempo requeridas por elementos de contenido específico. El protocolo del *tracker* maneja la comunicación inicial y el intercambio periódico de meta-información entre *trackers* y *peers*, tal como las listas de *peers* y la información de contenido. El protocolo de *peer* controla la publicación e intercambio de disponibilidad de contenido multimedia entre los *peers*.

El protocolo del *tracker* se diseña como un protocolo de tipo *request*/*response* entre *peers* y *trackers*, y maneja información necesaria para la selección de *peers* adecuados para *streaming* en tiempo real. El protocolo de *peers* se encuentra diseñado como un protocolo chismoso (*gossip-like* protocol en Inglés) con intercambios periódicos de información de los *peers* vecinos y de disponibilidad de *chunks* de contenido. Ambos protocolos corren sobre el protocolo de transporte TCP (o sobre UDP, en caso de que los requerimientos para correr sobre TCP no estén dados), y tienen soporte para realizar NAT traversal.

Un *peer* que se encuentra en búsqueda de un *chunk* de contenido utiliza los protocolos de *tracker* y de *peer* para localizar al *peer* remoto (o *peers* remotos) que puedan proveer dicho *chunk*. Obtener el *chunk* del *peer* remoto involucra algunos intercambios de señalización adicionales a la transferencia de los datos. El PPSP WG se encuentra evaluando los protocolos ya existentes para poder reutilizar o extender sus servicios. Ejemplos de protocolos de señalización a ser considerados son SIP, RTSP y HTTP según [32]. También se consideran los protocolos RTP y HTTP para transferencia de datos *multimedia*.

Los trabajos publicados por el PPSP WG (aún en desarrollo) que contienen la especificación del problema, los requerimientos y los protocolos para el *tracker* y *peer* son los siguientes:

1. Un documento con el planteamiento del problema, que trata de una mirada general al sistema de *streaming* P2P planteado, motivando el deseo por estandarizar los protocolos existentes, y discutir terminologías y conceptos comunes [32].
2. Un documento de requerimientos que detalla las especificaciones funcionales, requerimientos de operación y *performance* de los dos protocolos de PPSP [34].
3. Un documento que contiene un estudio de la arquitectura, y que sumariza las arquitecturas de *streaming* P2P existentes hoy en día, algunas de ellas se estudiaron aplicando ingeniería reversa ya que se tratan de arquitecturas propietarias [5].
4. Una especificación detallada del PPSP Peer Protocol [31].
5. Una especificación detallada del PPSP Tracker Protocol [30].
6. Una guía de usuario que describe cómo los dos protocolos de PPSP y los protocolos existentes de IETF pueden ser combinados para crear un sistema de *streaming* P2P operacional.

2.3 Protocolo del *Tracker*

El PPSP Tracker Protocol (PPSP-TP) es un protocolo de señalización y control de capa de aplicación, que provee comunicación entre *trackers* y *peers* al permitir que un *peer* envíe meta-información a los *trackers*, reporte el estatus del *streaming* y obtenga lista de *peers* de los *trackers*.

La arquitectura de PPSP requiere que los *peers* se puedan comunicar con el *tracker* para poder participar en un *swarm* e intercambiar contenido *multimedia*. Este *tracker* central es utilizado por los *peers* para registrar el contenido existente y su localización. El documento [30] describe el PPSP-TP, y cómo el mismo satisface los requerimientos para el PPSP de forma tal de mantener el estándar.

En dicho documento se explica la forma de operación del protocolo, en particular se explican los procedimientos por los cuales un *peer* procede a recibir el *streaming* de un

contenido seleccionado, y también cómo un *peer* procede a compartir un contenido determinado con otros *peers*.

El Tracker Protocol es modelado como un protocolo *request*/*response*, basado en texto y que utiliza el conjunto de caracteres UTF-8. Ambos tipos de mensaje utilizan los mismos cabezales Status-Code y Reason-Phrase del protocolo HTTP.

Los mensajes de tipo *request* son enviados desde los *peers* al *tracker* y pueden ser de dos tipos: CONNECT (para registrar y/o solicitar acciones por contenido de *streaming* a un *swarm*) y STAT_REPORT (enviado periódicamente para informar al *tracker* sobre el estatus del *peer* y brindar información estadística).

Los mensajes de tipo *response* son enviados desde el *tracker* hacia los *peers* y pueden ser de dos tipos: SUCCESSFULL y AUTHENTICATION_REQUIRED. El cuerpo de los mensajes d e *request* y *response* se corresponde con un documento XML, opcionalmente representado en forma binaria, y su especificación detallada se encuentra en [30].

En lo que refiere a la seguridad, dado que los sistemas de *streaming* P2P se ven sujetos a ataques por parte de *peers* y/o *trackers* maliciosos, se considera un nivel de protección suficiente para mantener ciertas propiedades de seguridad durante la comunicación entre *trackers* y *peers*. En particular, se provee de un sistema de autenticación para los *peers* a través de un servidor de autenticación, y un esquema de verificación de integridad de los mensajes para detectar contenido malicioso.

2.4 Protocolo de los *peers*

El PPSP Peer Protocol (PPSP-PP) es un protocolo de señalización de capa de transporte que permite diseminar el mismo contenido a un grupo de *peers* interesados en el *streaming*. PPSP-PP soporta *streaming* para contenido de audio/video bajo demanda y en vivo. Se basa en el paradigma P2P en el cual los clientes que consumen el contenido son puestos en igualdad de condiciones con los servidores que proveen el contenido inicialmente, para crear un sistema donde todos puedan proveer subida de ancho de banda en forma potencial.

Este protocolo se diseña para proveer un tiempo corto para comenzar la reproducción del lado del usuario, y para prevenir disrupciones de los *streams* por parte de *peers* maliciosos. El contenido de datos es identificado por un *hash* criptográfico que se encuentra en el *hash* raíz de un Merkle Hash Tree calculado en forma recursiva a partir del contenido (ver [31]). Este árbol de *hashes* permite a cada *peer* detectar cuándo un *peer* malicioso intenta distribuir contenido falso en el *streaming* y puede ser utilizado tanto para contenido estático como en vivo.

En [31] se describen los tipos de mensajes que utiliza el protocolo, los esquemas de direccionamiento de *chunks*, los mecanismos de protección para la integridad de contenido, la encapsulación de los mensajes en datagramas UDP, y la extensibilidad del protocolo.

En lo que refiere a los mensajes del protocolo, en general no se utilizan códigos de error como en el Tracker Protocol o respuestas de error; la ausencia de una respuesta indica el error de por sí. Los tipos de mensajes que se definen en el protocolo son: HANDSHAKE, HAVE, DATA, ACK, INTEGRITY, SIGNED_INTEGRITY, REQUEST, CANCEL, CHOKE,

UNCHOKE, PEX_REQ y PEX_RES. La especificación y uso operacional de cada uno de ellos se encuentra ampliamente descrita en [31].

3 Redes comerciales de streaming P2P

3.1 Octoshape

Octoshape[53] es una plataforma de *streaming* multimedia P2P mallada, conocida por haber sido utilizada por la cadena CNN para hacer *broadcast* de su contenido en vivo. Octoshape llegó a un pico de más de un millón de espectadores simultáneos y proveyó de tecnologías innovadoras de envío de contenido, como por ejemplo, transporte resistente a pérdidas y optimización en adaptación de los caminos de entrega de contenido [5].

Como se puede apreciar en la Figura 34, la arquitectura de Octoshape no cuenta con una entidad *tracker*, y por lo tanto no cuenta con un protocolo de *tracker* (los *peers* que se encuentran mirando un contenido por el mismo canal son insertados como meta-datos del mismo contenido).

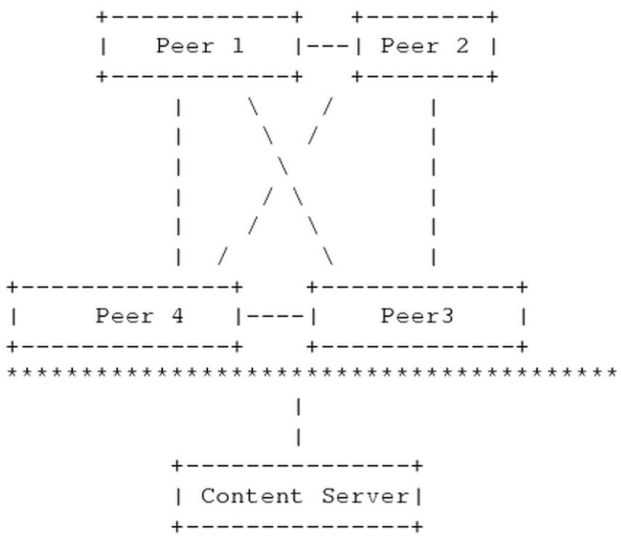

Figura 34. Arquitectura Básica de Octoshape [3]

En lo que refiere al protocolo de los *peers*, tan pronto como un *peer* se une a un canal, le notifica a todos los otros *peers* acerca de su presencia y de tal forma que cada *peer* mantiene una suerte de registro con la información necesaria para contactar a los otros *peers* que se encuentran observando el mismo canal.

Respecto a la estrategia de distribución de datos, la solución planteada por Octoshape consiste en partir el flujo original en un número fijo K de flujo de datos menor, pero con la restricción de que un número N > K de flujos de datos se construya efectivamente. De esta forma, un *peer* que recibe K de los N flujos de datos disponibles se encuentre apto para reproducir el flujo original.

A modo de mitigar el impacto de la pérdida de un *peer*, además de la lista de referencia a otros *peers* de un canal, los *peers* mantienen una lista de *peers* prontos para ser contactados (*stand by list*, en Inglés). Esta última lista es utilizada para que el *peer* sepa qué *peer* referenciar ante la pérdida de otro *peer* del cual se encuentre consumiendo contenido actualmente.

[53] *Octoshape*. http://www.octoshape.com/. Mayo de 2013.

3.2 PPLive

PPLive[54] es el *software* de televisión P2P (P2PTV) más utilizado en China y en Asia en general. Se estima que la cantidad de usuarios que lo utilizan en forma concurrente para *streaming* en un solo canal se aproxima a las 200.000 según [4]. Este *software* tiene sus orígenes en un proyecto de estudiantes de la Universidad de Ciencia y Tecnología Huazhong en China. Los protocolos PPLive son propietarios y su código no es abierto [5].

Para el protocolo del *tracker* de PPLive, los peers obtienen en primer lugar una lista de canales de un servidor de lista de canales. Luego, el *peer* selecciona un canal y pregunta al *tracker* por la lista de *peers* asociada al canal mencionado. En lo que refiere al protocolo de los *peers*, un *peer* contacta a los otros *peers* de la lista para obtener listas de *peers* adicionales para ser integradas con la lista original recibida del *tracker*. Esto persigue el objetivo de construir y mantener un *overlay* de mallas para la gestión de *peers* y envío de datos.

Para el caso de video bajo demanda, los *peers* se intercambian mensajes de tipo *buffered map*. Este tipo de mensajes indica qué *chunks* se encuentran en propiedad de un cierto *peer* y pueden compartirse, e incluye el *offset* (ID del primer *chunk*), el largo del *buffer map*, y una cadena de ceros y unos que indican qué *chunks* se encuentran disponibles (comenzando por el *chunk* designado por el *offset*). El intercambio de datos se realiza sobre el protocolo de transporte UDP.

La política de descarga de PPLive consiste en los siguientes tres pasos, según la ingeniería inversa explicada en [3]:

1. El top diez de los *peers* contribuyen a la mayor parte del tráfico de descarga. PPLive obtiene el video de un número reducido de *peers* en cualquier momento, y se intercambian periódicamente de un *peer* a otro para descargar.
2. PPLive puede enviar múltiples solicitudes para distintos *chunks* hacia un *peer* en un solo momento.
3. PPLive aplica una política de programación de descarga según la cual se le da mayor prioridad a los *chunks* más raros y a los *chunks* con contenido más cercano al *deadline* de reproducción.

3.3 New Coolstreaming

Coolstreaming[55] fue lanzado a principios de 2004 con una estructura basada en mallas, y representó en su momento una de las aplicaciones de *streaming* en vivo P2P más exitosas. Sin embargo, sufría de una baja *performance* en *delay* y de un alto *overhead* asociado con cada transmisión de *chunks* de video. Con el objetivo de sobreponerse a estas limitaciones surge New Coolstreaming, que adopta una estructura de overlay híbrida basada en mallas y un mecanismo de envío de contenido híbrido de tipo *pull-push* [5].

[54] *PPLive*. http://www.pptv.com/. Mayo de 2013
[55] *Coolstreaming*. http://www.coolstreaming.us/. Mayo de 2013.

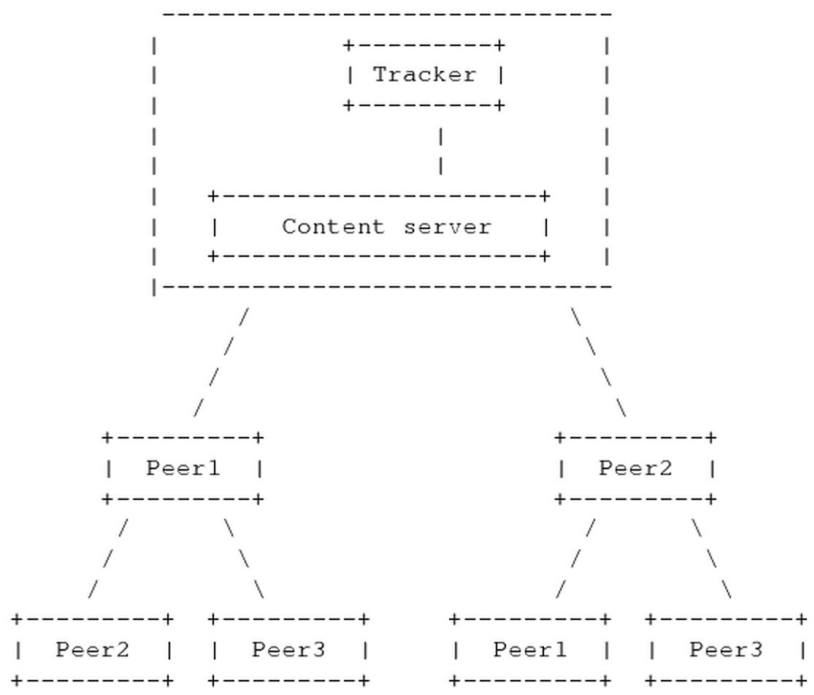

Figura 35. Arquitectura Básica de New Coolstreaming [3]

4 GoalBit

4.1 Arquitectura

La arquitectura de GoalBit se encuentra conformada por cuatro componentes:

1. El *Broadcaster* es quien recibe el contenido a distribuir desde su origen, lo procesa y lo introduce en la red P2P. Por el rol que desarrolla en la red, constituye un componente crítico, siendo vital que permanezca activo. El mismo no distribuye directamente el contenido a los consumidores, o sea, los *peers*.
2. Los *Super-peers* son *peers* con alta disponibilidad y gran capacidad de ancho de banda, que reciben el contenido del *broadcaster* y lo distribuyen hacia los *peers*.
3. Los *Peers* son quienes reproducen el contenido asociado al flujo de video al cual se encuentran conectados. Representan un componente dinámico de la red ya que los nodos suelen conectarse y desconectarse de manera frecuente.
4. El *tracker* es quien posee la información de todos los nodos, cualquiera sea su tipo, que se encuentran conectados en la red. Lleva un registro de los nodos que se encuentran relacionados con cada contenido. Cumple un rol central en la distribución de los contenidos, devolviendo listados de *peers* asociados a cierto contenidos a cada *peer* que lo solicite.

4.2 GoalBit Transport Protocol

Este protocolo es una extensión del protocolo BitTorrent optimizada para la transferencia de video en vivo según se especifica en [12].

Utiliza una ventana deslizante donde se alojan las piezas descargadas, y cada *peer* solo puede compartir o descargar piezas que se encuentren dentro de ésta. Cada *peer* va reproduciendo secuencialmente en orden las piezas de esta ventana, llevando un índice de ejecución (siguiente pieza a consumir), *buffer* activo (secuencia de piezas ya descargadas a partir del índice de ejecución) y el índice de *buffer* activo (la última pieza a reproducir del buffer activo).

Así como en BitTorrent se tienen los archivos de extensión .torrent, en GoalBit se tienen los archivos de extensión .goalbit como descriptores de contenidos. Alguna información importante que contiene este archivo es por ejemplo el identificador del contenido, el tamaño de las piezas, la URL del *tracker* y el *bitrate* del flujo, entre otros.

4.2.1 Comunicación entre los componentes

En primera instancia, un usuario de la plataforma GoalBit deberá obtener un archivo de extensión .goalbit desde alguna fuente, como por ejemplo un sitio *web*. Luego, utilizando la URL del *tracker* disponible en este archivo para comunicarse con él y unirse a la red GoalBit.

Tracker - Peer

Cuando un *peer* ingresa a la red inicia una comunicación con el *tracker* en primera instancia, quien registra al nuevo *peer. E*n su respuesta, le devuelve una lista con otros *peers* a comunicarse, un índice desde dónde comenzar a reproducir y un tipo de *peer*. Para este fin, el *tracker* y el *peer* se comunican utilizando un mensaje de tipo *announce*.

Posteriormente, cada *peer* tiene comunicaciones periódicas con el *tracker.* Por defecto cada treinta segundos le envía al *tracker* su índice de *buffer* activo actual, que consiste de información sobre la calidad del servicio y eventualmente del número de nuevos *peers* que desea solicitar. Estos intercambios también se basan en el mensaje de tipo *announce*.

Peer - Peer

El diálogo entre *peers* es de mayor complejidad que entre el *tracker* y los mismos. Existen dos tipos de interacción entre *peers*, intercambio de información contextual e intercambio de piezas.

Dentro del intercambio de información contextual se utilizan los siguientes tipos de mensajes:

1. HANDSHAKE - Este tipo de mensaje se utiliza cuando dos *peers* comienzan un vínculo. En él, estos intercambian mutuamente información como el tipo de cada uno y el largo y la base de sus ventanas deslizantes.
2. BITFIELD - Se intercambian mutuamente luego del HANDSHAKE y por única vez entre los *peers* para saber el estado de la ventana deslizante entre *peers* vinculados (cuáles piezas posee el *peer* y cuáles no).
3. HAVE - Este mensaje es enviado por un *peer* a todos sus *peers* vinculados (para el flujo respectivo a la pieza) cuando el primero recibe un nuevo *chunk* y el destinatario del mensaje tiene la pieza recibida en su ventana deslizante y no la posee aún.
4. WINDOW UPDATE - Se utiliza entre los *peers* para actualizar la posición de la ventana deslizante cuando ésta es desplazada. En el mensaje solamente se manda la nueva base de la ventana.
5. KEEP-ALIVE – Se envían mensajes de este tipo para evitar que la conexión sea terminada, cuando no hay intercambio de mensaje entre *peers* por un determinado tiempo.

En cambio, en el intercambio de *chunks*, los mensajes son los siguientes:

1. INTERESTED - Es enviado de un *peer* a otro cuando este último dispone de un *chunk* en su ventana, que el primero no tiene, y por ende pretende recibirlo.
2. NOT INTERESTED - Si un *peer* había mostrado interés en un *chunk* previamente mediante el mensaje INTERESTED y pierde interés en éste (por haberlo conseguido de otro *peer*).
3. CHOKE - Se usa para deshabilitar a otro *peer* a descargar *chunks* del primero.
4. UNCHOKE - Se usa para habilitar a otro *peer* a descargar *chunks* del primero.
5. REQUEST - Un *peer* habiendo enviado un mensaje tipo INTERESTED y habiendo recibido un mensaje tipo UNCHOKE por parte de otro *peer*, puede pedir una pieza usando un mensaje REQUEST con el identificador de la pieza a solicitar, junto a otra información. Los *chunks* se componen de *slices* y los intercambios son en base a *slices* de estas y no las piezas completas.
6. CANCEL - Se utiliza este mensaje para cancelar el pedido de un *slice*.
7. PIECE - Este mensaje contiene un *slice* del *chunk* conjuntamente con información de identificación de este.
8. DONT HAVE - Este tipo de mensaje es usado por los *seeders* para informar a un *peer* que un *chunk* no ha sido generada aún. Esto puede suceder cuando un *peer* consume *chunks* más rápido de la velocidad con que se generan en el *broadcaster.*

4.2.2 Estrategias del protocolo

El protocolo GoalBit utiliza diferentes estrategias para atacar distintos problemas propios de cualquier red P2P. En las siguientes sub-secciones se describen algunos de estos problemas y los diferentes algoritmos o estrategias usadas para resolverlos.

4.2.2.1 Selección de *Peers*

GoalBit utiliza dos estrategias con respecto a la selección de *peers*. La predominante se llama *tit-for-tat*. Esta consiste en que, dado un *peer*, este le envía *chunks* solo a una cantidad dada de *peers* elegidos según quién le ha dado más *chunks* a él. A su vez, también se usa una política llamada *optimistic unchoking*, que es la que permite la incorporación de nuevos *peers* a la red ya que en esta política no se toma en cuenta la generosidad de los *peers*.

En GoalBit, ambas políticas o estrategias se utilizan paralelamente de la siguiente manera. Si dividimos el tiempo en intervalos, y en cada iteración se habilitan un número determinado de *peers*, se aplica *optimistic unchoking* para elegir uno de esos *peers* cada un número dado de iteraciones. En el resto se eligen todos los *peers* usando *tit-for-tat*.

4.2.2.2 Selección de *chunks*

En la red de distribución de archivos BitTorrent se utiliza una estrategia llamada *rarest-first* que consiste en pedir el *chunk* que menos se encuentra disponible en la red. Esto permite distribuir uniformemente las piezas en la red y minimizar el riesgo de que los archivos queden incompletos. Por otra parte, en las redes de reproducción de video en vivo, se utilizan estrategias llamadas *greedy*, que consisten en pedir siempre la siguiente pieza faltante en la línea de reproducción.

Para la reproducción de video en vivo ninguna es totalmente eficiente. Según [9], usar puramente *rarest-first* impacta en los tiempos de *buffering* inicial, y usar puramente *greedy* genera baja continuidad. Por ende, GoalBit utiliza una estrategia híbrida, manteniendo tres rangos dentro de la ventana de reproducción mencionada en el *Anexo 4.2 GoalBit Transport Protocol*. Los rangos son llamados *urgente*, *normal* y *futuro*.

En caso de que falten piezas pertenecientes al rango *urgente*, entonces éstas tienen la prioridad sobre las de otros rangos, y dentro de este se piden primero las que están más cercas de ser reproducidas. Si las piezas *urgentes* ya están descargadas, entonces se piden piezas de los otros rangos en base al muestreo de una variable aleatoria exponencial que da mayor prioridad al rango *normal* que al *futuro*.

4.2.2.3 Armado del *swarm* de un *peer*

La selección de peers para la conformación del *swarm* es un tema de gran importancia en el funcionamiento de una red P2P, influyendo drásticamente en el desempeño de los *peers* en la red P2P. Usualmente se utilizan algunas técnicas para brindarle inteligencia a la selección de los *peers* que conforman un *swarm*. Algunas de ellas son:

- *Proactive network Provider Participation for P2P* (P4P) donde los ISP aportan al *tracker* información del estado de la red, evitando así incluir *peers* en el *swarm* que necesiten usar enlaces altamente congestionados.

- ONO[56] es una técnica que utiliza como criterio de selección la cercanía geográfica entre *peers*, disminuyendo por ende la latencia en las comunicaciones y permitiendo mayor eficiencia en las mismas. Para medir la cercanía geográfica se suelen utilizar consultas DNS a CDNs conocidas, como por ejemplo Google. En base a esa respuesta se concluye que si dos *peers* son enviados al mismo servidor del CDN entonces se encuentran próximos geográficamente.

[56] *Ono P2P*. http://www.aqualab.cs.northwestern.edu/projects/118-ono-reducing-p2p-cross-isp-traffic-while-improving-users-performance. Mayo de 2013.

5 HTTP Live Streaming

5.1 Arquitectura del *streaming* HTTP

HLS permite enviar audio y video en vivo o pregrabado con soporte para encriptado y autenticación, desde un servidor *web* ordinario a cualquier dispositivo que corra iOS 3.0 o mayor, o una computadora con Safari 4.0 instalado o mayor. Conceptualmente, HLS consiste de tres partes fundamentales: un servidor, un distribuidor y clientes.

El servidor se responsabiliza de tomar flujos como entrada y codificarlos digitalmente, encapsularlos y preparar los datos *multimedia* encapsulados para su distribución. A su vez, se encarga de preparar los archivos Playlist. El distribuidor consiste de un servidor *web* estándar que se responsabiliza de aceptar pedidos de los clientes y enviar los datos *multimedia* preparados y los recursos asociados al cliente (pueden utilizarse CDNs para distribuciones de larga escala). Por último, los clientes se responsabilizan de determinar los datos *multimedia* apropiados para el pedido, descargar los recursos y re-ensamblarlos para que los datos *multimedia* sean presentados al usuario como un flujo continuo.

En una configuración típica, un codificador por *hardware* toma el video y audio de entrada y lo codifica en video H.264 y audio AAC, y genera como salida un MPEG-2 Transport Stream, que es posteriormente dividido en una serie de pequeños archivos de datos *multimedia* por parte de un *software* segmentador de flujo. Estos archivos son almacenados en un servidor *web*. El segmentador también crea y mantiene un archivo Playlist conteniendo la lista de archivos de *multimedia*. La URL del archivo Playlist es publicada por el servidor *web*. En la Figura 36 se puede apreciar una configuración básica de la arquitectura cliente-servidor de HLS:

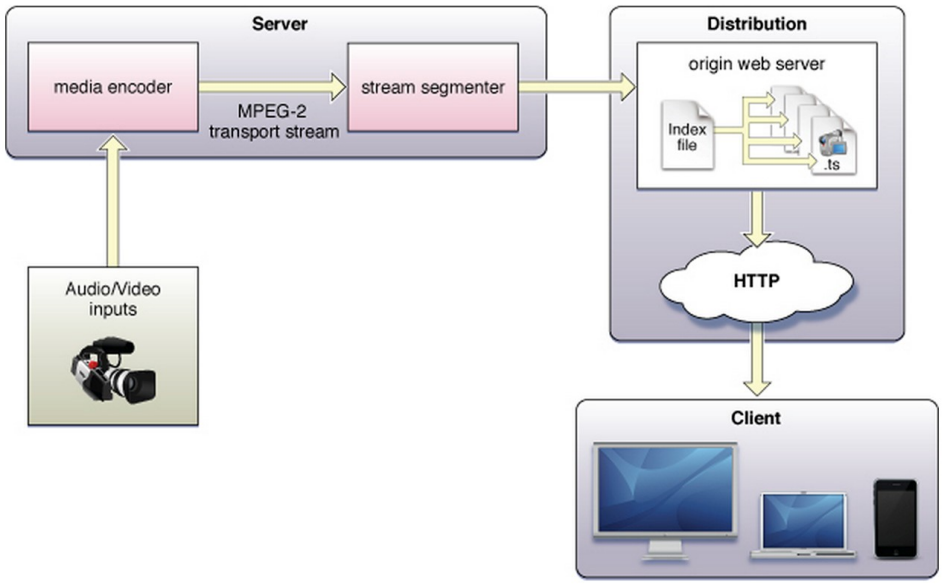

Figura 36. Arquitectura Básica HLS [23]

El *software* segmentador es quien se encarga de crear los archivos Playlist que contienen la lista de archivos *multimedia* y meta-datos adicionales. Este tipo de archivos también se conocen como archivos de índice. La URL del archivo Playlist es accedida por parte de los clientes, quienes luego envían pedidos por los archivos indexados en la secuencia del Playlist.

Los archivos Playlist tienen un formato específico que se encuentra ampliamente detallado en [6]. Un archivo Playlist tiene formato de texto plano y cada una de sus entradas contiene una especificación que puede ser: una ruta absoluta, una ruta relativa a donde se encuentra almacenado el archivo M3U, o una URL.

Los archivos de segmento *multimedia* producidos por el segmentador de flujo consisten en una serie de archivos .ts que contienen segmentos de un MPEG-2 Transport Stream que transportan un video H.264 y audio AAC.

5.2 Interacción Cliente - Servidor

Aquí se describe cómo el servidor (en conjunto con el distribuidor descrito en la parte anterior) genera el archivo Playlist, los segmentos *multimedia* y cómo el cliente debe descargarlos y reproducirlos, según [6].

Entre las variadas tareas que debe realizar el servidor, se encuentran: crear los archivos Playlist y mantenerlos, crear los segmentos *multimedia*, encriptar dichos segmentos y proveer variantes de flujos. Por otro lado, los clientes deben ser capaces de descargar el archivo de Playlist, reproducir el mismo y refrescarlo en forma periódica, determinar el siguiente segmento a descargar, y desencriptar los segmentos *multimedia*.

El servidor en primera instancia debe dividir el flujo en segmentos *multimedia* individuales cuya duración sea menor o igual que la duración de una constante dada. Además, el servidor debe intentar dividir el flujo en puntos que soporten el *decode* efectivo de segmentos *multimedia* individuales. Finalmente debe también crear una URL para cada uno de estos segmentos *multimedia* que permita a los clientes obtener el segmento de datos, e incluir dichas URLs en el archivo de Media Playlist en el orden apropiado para ser reproducidas.

El servidor debe a su vez realizar tareas de mantenimiento del archivo Playlist. Por ejemplo, debe encargarse de agregar y/o remover URLs y directivas al archivo, o inhabilitar al mismo en caso de querer remover una presentación *multimedia* entera.

Los segmentos *multimedia* pueden ser encriptados por el servidor. Para ello, el servidor debe definir una URL que permite a los clientes autorizados obtener un archivo Key que contiene una clave de desencriptado.

En lo que refiere a las variantes de flujos, el servidor debe ser capaz de brindar múltiples archivos de Media Playlist de forma tal de proveer distintos *encodings* de la misma presentación *multimedia*. En caso de ser así, debe proveer de un archivo Master Playlist que contenga una lista de variantes del flujo, de forma de permitir a los clientes elegir entre los distintos *encodings* disponibles.

Por otro lado, una vez que el servidor haya generado los archivos Playlist, el cliente debe obtenerlos desde una URL específica. Si el archivo Playlist obtenido es un Master Playlist, el cliente debe obtener el archivo Media Playlist desde el Master Playlist descargado. El cliente debe verificar que el archivo comience con el *tag* EXTM3U y, en caso de contener el *tag* EXT-X-VERSION, que el mismo especifique la versión de protocolo soportada por el

cliente. A su vez, debe ignorar *tags* que no reconozca y determinar el siguiente segmento *multimedia* a cargar.

Posteriormente, el cliente debe elegir qué segmento *multimedia* reproducir primero del archivo de Media Playlist cuando comienza el *playback*. Cuando esto sucede, los segmentos *multimedia* deben ser reproducidos en el orden en que aparecen en el archivo de Playlist. El cliente podría presentar los datos *multimedia* disponibles en la forma en que quisiera, incluyendo *playback* regular y acceso aleatorio.

El cliente debe refrescar periódicamente el archivo de Media Playlist. Cuando un cliente descarga por primera vez el archivo Playlist o descubre que el mismo tuvo cambios desde la última vez en que fue abierto, el cliente debe esperar por un periodo de tiempo determinado para poder refrescarlo nuevamente. Este tiempo se conoce como tiempo de espera de recarga mínimo, y se mide a partir de que el cliente comienza a descargar el archivo Playlist. El tiempo de espera de recargo mínimo inicial es la duración del último segmento *multimedia* en el Playlist.

6 Lenguaje HTML5

HTML5 [10] consiste en el último estándar del lenguaje HTML y se encuentra bajo un continuo proceso de desarrollo. Uno de sus cometidos fundamentales consiste en abarcar las especificaciones de HTML 4.01[57] (HTML 4), eXtensible HTML 1.1[58] (XHTML) y DOM Level 2 HTML[59] (DOM2HTML).

HTML5 logró generar un estándar de muchas de las funcionalidades que han sido utilizadas por los desarrolladores *web* a lo largo de los últimos años, pero que no habían logrado ser documentadas ni estandarizadas. Tal como sus predecesores, HTML5 es una plataforma independiente implementada dentro de cada navegador *web*. Cabe la pena destacar que HTML5 es una colección de componentes individuales, de las cuales muchas aún se encuentran en proceso de borrador. Por esta razón, varios navegadores soportan/implementan sólo una parte de los componentes de HTML5.

HTML consiste en un lenguaje para crear documentos compuesto por *tags* y texto. Los *user agents* que reconocen el lenguaje HTML, tales como los navegadores *web*, *parsean* el documento y crean el Document Object Model (DOM) a partir del mismo. El DOM consiste en una API que permite acceder a una estructura de datos que representa el documento HTML.

Esta API permite además inspeccionar y modificar la página *web* en forma dinámica dentro del navegador a través del lenguaje JavaScript. A diferencia de las especificaciones anteriores a HTML5, la API provista por el DOM compone una parte fundamental de la especificación de HTML5.

En las siguientes secciones se describen algunos de las propiedades más importantes de HTML, HTML5 y JavaScript que se utilizan para la implementación del proyecto.

6.1 Lenguaje HTML

6.1.1 Evolución del lenguaje HTML

HTML fue considerado en un principio para ser una aplicación de Standard Generalized Markup Language (SGML) y fue formalmente definido como tal por la IETF. Dicha institución creó el HTML Working Group sobre el año 1996, luego del cual las especificaciones del lenguaje HTML pasaron a ser mantenidas por la W3C.

En 2004, una mayoría de los miembros de la W3C votaron en contra de continuar con el lenguaje HTML a favor de utilizar las tecnologías basadas en XML. En respuesta a esto, se creó una comunidad interesada en evolucionar el lenguaje HTML y las tecnologías relacionadas para formar lo que se conoce como Web Hypertext Application Technology Working Group (WHATWG), que continuó con la tarea de evolucionar HTML a lo que se conoce como HTML5.

---

[57] *HTML 4.01 Specification*. http://www.w3.org/TR/REC-html40/. 24 de Diciembre de 1999.
[58] *XHTML 1.1, Module Based XHTML Specification.* http://www.w3.org/TR/xhtml11/. 23 de Noviembre de 2010
[59] *Document Object Model (DOM) Level 2 HTML Specification*, http://www.w3.org/TR/DOM-Level-2-HTML/. 6 de Enero de 2003.

Irónicamente, en 2006 la W3C demostró interés en participar en la evolución de HTML5 y en 2007 comenzó a trabajar en conjunto con WHATWG. La especificación de HTML5 paso a ser adoptada desde ese momento como un punto de partida en el trabajo del nuevo HTML Working Group de W3C.

A pesar de que la especificación se encuentra aún en proceso de borrador y llevará un tiempo más en completarse, gran parte de las secciones de la misma se encuentran ya implementadas en los navegadores y en forma estable. Incluso muchas de las secciones de los borradores han sido implementadas por algunos navegadores. Tal es el caso del navegador Chrome Canary de Google con su implementación de WebRTC y de las APIs getUserMedia y Media Source, como se detallan en la Sección *2.3.5.3.1 Obtención de datos multimedia locales*.

Últimamente se ha puesto en tela de discusión si las especificaciones deberían ser completadas antes de comenzar a ser implementadas o vice-versa. Finalmente se optó porque ambos tienen que transcurrir en paralelo, ya que completar la especificación en primer lugar puede conllevar a problemas de implementación posteriormente, y el *feedback* obtenido a partir de la implementación resulta muy valioso para la especificación.

Además, contar con una implementación ya provista por algunos navegadores permite restringir que la especificación no se vea rediseñada por completo, ya que las aplicaciones que se basan en dichas implementaciones podrían dejar de funcionar a partir de ese momento.

6.1.2 Lenguaje HTML5

Como se mencionó en el *Anexo 6.1.1 Evolución del lenguaje HTML*, HTML5 no consiste de una única entidad, sino que se compone de varios elementos, entre los cuales se incluyen revisiones de las especificaciones de HTML, DOM2, CSS3 y JavaScript. En particular, permite el uso de recursos *multimedia* en una aplicación *web*. Previo a esta tecnología, las experiencias de este tipo se reducían al uso de *plugins* o a aplicaciones de escritorio.

Con HTML5 se permite la creación de aplicaciones y páginas *web* que funcionan casi como aplicaciones de escritorio, lo que conlleva a que todos los usuarios puedan acceder a la misma plataforma *web* al mismo tiempo. Esto evita el viejo problema de que cada usuario tenga que descargar una versión de la aplicación para un dispositivo distinto. Esto se puede apreciar hoy en día con los dispositivos móviles: un usuario puede descargar una aplicación *web* hecha en HTML5 para un dispositivo Android o iPhone indistintamente.

Un aspecto interesante que brinda esta tecnología es que permite crear aplicaciones que funcionan incluso sin encontrarse conectadas a Internet. El truco consiste en la capacidad de almacenar datos y contenido en forma local al navegador, hasta poder sincronizar esta información posteriormente al acceder a Internet.

Otra ventaja referida a este punto consiste en la capacidad de almacenar información en el *cache* del navegador de tal forma de que la misma pueda ser re-accedida incluso cuando la página es recargada. Entre los varios ejemplos de este tipo de aplicación se encuentra la bandeja de entrada de Gmail para dispositivos móviles, la cual almacena información de

borrado de correos electrónicos, clasificación de los mismos, etc a pesar de no contar con acceso a Internet[60].

HTML5 permite también el uso de una gran cantidad de elementos gráficos con los que antes no se contaba, tales como animaciones, juegos y películas, entre otros. Incluso se provee de soporte para efectos gráficos de gran intensidad tales como iluminación y sombras, efectos 3D y gráficos vectoriales.

Los nuevos motores de JavaScript, tales como V8[61] de Google y SpiderMonkey[62] de Firefox son también una ventaja adicional, ya que son lo suficientemente rápidos como para correr estas aplicaciones en tiempo real. Actualmente los navegadores aprovechan las capacidades de la Graphics Processing Unit (GPU) para acelerar las tareas de cómputo, lo que contribuye a la mejora de la experiencia de usuario.

Además de mantener la compatibilidad de los *tags* con las versiones anteriores de HTML, y de mantener la estructura y formato del lenguaje, se adicionan nuevos *tags* al lenguaje tales como:

- <audio> para la reproducción sonido [11].
- <video> para la reproducción de video [12].
- <canvas> para generación de gráficos dinámicos.

Con HTML5 se provee la funcionalidad de agregar audio a un documento *web* a través del *tag* <audio>. Cabe destacar que, a pesar de poder tener control sobre este *tag* mediante el uso de atributos HTML y/o JavaScript, la especificación del *tag* no cubre qué tipos de *codecs* son soportados por el navegador, y este aspecto varía entre todos los navegadores. Este es un punto que se debió investigar al comienzo de la etapa de implementación del proyecto.

HTML5 también provee la funcionalidad de agregar video directamente a un documento *web*. Al igual que para el elemento <audio>, el *tag* <video> fue estandarizado por la W3C, no siendo así el soporte del *codec* de video por parte de los navegadores.

En la siguiente sección se ahonda sobre el *tag* <video> de HTML5 y el estudio del mismo que debimos realizar para abarcar la fase de implementación del proyecto.

6.2 *Tag* <video>

El *tag* <video> provisto por HTML5 permite embeber videos en las páginas *web*. Antes de su aparición, la única forma de lograr esto era a través del uso de *plugins* tales como Apple QuickTime o Adobe Flash.

---

[60] Andrew Grieve. *Gmail for mobile HTML5 Series: Using AppCache to Launch Offline*. 28 de Abril de 2009.

[61] *V8 JavaScript Engine,* https://code.google.com/p/v8/. Julio de 2013.

[62] *SpiderMonkey JavaScript Engine*, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. Julio de 2013.

Actualmente, el *tag* <video> de HTML5 es soportado por varios navegadores, incluidos los de dispositivos móviles. En la Tabla 11 se puede apreciar el soporte nativo del *tag* discriminado por cada navegador.

	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Android Browser
Versión	9.0+	3.5+	4.0+	4.0+	10.5+	3.2+	2.3+

Tabla 11. Soporte nativo de cada navegador para el tag <video> de HTML5 [13]

El *tag* <video> permite especificar múltiples tipos de video y cada navegador elegirá el que soporta de ellos, o ninguno en caso de no soportar ninguno. Los navegadores que no soportan el elemento video de HTML5 ignoran el *tag* <video> en forma completa, pueden aceptar utilizar un *plugin* en su lugar como última opción. Para saber qué tipo de formatos de video soporta un navegador web, se utiliza la librería Modernizr[63], que permite detectar soporte HTML5 y CSS3 de un navegador, de la siguiente forma:

```
if (Modernizr.video) {
  if (Modernizr.video.webm) {
    // try WebM
  } else if (Modernizr.video.ogg) {
    // try Ogg Theora + Vorbis in an Ogg container
  } else if (Modernizr.video.h264){
    // try H.264 video + AAC audio in an MP4 container
  }
}
```

Los resultados que se obtienen a partir de ejecutar el código anterior en el navegador web son los listados en la Tabla 12.

Para el prototipo se optó por utilizar el navegador Google Chrome, dado que es el navegador que se encuentra más avanzado en el desarrollo de las APIs de HTML5 que se requirieron para implementar. Por mismas razones, el contenedor de archivo utilizado es WebM (ver el *Anexo 1 Formatos de contenedor y codecs*).

[63] *Modernizr JavaScript Library*. http://modernizr.com/. Mayo de 2013.

Codec / Contenedor	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Android Browser
VP8 y Vorbis en WebM	-	4.0+	6.0+	-	10.6+	-	2.3+
Theora y Vorbis en Ogg	-	3.5+	5.0+	-	10.5+	-	-
H.264 y MP3 en MP4	9.0+	(Parcial)	5.0+	-	-	-	(Parcial)
H.264 y AAC en MP4	9.0+	(Parcial)	5.0+	3.1+	-	3.2+	(Parcial)
Otro formato	-	-	-	3.1+ (usando QuickTime)	-	-	-

Tabla 12. Soporte nativo de codec para el tag <video> de HTML5 [14]

6.3 Lenguaje JavaScript

JavaScript es el lenguaje de programación *web de-facto*, y se encuentra en la *web* desde mucho antes de HTML5. Su diseño fue originado para ser un lenguaje liviano para programar del lado del cliente, orientado a objetos e interpretado[64].

Debido a que el mismo se originó para correr del lado del cliente (aunque ahora existen servidores que corren en este lenguaje tales como Node.js), el mismo responde de forma rápida a las interacciones con el usuario.

A pesar de esto, JavaScript es también uno de los lenguajes de programación mayormente malinterpretados, y no fue considerado en forma seria por varios programadores profesionales al momento de su surgimiento.

El primer inconveniente de este malentendido comienza por el nombre del lenguaje. El prefijo Java sugiere en forma incorrecta que se encuentra relacionado con el lenguaje de programación Java, mientras que el sufijo Script sugiere que no se trata de un lenguaje de programación en sí. Además, las versiones iniciales del lenguaje no contaban con funcionalidades tales como el manejo de errores, herencia y funciones internas. A pesar de que todas estas funcionalidades se encuentran implementadas hoy en día, y de que la última versión del lenguaje compone un lenguaje completamente orientado a objetos (pero basado en prototipos), las versiones previas crearon una mala opinión de las capacidades del lenguaje.

JavaScript es también conocido por algunas de sus grandes deficiencias como lenguaje, lo cual ha dado cabida a muy malas practicas de programación que han sido adoptadas por muchos desarrolladores. Igualmente, el núcleo del lenguaje se compone de un muy

[64] *JavaScript Language*. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Mayo de 2013.

expresivo lenguaje de programación, y se convirtió en el único lenguaje de programación *web* adoptado por todos los navegadores.

La principal razón por la cual JavaScript es el lenguaje *web* más popular se sustenta en la ejecución asíncrona de su código mediante el uso de funciones de *callback*[65]. Las funciones d e *callback* son métodos que pueden ser pasados como argumento a otro método, permitiendo especificar métodos que son llamados por cualquier evento.

A pesar de que esta característica complica la legibilidad del código, trae como beneficio un menor tiempo de ejecución. Dado que la decisión de implementación del código no bloqueante depende de cada navegador, la implementación de JavaScript se implementa usualmente como una hebra sola.

A modo de poder almacenar y transmitir información entre los servidores *web* y los clientes, se definió el formato de intercambio de datos conocido como JavaScript Object Notation (JSON)[66]. JSON consiste en un formato de datos liviano y fácilmente legible para los programadores, y se utiliza principalmente para serializar y transmitir datos a través de la red reemplazando el lenguaje XML. Su estructura consiste principalmente de una colección de pares clave, valor.

[65] *Callback functions*. https://developer.mozilla.org/en-US/docs/Mozilla/js-ctypes/js-ctypes_reference/Callbacks. Mayo de 2013.
[66] *JavaScript Object Notation (JSON).* https://developer.mozilla.org/en/docs/JSON. Mayo de 2013.

7 *Web Sockets*

La tecnología WebSocket permite bajar la latencia, debido a que, una vez establecida una conexión por *web sockets*, el servidor puede enviar mensajes al cliente apenas se encuentre disponible para hacerlo (en forma asíncrona). La Figura 37 compara el comportamiento de los *web sockets* frente a la técnica de *polling*.

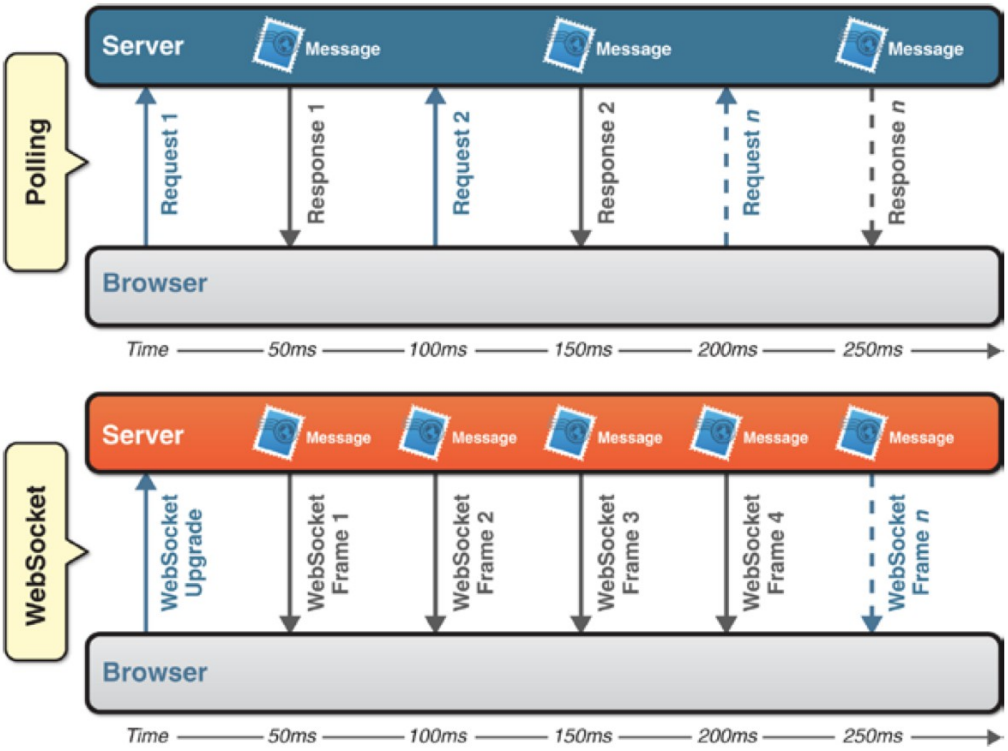

Figura 37. Polling vs. WebSockets [16]

La especificación de WebSocket detalla que al abrir una comunicación con *web sockets* entre cliente y servidor, se crea en verdad una conexión *full duplex* persistente entre ambos a través de *sockets*, y ambas partes comienzan a enviarse datos en forma asíncrona.

La Figura 38 ilustra una arquitectura básica basada en *web sockets* en la cual los navegadores utilizan una conexión *full duplex* directa con otros *hosts* remotos [15].

Otra de las ventajas sustanciales de los *web sockets* consiste en su habilidad para pasar a través de *firewalls* y *proxies*. Cuando un *web socket* detecta la presencia de un servidor *proxy*, automáticamente crea un túnel para pasar a través del mismo.

El túnel se establece al enviar un HTTP CONNECT al servidor *proxy*, lo cual exige que el servidor abra una conexión TCP a un *host* y puerto específico. Existe también la posibilidad de crear *web sockets* seguros que corran sobre SSL. Actualmente, Chrome es el navegador que soporta los *web sockets* de forma nativa en forma completa, abarcando lo anteriormente mencionado.

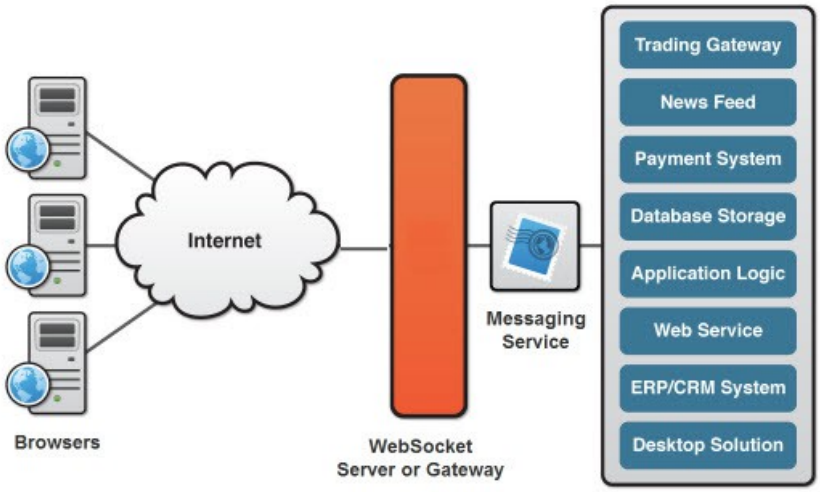

Figura 38. Comunicación WebSocket full-duplex entre navegadores y hosts remotos [16]

Los *web sockets*, además de ser una API provista por los navegadores para establecer conexión *full duplex*, consisten en un protocolo de capa de aplicación. El establecimiento del protocolo WebSocket consta de un *handshake* de dos fases [16], como se puede apreciar en la Figura 39.

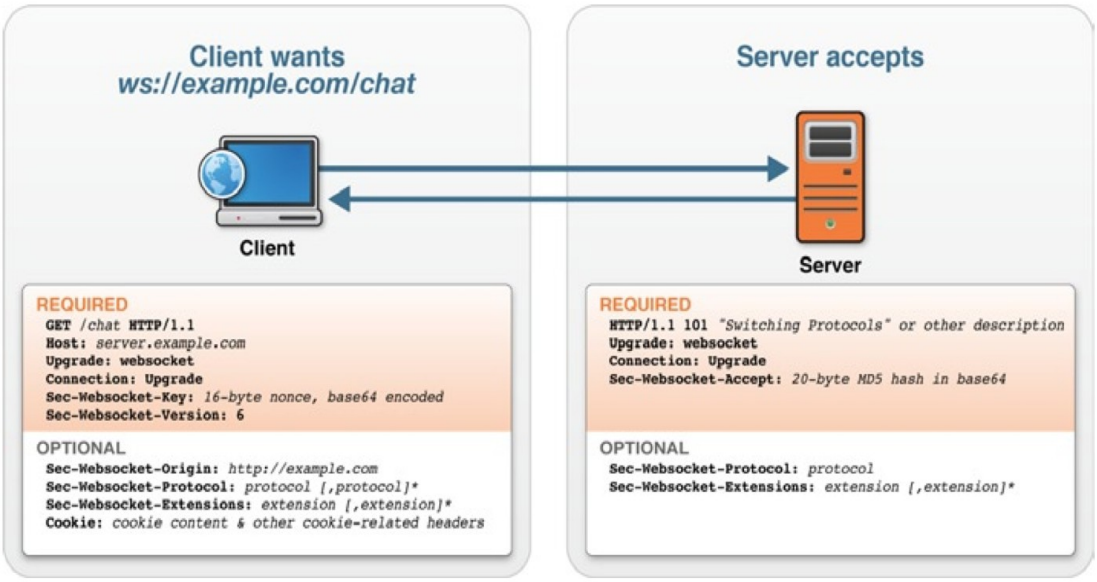

Figura 39. Ejemplo de un opening handshake de WebSocket [16]

1. *WebSocket Opening Handshake*: toda conexión WebSocket comienza con un HTTP *request* que incluye un cabezal especial *Upgrade* que indica que el cliente desea realizar un *upgrade* de la conexión a un protocolo diferente. Por ejemplo, en caso que el cliente desee establecer una conexión por WebSocket a ws://echo.websocket.org/echo, se envían los siguientes paquetes HTTP entre cliente y servidor:

request HTTP del cliente:

```
GET /echo HTTP/1.1
Host: echo.websocket.org
Origin: http://www.websocket.org
Sec-WebSocket-Key: 7+C600xYybOv2zmJ69RQsw==
Sec-WebSocket-Version: 13
Upgrade: websocket
```

response HTTP del servidor:

```
101 Switching Protocols
Connection: Upgrade
Date: Wed, 20 Jun 2012 03:39:49 GMT
Sec-WebSocket-Accept: fYoqiH14DgI+5ylEMwM2sOLzOi0=
Server: Kaazing Gateway
Upgrade: WebSocket
```

Las conexiones por *web sockets* serán exitosas solo si el servidor responde con un código 101, el header *Upgrade*, y el header *Sec-WebSocket-Accept*. El valor de este último cabezal deriva del cabezal *Sec-WebSocket-Key* del *request*.

2. Respuesta de clave computada: Para completar el *handshake* en forma exitosa, el servidor WebSocket debe responder con una clave computada. Esta respuesta exhibe que el servidor entiende el protocolo WebSocket.

Una vez establecida la conexión WebSocket, las tramas de datos son enviados entre cliente y servidor en modo *full duplex*, con el formato de *frame* de la Figura 40.

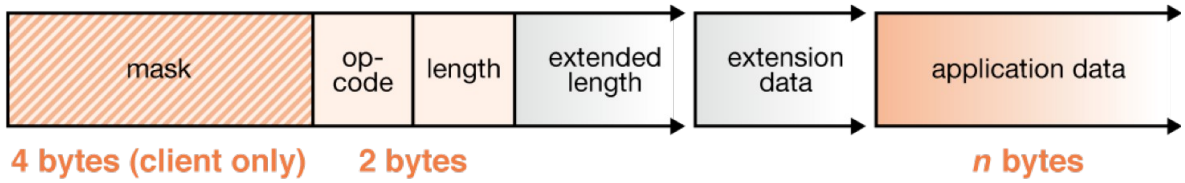

Figura 40. Formato de un frame WebSocket [16]

En [18] se puede acceder a una lista de demostraciones *online* del funcionamiento del protocolo WebSocket.

8 WebM Byte Streams

En esta sección se define el formato de los segmentos para implementaciones que optan por soportar el formato de contenedor WebM. En particular, la API Media Source de HTML5 utiliza este formato para la concatenación de segmentos de audio/video sobre el *buffer* de un elemento HTMLMediaElement (Por información más detallada ver [19]). Este contenedor de video tiene dos tipos de segmentos, el de inicialización y el *multimedia*.

Particularmente, el segmento de inicialización que contiene meta-datos, es una secuencia de *bytes* que contiene toda la información necesaria para decodificar la secuencia de segmentos *multimedia* que lo sucede. Esta información incluye datos de inicialización del *codec*, identificadores sobre al multplexación de segmentos y los *offset* de comienzo de los mismos.

Luego, los segmentos *multimedia* son una secuencia de *bytes* que contienen datos empaquetados y que representan una porción de la línea del tiempo de reproducción. Estos segmentos siempre están asociados al segmento de inicialización más próximo.

Para mayor información sobre este formato contenedor de media, ver [38]. Un mayor análisis a detalle sobre los pormenores de este, excede el alcance de este proyecto y por ende no será presentado en esta sección.

9 WebRTC

9.1 Arquitectura *Full Mesh*

En una arquitectura *Full Mesh* pura, como la que se muestra en la Figura 41, la idea es que cada navegador establezca un conjunto de Peer Connections con cada uno de los otros clientes de la red.

Todos los clientes deben conocer a todos los otros *peers* para que esto ocurra. Si un navegador esta recibiendo flujos de audio del resto de los clientes, se podría mezclar el audio de cada uno para reproducir una sola línea de sonido.

Por otra parte, si lo que se está recibiendo son flujos de video, entonces el navegador podría *renderizar* dichos flujos en ventanas separadas para mostrarlo al usuario. A medida que diferentes clientes se unen a la sesión, se generan nuevas Peer Connections entre los navegadores para transmitir los datos.

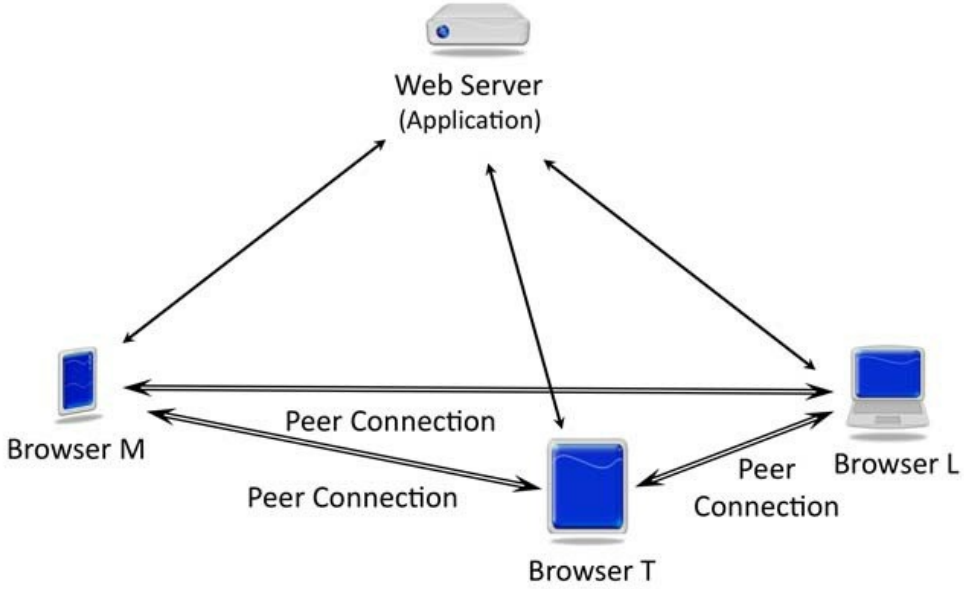

Figura 41. Múltiples Peer Connections entre navegadores [25]

9.2 Arquitectura *Centrally Mixed*

Un enfoque diferente es el denominado *Centrally Mixed*. En esta arquitectura aparece el concepto de *Mixer* o *Selector*, que es un segundo servidor (aparte del servidor *web*) centralizado, que se encarga de manejar las Peer Connections con los diferentes clientes.

Por lo tanto, en esta segunda disposición, en principio un cliente no tiene por qué estar al tanto del resto de los *peers*. Dicho servidor también puede servir el contenido *multimedia* para el resto de los navegadores. Por lo tanto, el *Mixer* actúa como una entidad centralizada que regula la sesión entera. En la Figura 42 puede verse más claramente lo planteado.

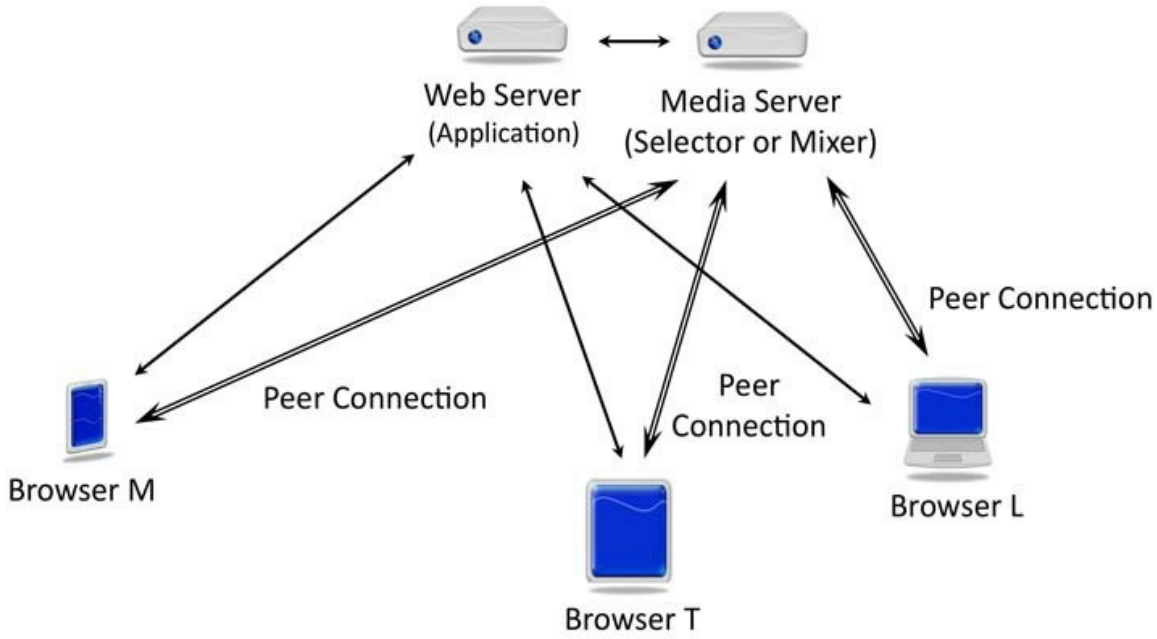

Figura 42. Una sola Peer Connection por navegador con el Media Server [25]

Como se describe en la Figura 42, la arquitectura *Centrally Mixed* requiere que se establezca una sola Peer Connection entre el navegador y el servidor de datos *multimedia*. Cada navegador envía información al servidor central, quien luego se encarga de propagarla hacia el resto de los navegadores.

Desde la perspectiva del navegador M, los flujos de datos que vienen desde los navegadores L y T son recibidos sobre una única Peer Connection con el servidor central. A medida que nuevos clientes se integran a la sesión, no hay necesidad de establecer nuevas Peer Connections con el navegador M.

De lo anterior se desprende que la opción *Full Mesh* presenta las siguientes ventajas:

- No necesita la infraestructura que involucra mantener un servidor central.
- Consigue obtener menores valores de latencia en los flujos y también mayor calidad de servicio (siempre y cuando la sesión no involucre a demasiados clientes). Esto se debe a que el ancho de banda y la capacidad de procesamiento crece en cada navegador a medida que un nuevo cliente se une, ya que se tiene que establecer una conexión con cada uno de los navegadores de la sesión.

Por otra parte, la opción *Centrally Mixed* es capaz de escalar a sesiones con grandes cantidades de clientes, minimizando el procesamiento en cada navegador cada vez que un nuevo navegador se une a la misma. A pesar de esto, puede verse como una estructura ineficiente si la cantidad de clientes de la sesión no es demasiado grande.

En conclusión, ambas organizaciones son válidas en su contexto. Si se piensa cuál de estas soluciones se adapta más al problema de realizar una red P2P colaborativa de *streaming* de audio y video, el concepto del servidor central se adapta al de los *Super Peers* definidos en *GoalBit* y también al del *tracker* como entidad reguladora.

Sin embargo, en una red colaborativa también es necesaria la comunicación directa entre los clientes de un mismo *swarm*, por lo que a primera vista, para solucionar este problema, se necesitaría una opción híbrida entre una red *Full Mesh* y una red *Centrally Mixed*.

9.3 Protocolos utilizados por WebRTC

9.3.1 Real-Time Transport Protocol

El protocolo más importante utilizado por WebRTC es el Real-Time Transport Protocol (RTP). Más precisamente, WebRTC solo utiliza la versión segura de RTP: Secure RTP o SRTP.

SRTP es el protocolo utilizado para transportar paquetes de audio y video entre los clientes WebRTC y normalmente corre sobre UDP. Dichos paquetes contienen la información digitalizada del contenido de audio o video que puede surgir del micrófono o la cámara *web* de un usuario final.

Una Peer Connection satisfactoria, en conjunto con un acuerdo de formatos a transmitir entre los nodos involucrados, resultará en una conexión SRTP por donde se transmitirán los datos *multimedia* entre dos navegadores, o entre un navegador y un servidor.

Para conexiones donde lo que se quiera intercambiar no sean datos *multimedia* de la cámara o del micrófono, SRTP no se utiliza. Para estas situaciones, la idea es usar la API RTCDataChannel de WebRTC para abrir una conexión diferente y así poder intercambiar otro tipo de formatos de información.

9.3.2 Session Description Protocol

La descripción de una sesión WebRTC se genera a través del uso del protocolo Session Description Protocol (SDP). Una descripción de la sesión se utiliza para definir las características del contenido *multimedia* a intercambiar.

En particular, la definición de las características de la sesión es una de las primeras tareas a realizar antes de lograr la conexión SRTP e intercambiar datos. Mediante el uso de este protocolo, los nodos se envían entre sí una serie de mensajes que ayudan a definir, por ejemplo, en qué formato estarán codificados los datos *multimedia* a intercambiar. Esta interacción es conocida como el intercambio *offer*/*answer* y su objetivo es generar un acuerdo entre los *peers*.

9.3.3 Interactive Communication Establishement

Otro protocolo clave utilizado en WebRTC es Interactive Communication Establishment (ICE). En forma resumida, sus dos funciones más importantes son:

1. Permitir a clientes WebRTC el intercambio de datos a través de dispositivos que estén detrás de una red con Network Address Translation (NAT).
2. Proveer una verificación del consentimiento de una comunicación. Esto significa que los paquetes a enviar sólo se enviarán a navegadores que estén esperando el tráfico. Esto es necesario por razones de seguridad.

ICE utiliza la técnica Hole Punching para lograr su cometido (ver el *Anexo 9.5.3 Hole Punching* de *9 WebRTC*).

9.3.4 Session Traversal Utilities

Session Traversal Utilities for NAT (STUN) es un protocolo que ayuda a resolver el problema del NAT Traversal. En WebRTC, se incorpora un cliente STUN en el navegador. Este cliente envía determinado paquete a un servidor STUN, con el objetivo de obtener su dirección IP pública y puerto. Esta información es luego utilizada para construir una lista de direcciones candidatas para realizar Hole Punching con ICE.

9.3.5 Traversal Using Relays arount NAT

Traversal Using Relays around NAT (TURN) es una extensión de STUN que provee un servicio de *relay* para cuando no es posible realizar el Hole Punching de ICE.

En WebRTC, se incorpora también al navegador un cliente TURN que se pueda comunicar con un servidor TURN alojado en algún lugar de la red con una dirección IP estática y pública.

9.3.6 Transport Layer Security

Transport Layer Security (TLS) es la nueva versión de SSL y su objetivo es aportar servicios de confidencialidad y autenticación a la capa superior. La confidencialidad se logra a través de la utilización de métodos de encriptación sobre el *payload* de la capa superior. La autenticación se provee utilizando certificados digitales.

WebRTC utiliza TLS para aportar seguridad a mensajes de señalización. DTLS (Datagram TLS) es una implementación de TLS que corre sobre UDP y una versión de la misma puede ser utilizada para brindar servicios de seguridad a SRTP. La misma es conocida como DTLS-SRTP.

9.3.7 Stream Control Transport Protocol

El Stream Control Transport Protocol (SCTP) es un protocolo de capa de transporte que provee transporte confiable de datos sobre IP, con control de congestión y la capacidad de mantener múltiples flujos en una sesión.

SCTP no es normalmente soportado en la mayoría de los sistemas operativos, por lo que es necesario agregar la implementación dentro del navegador.

9.4 Señalización

Como se explicó en la Sección *2.3.5.3.2 Conexión entre dos peers*, la API RTCPeerConnection de WebRTC permite comunicar datos de *streaming* entre dos o más navegadores. Para ello, la misma requiere de un mecanismo para coordinar la comunicación y enviar mensajes de control, también conocidos como mensajes de señalización.

La señalización es el proceso por el cual se coordina la comunicación entre *peers*. Para que una aplicación WebRTC pueda establecer una comunicación, sus clientes deben intercambiar la siguiente información:

- Mensajes de control de sesión para abrir o cerrar una comunicación.
- Mensajes de error.
- Meta-datos *multimedia* tales como *codecs* y configuración de *codecs*, ancho de banda y tipos de *multimedia*.
- Datos de seguridad.

- Datos acerca de la red, tales como la dirección IP y puerto públicos del *peer* que corre detrás de una NAT.

Este proceso de señalización requiere que los clientes tengan una forma de pasarse mensajes. Este mecanismo no es implementado por la API de WebRTC: el desarrollador debe utilizar un canal *full duplex* a elección tal como *web sockets*.

El protocolo de señalización a ser utilizado por los *peers* no es especificado por el estándar de WebRTC para evitar la redundancia y maximizar la compatibilidad de tecnologías ya establecidas. Este enfoque es resaltado en la especificación del protocolo JavaScript Session Establishment Protocol (JSEP) [36], que brinda una arquitectura de tipo *offer*/*answer* para la señalización:

"La idea detrás del establecimiento de una llamada WebRTC es la de especificar en forma completa y controlar el plano multimedia, pero delegar el plano de la señalización a la aplicación lo más posible. Lo razonable es que distintas aplicaciones puedan preferir utilizar distintos protocolos, tales como los protocolos de señalización SIP o Jingle, o alguno customizado para la aplicación en particular… Según este enfoque, la información clave que precisa ser intercambiada es la descripción de la sesión multimedia, que especifica la información necesaria de transporte y configuración de media para establecer el plano de media."

La arquitectura de JSEP también evita que los navegadores tengan que guardar estado. Ésto es, que el navegador funcione como una máquina de estados de señalización. Esto evita el problema de que al recargar un navegador se pierdan los datos de señalización del navegador. En su lugar, la información del estado de la señalización se guarda en el servidor (ver Figura 43).

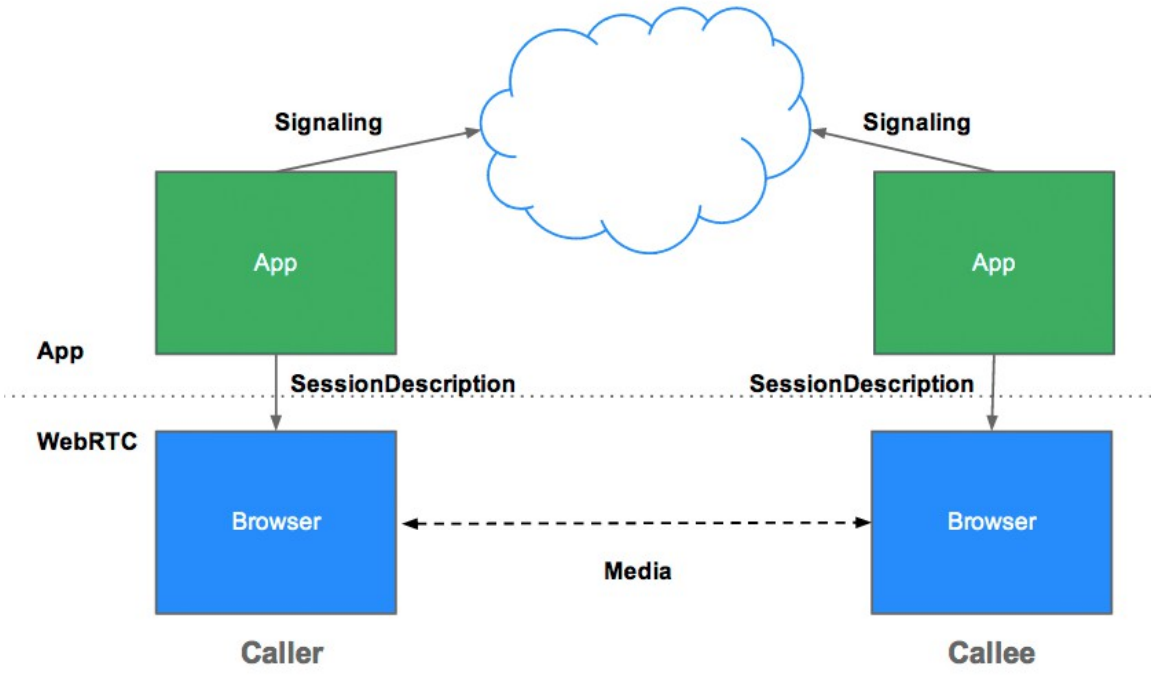

Figura 43. Arquitectura JSEP [36]

JSEP requiere el intercambio de mensajes de *offer* y de *answer* por parte de los *peers*. Estos mensajes se comunican en el formato Session Description Protocol (SDP), que luce de la siguiente forma:

```
v=0
o=- 7614219274584779017 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE audio video
a=msid-semantic: WMS
m=audio 1 RTP/SAVPF 111 103 104 0 8 107 106 105 13 126
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:W2TGCZw2NZHuwlnf
a=ice-pwd:xdQEccP40E+P0L5qTyzDgfmW
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=mid:audio
a=rtcp-mux
a=crypto:1 AES_CM_128_HMAC_SHA1_80 inline:9c1AHz27dZ9xPI91YNfSlI67/EMkjHHIHORiClQe
a=rtpmap:111 opus/48000/2
…
```

SDP define un formato para describir los parámetros iniciales de una sesión de *streaming multimedia* (por ejemplo la calidad de resolución de un video) y se utiliza para realizar la negociación de los formatos a transmitir entre los dos navegadores. Los *peers* deben utilizar este protocolo previo a la comunicación de datos *multimedia*, de lo contrario, los reproductores residentes en los mismos no sabrían cómo decodificar los flujos de datos arribantes.

Una vez que el proceso de señalización se completó en forma exitosa, se puede hacer *streaming* de los datos de un *peer* a otro, entre llamador y llamado, o a través de un servidor TURN como se describió en la Sección *2.3.5.3.2 Conexión entre dos peers*.

La negociación de formatos suele darse al comienzo del acuerdo de la sesión, pero también está previsto que durante un intercambio de datos sea necesario renegociar los parámetros previamente establecidos. Esto significa que si un *streaming* se está realizando en determinada calidad de video, en un momento dado dicha calidad puede ser re-evaluada, permitiendo modificar el tipo de formato en el que se están enviando los datos *multimedia* para lograr aumentar la *performance* de la transmisión o incluso una mejora en la experiencia de usuario o calidad de servicio.

9.5 *Multimedia* P2P con WebRTC

WebRTC utiliza flujos *multimedia* únicos entre *peers*, en donde el video, el audio y las conexiones de datos son establecidas directamente entre los navegadores. Desafortunadamente, el protocolo NAT y los f*irewalls* dificultan (y a veces imposibilitan) esta tarea, por lo que se requiere de protocolos y procedimientos especiales para solucionar el problema.

9.5.1 Flujo multimedia en WebRTC

A modo de ejemplificar el flujo de los datos *multimedia* en WebRTC, se utilizan la realidad planteada en la Figura 44 según [25], en la que se cuenta con 4 navegadores, 3 de los cuales corren en dispositivos conectados por WiFi, y uno conectado a través de un *router* corporativo.

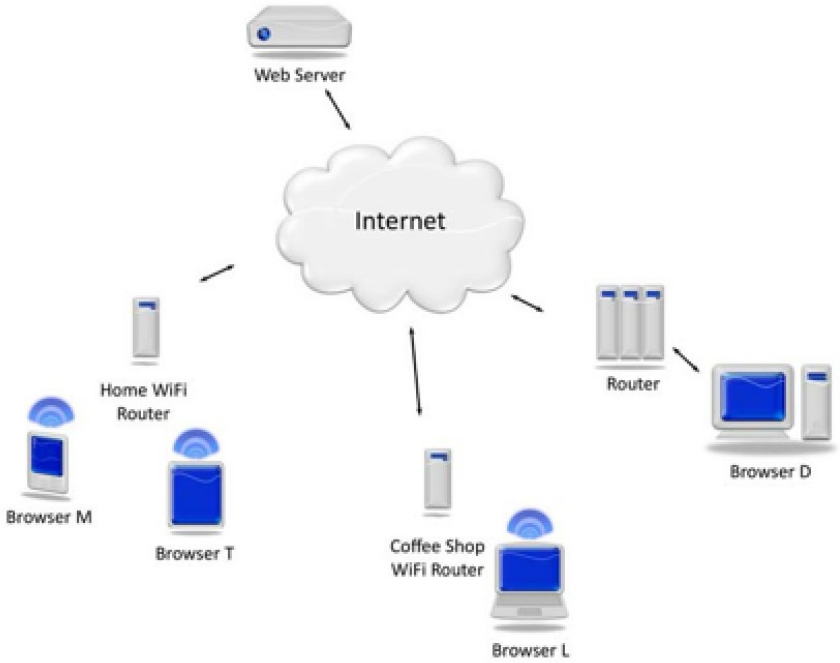

Figura 44. Navegadores WebRTC conectados a Internet [25]

Sin contar con WebRTC es posible establecer flujos *multimedia* a través de la red. Sin embargo, estos flujos deben trazar el mismo recorrido que el tráfico normal de internet. En otras palabras, los paquetes deben navegar desde el navegador M al servidor *web*, y luego del servidor *web* al otro D, por ejemplo (ver Figura 45). En una arquitectura así. el servidor *web* debe manejar todo el tráfico extra. Los *streamings* de video de alta definición pueden utilizar un ancho de banda considerable, lo que limita la escalabilidad de este tipo de arquitectura.

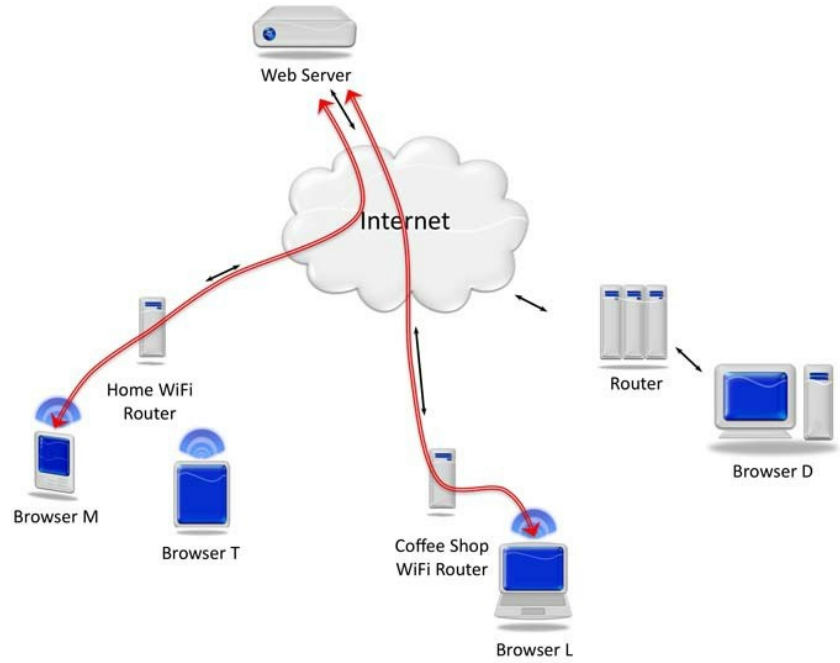

Figura 45. Flujo multimedia sin WebRTC [25]

El objetivo de la API RTCPeerConnection es habilitar la capacidad de establecer conexiones P2P directamente entre los navegadores. Un flujo de datos *multimedia* que utilice esta API puede verse en la Figura 46.

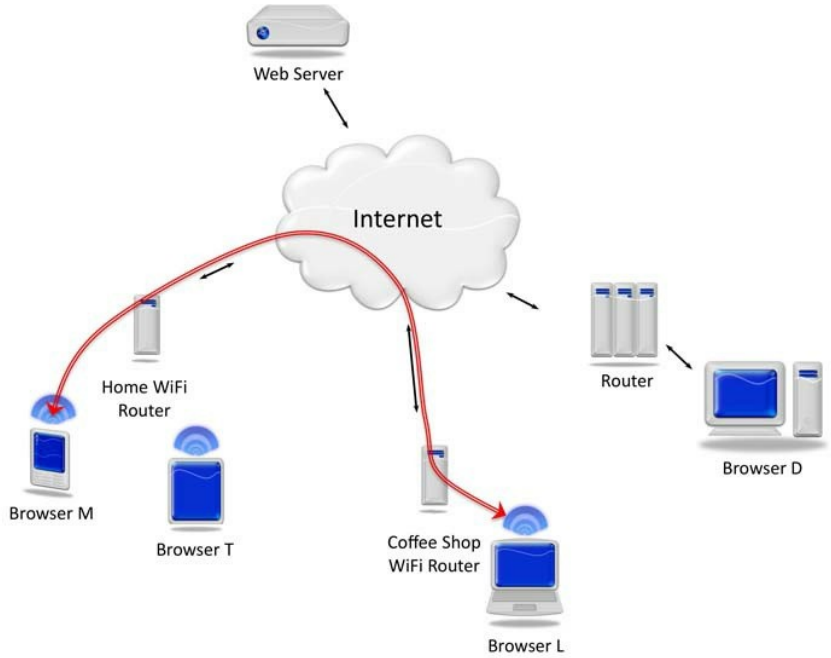

Figura 46. Flujo multimedia P2P con WebRTC [25]

Este camino puede tener menos saltos en Internet, tomar menos tiempo (porque hay menor latencia), y tener menos probabilidad de pérdida de paquetes. Como resultado, este tipo de flujo P2P puede generar conexiones de mejor calidad. A su vez, reduce el ancho de banda utilizado por el servidor *web* y también mitiga el problema de la cercanía de dos navegadores. Por ejemplo, si dos navegadores se encuentran en Montevideo, no es necesario que los paquetes viajen hasta un servidor *web* en Estados Unidos y luego regresen para cerrar el circuito, sino que pueden comunicarse directamente a través del ruteo más corto posible.

A pesar de esto, establecer un flujo *multimedia* de este estilo puede resultar complicado, dado que la mayoría de los dispositivos conectados a Internet se encuentran detrás de una NAT, como se explica en la siguiente sección.

9.5.2 WebRTC y NAT

Un escenario más real al planteado en la Figura 44 es el que se presenta en la Figura 47, en donde cada navegador corre sobre un dispositivo que se oculta detrás de una NAT.

Muchos protocolos de Internet no tienen dificultades para realizar el NAT Traversal, en particular el protocolo TCP de capa de transporte. A pesar de esto, los protocolos P2P y los que utilizan el protocolo UDP de capa de transporte pueden enfrentarse con muchas dificultades, como es el caso de WebRTC, que utiliza UDP y TCP.

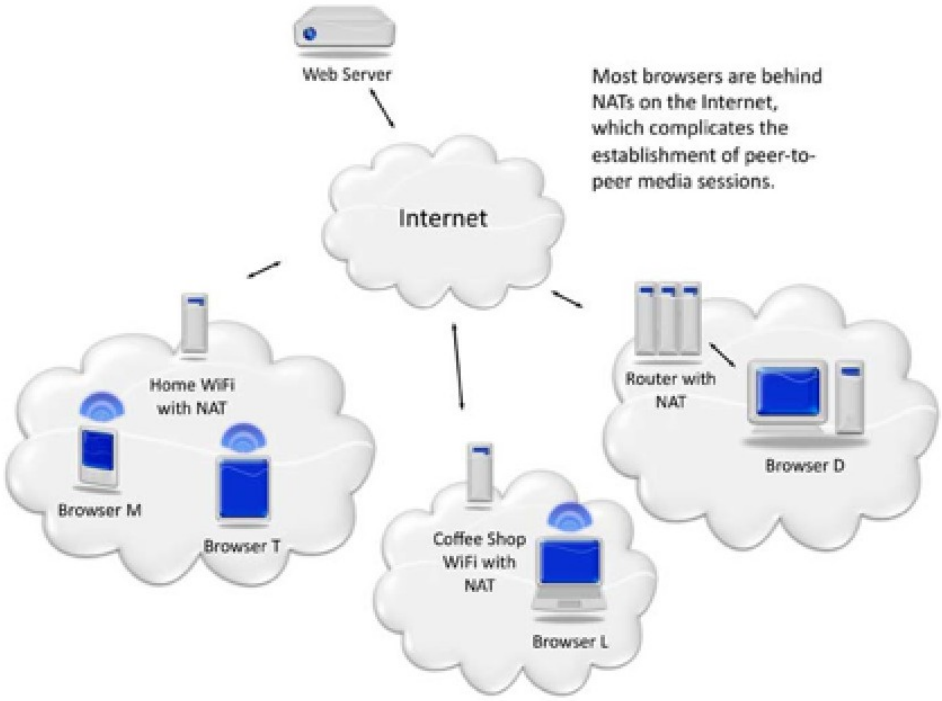

Figura 47. Navegadores WebRTC detrás de una NAT [25]

El escenario de la Figura 48 muestra un flujo *multimedia* de datos P2P a través de múltiples NAT que puede ser establecido en WebRTC, utilizando técnicas de Hole Punching (ver el *Anexo 9.5.3 Hole Punching*). El flujo *multimedia* puede pasar a través del servidor *web* y fluir directamente entre ambos navegadores a través de sus NATs.

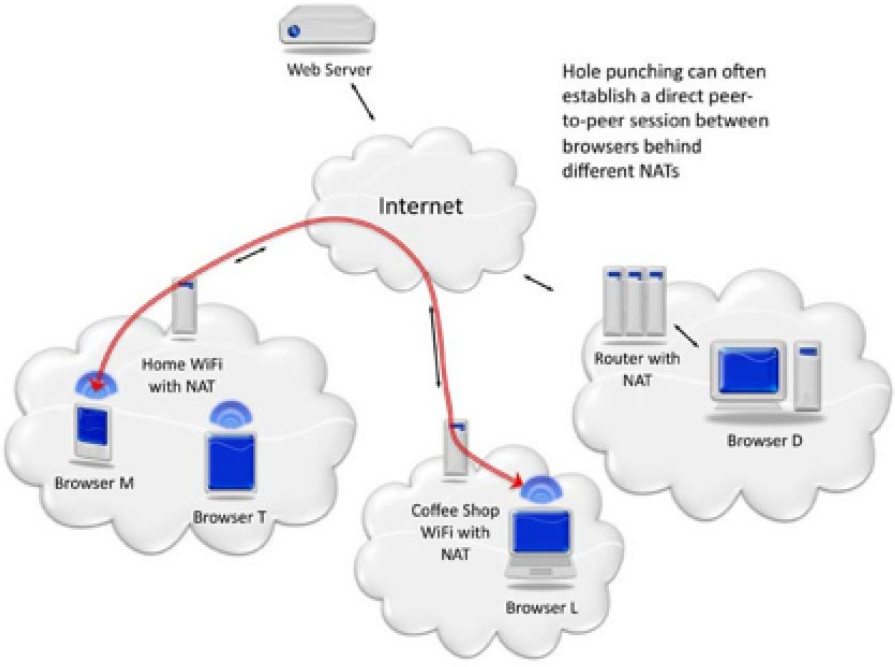

Figura 48. Flujo multimedia a través de múltiples NAT en WebRTC [25]

Por otro lado, el escenario de la Figura 49 plantea el caso donde se establece una sesión *multimedia* entre dos navegadores que se encuentran detrás de una misma NAT. En este caso particular, el camino óptimo del flujo es el de rutear dentro de la NAT y nunca salir a la

Internet pública. Al igual que para el caso anterior se precisa de la técnica de Hole Punching para lograr el flujo planteado.

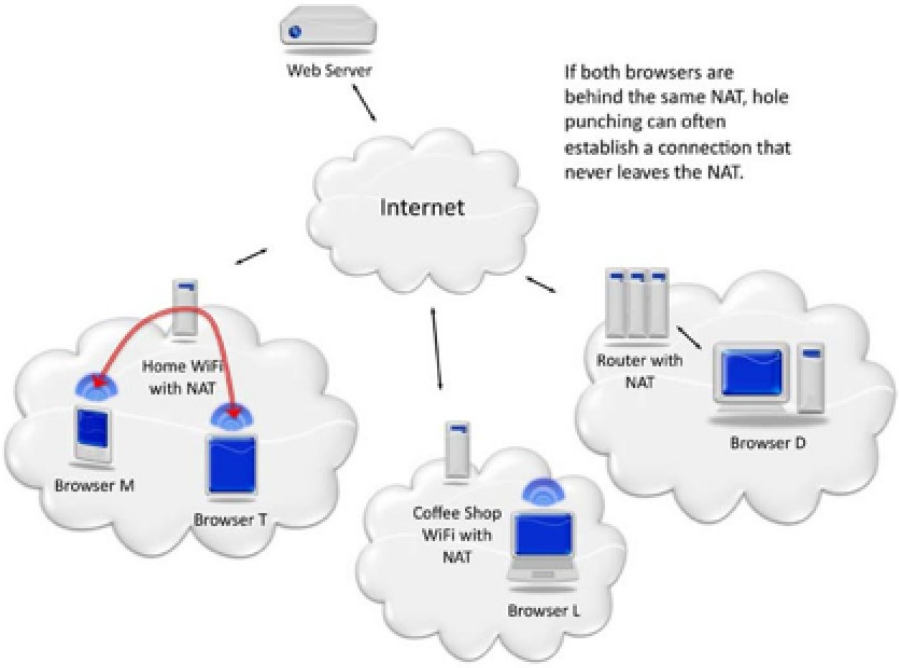

Figura 49. Flujo multimedia entre navegadores detrás de misma NAT en WebRTC [25]

Con respecto al enfoque que implementa WebRTC para resolver el problema del NAT Traversal, el mismo expone un aspecto importante en lo que refiere a la privacidad de datos en la *web*. Más específicamente, en lo que concierne a la privacidad de las direcciones IP.

A modo de ejemplo, cuando el navegador M intenta comunicarse en forma P2P con el navegador T usando WebRTC, el primer navegador M expone su IP pública al servidor. Esta dirección IP puede revelar muchos datos acerca del usuario, tales como información espacial.

Otro ejemplo que refiere a este problema de privacidad de direcciones IP ocurre cuando dos navegadores intentan establecer una conexión entre ellos. Por ejemplo, cuando M establece una conexión con T, ambos navegadores conocerán las direcciones IP públicas del otro. A su vez, M podría llegar a conocer la dirección IP privada de T si la misma fuera compartida como dirección candidata por el Hole Punching. Esto consiste en una exposición de privacidad de datos introducida por la tecnología WebRTC que puede ser maliciosamente utilizada para extraer información de otros usuarios.

Información más detallada acerca de este problema y posibles soluciones al mismo se encuentran en [25], Sección 5.2.

9.5.3 Hole Punching

Si bien establecer sesiones P2P a través de una NAT puede resultar complejo, existen técnicas como el Hole Punching que son efectivas para resolver el problema (exitoso en el 85% de los casos según [25]). A pesar de esto, el porcentaje de éxito en el establecimiento de conexiones consiste en un promedio de muchos usuarios a través de muchas redes.

Algunas redes tendrán un porcentaje de éxito menor, como es el caso de los dispositivos móviles en Estados Unidos, que en el año 2013 resultó en un 30% según [25].

Para poder utilizar Hole Punching, se deben cumplir los siguientes requisitos:

1. Los dos navegadores que están intentando establecer la conexión directa deben enviar paquetes de tipo Hole Punching al mismo tiempo. Como resultado, ambos deberán estar al tanto de la sesión a ser establecida y conocer las direcciones IP a donde enviar los paquetes. No existe nada especial acerca de un paquete de tipo Hole Punching. Simplemente se trata de un paquete IP que es enviado para *testear* si una dirección de destino particular es alcanzable a través de la NAT.
2. Los dos navegadores deben conocer tantas direcciones IP como sea posible, que puedan ser utilizadas para alcanzarlos. Las mismas pueden ser direcciones IP privadas (detrás de la NAT), direcciones IP públicas o direcciones IP de servidores de *relay*.
3. Como último recurso, se utiliza un servidor de *relay* que tenga una dirección IP pública y que sea accesible por ambos navegadores cuando lo precisen.
4. Se deben utilizar flujos simétricos. Ésto es, que el tráfico UDP parezca operar de manera similar a como opera una conexión TCP.

El primer requisito se cumple con la ayuda del servidor *web*. Éste sabe que los navegadores intentarán realizar una conexión punto a punto, y por lo tanto se asegura que ambos navegadores comiencen el Hole Punching aproximadamente al mismo tiempo.

Para satisfacer el segundo requerimiento se utiliza un servidor STUN. Cada navegador hace un pedido al servidor STUN enviando un paquete de tipo STUN. Los servidores STUN responden indicando la dirección IP que se encuentra en el paquete STUN recién recibido. Ésto es, la dirección IP pública del último nodo que esté proveyendo funcionalidades de NAT al navegador de la red interna (por ejemplo, la dirección IP pública del *router* de una red doméstica).

Esta dirección IP aprendida por el servidor STUN es luego compartida con el otro navegador y se convierte en una dirección candidata. La dirección IP privada se obtiene a través del sistema operativo, consultando las interfaces de las tarjetas de red del dispositivo. En la Figura 50 se puede observar este proceso.

El tercer requisito se resuelve mediante el uso de servidores TURN. Al igual que un navegador hace un pedido a un servidor STUN previo a iniciar el Hole Punching, el navegador envía también un pedido a un servidor TURN para obtener la dirección IP de un servidor de *relay*. Esta última dirección IP de *relay* se agrega a la lista de direcciones candidatas.

Finalmente, para el cuarto y último requisito, el navegador debe enviar datos *multimedia* desde el mismo puerto UDP por el que el navegador se encuentra escuchando por datos *multimedia* entrantes. Esto hace que las dos sesiones RTP de un sólo sentido que corren sobre UDP parezcan para la NAT como si tratasen de una sola sesión RTP bidireccional.

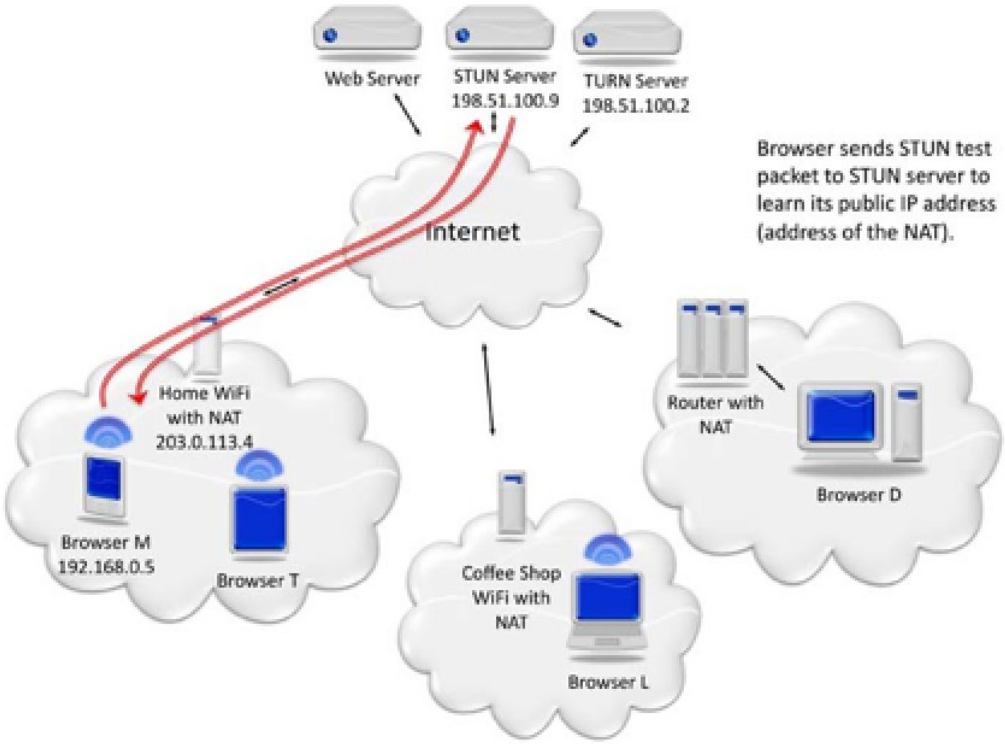

Figura 50. Uso del servidor STUN por parte de un navegador [25]

Si bien en la mayoría de los casos se logra establecer una conexión P2P entre los navegadores, cuando las reglas NAT son demasiado restrictivas el camino directo fallará y no quedará más opción que utilizar un servidor de *relay multimedia*. Esto resultará en que todos los datos pasarán a través de un servidor TURN, como se muestra en la Figura 51.

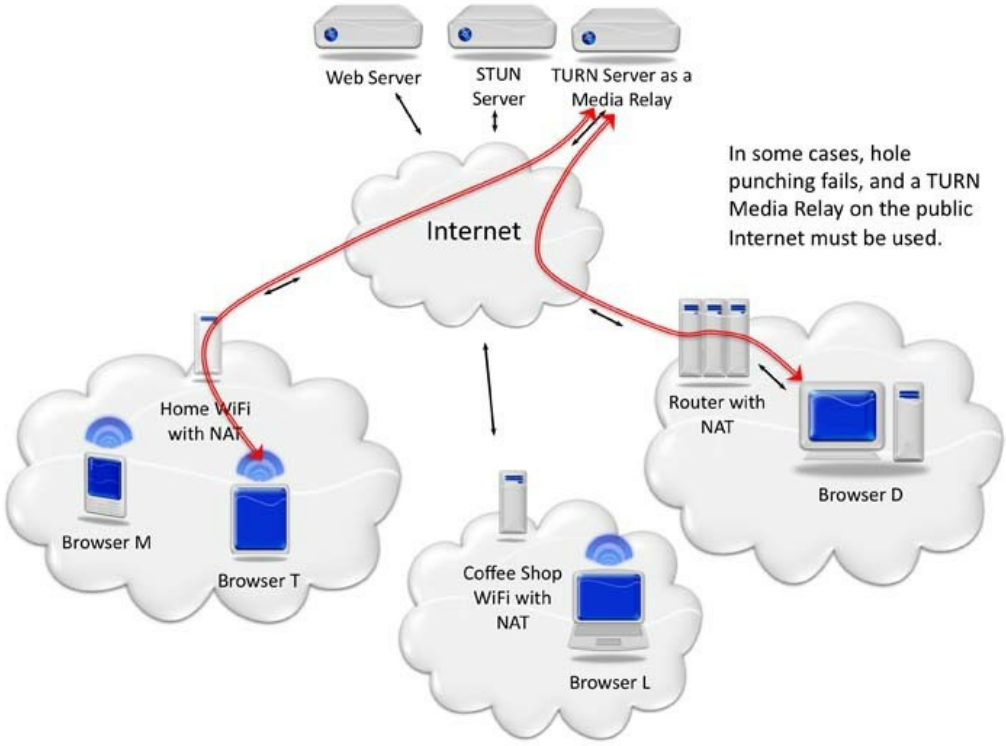

Figura 51. Datos multimedia a través de un servidor TURN de relay [25]

Claramente, este caso no es el ideal, pero por lo menos no se utiliza el servidor *web* para realizar el *relay*. También vale la pena repetir que este no es el caso más normal, sino una opción que se utiliza como última instancia, debido al fracaso del resto de los candidatos.

9.5.4 ICE y direcciones candidatas

Como se mencionó en la Sección *2.3.5.3.2 Conexión entre dos peers*, ICE es el protocolo que implementa el Hole Punching, y utiliza STUN y TURN para que los navegadores puedan comunicarse de forma directa como se puede ver en la Figura 52. Se explica ahora cómo utiliza las direcciones candidatas para lograr el Hole Punching.

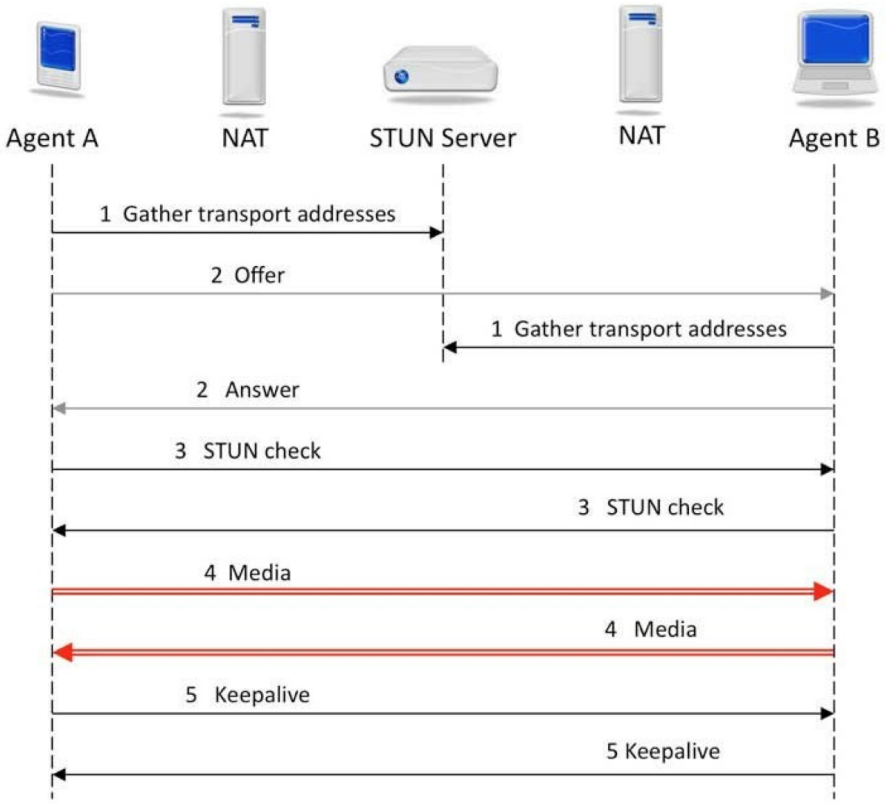

Figura 52. Flujo de ICE en alto nivel [25]

El procedimiento de ICE según la Figura 52 consisten en:

1. Juntar direcciones candidatas para el transporte de datos *multimedia*.
2. Intercambiar direcciones candidatas a través del canal de señalización.
3. Realizar chequeos de conectividad.
4. Seleccionar par de direcciones candidatas y comenzar el flujo de datos *multimedia*.
5. Enviar paquetes de tipo *keep-alive*.
6. Si cualquiera de los dos navegadores detecta un cambio de las direcciones IP utilizadas, realizar un ICE *restart*.

Las siguientes secciones explican el procedimiento anterior.

Paso 1. Recolección de direcciones candidatas

El primer paso del procedimiento ICE consiste en reunir direcciones candidatas para el transporte de datos *multimedia*. Las claves candidatas consisten en un par <dirección

IP>:<puerto> por donde los datos *multimedia* puedan llegar a ser recibidos por una Peer Connection. Estas direcciones deben ser reunidas en el momento de la llamada.

En la Figura 52, el agente ICE A comienza a reunir direcciones candidatas tan pronto como A inicia la Peer Connection con el navegador B. A su vez, el agente B comienza a reunir direcciones candidatas tan pronto como la Peer Connection pedida por A es recibida a través del canal de señalización.

Existen cuatro tipos de direcciones candidatas, como se puede ver ver en la Tabla 13. Se sugiere ver [25], sección 9.2.1 por más información acerca de las mismas.

Tipo de dirección candidata	**Uso**
Host	Dirección local obtenida a partir del sistema operativo y que representa la dirección de la Network Interface Card (NIC). Si se encuentra detrás de una NAT, se tratará de una dirección privada.
Server Reflexive	Dirección obtenida por un STUN *check* a un servidor STUN. Si se encuentra detrás de una NAT, se tratará de la dirección IP pública de la NAT más exterior.
Peer Reflexive	Dirección obtenida por un servidor STUN debido al pedido de otro agente ICE. Se trata de una nueva dirección candidata que es descubierta durante los chequeos de conectividad y que no es enviada por el canal de señalización.
Relayed	Dirección de un servidor multimedia de *relay*. Usualmente es obtenida realizando un pedido a un servidor TURN.

Tabla 13. Tipos de direcciones candidatas [25]

Paso 2. Intercambio de direcciones candidatas
El segundo paso del procedimiento seguido por ICE consiste en el intercambio de direcciones candidatas a través del canal de señalización. En primer lugar, las direcciones candidatas son ordenadas y priorizadas. En general, las candidatas de tipo *Host* son las de mayor prioridad, seguidas por las *Reflexive*, y por las *Relayed*.

Las direcciones candidatas son asociadas con un flujo *multimedia* particular en SDP. El comportamiento por defecto de WebRTC consiste en multiplexar todos los datos *multimedia* (video, audio y datos) sobre la misma dirección.

Paso 3. Chequeos de conectividad de STUN
Los agentes ICE comienzan con los chequeos de conectividad tan pronto como hayan enviado y recibido las direcciones candidatas. En la Figura 52, para el agente A se da cuando la respuesta SDP es recibida desde el agente B. Para el agente B, esto ocurre cuando la respuesta SDP es enviada por A. Durante esta fase, los agentes ICE generan

respuestas STUN a cualquier pedido de conexión STUN que reciban de sus *peers* al pasar la autenticación.

El primer paso consiste en hacer parejas de candidatos basadas en los tipos de direcciones IP (IPv4 o IPv6) y otros factores. El propósito de generar parejas es el de reducir el número de chequeos de conectividad realizados, a modo de minimizar el tiempo requerido para obtener un candidato. Durante este paso pueden descubrirse direcciones candidatas de tipo *Peer Reflexive* que son automáticamente emparejadas.

En [25], sección 9.2.3, se encuentran los cinco tipos de estados de conectividad en ICE.

Paso 4. Selección de par de direcciones candidatas
Los chequeos de conectividad continúan hasta que todos los chequeos posibles hayan sido completados, o hasta que un par haya sido seleccionado. La elección de un par la realiza un agente ICE. El protocolo ICE cuenta con un algoritmo para elegir qué navegador es el agente ICE Controlador, y qué navegador es el agente ICE Controlado.

El agente ICE Controlado se entera de que el otro agente ICE ha seleccionado una pareja de candidatos cuando recibe un chequeo de conectividad STUN con un atributo que indica que ésta será la pareja a ser utilizada. El agente ICE Controlado responde entonces al chequeo de conectividad haciendo un *echo* de la pareja a utilizarse. Un vez hecho esto, los datos *multimedia* comienzan a ser enviados por los navegadores utilizando la pareja candidata.

Paso 5. *Keep-Alives*
Para asegurarse que los mapeos NAT no realizan un *time out* durante la sesión *multimedia*, ICE continúa enviando chequeos de conectividad STUN durante intervalos de 15 segundos. Esto asegura que los paquetes sean enviados, incluso cuando los datos *multimedia* se encuentren pausados.

En caso de que la sesión *multimedia* aún se encuentre activa, el otro agente ICE genera una respuesta STUN. La recepción de esta respuesta STUN por parte del otro agente ICE se toma como un indicador de que los datos *multimedia* aún pueden ser enviados. Si la respuesta STUN no es recibida, se realiza un ICE *restart* (ver siguiente sección).

Paso 6. ICE *Restart*
El ICE *Restart* se ejecuta cuando un agente ICE detecta un cambio en la dirección de transporte base. Esto ocasiona que el agente ICE vuelva al paso 1 de la Figura 52, donde deberá reunir candidatos y enviarlos en una oferta SDP al otro agente ICE. Consecuentemente, el otro agente ICE debe también volver al paso 1, y todo el proceso deberá ser repetido.

Cabe destacar que esto ocurre si una página de un navegador que mantenga una Peer Connection es recargada por parte del usuario. A esto se le conoce como “rehidratación”, y consiste en un área activa de discusión en los estándares acerca de cuál es la mejor forma de encarar esta situación.

9.6 Ejemplo de implementación de aplicación WebRTC

A modo de recapitular todos los conceptos vistos en la Sección *2.3.5 WebRTC* y en el *Anexo 9 WebRTC*, se optó por incluir un ejemplo de establecimiento de una sesión en WebRTC en forma de triángulo. La Figura 53 muestra una vista de alto nivel de cómo WebRTC establece una sesión.

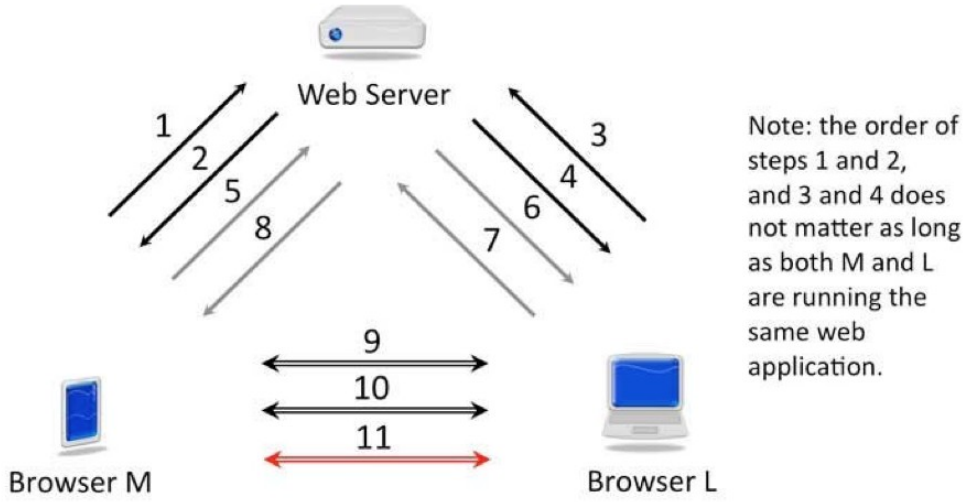

Figura 53. Establecimiento de una sesión WebRTC [25]

1. *Navegador M pide al servidor web una página web.*
2. *El servidor web provee la página web a M con JavaScript que corre código WebRTC.*
3. *El navegador L pide al servidor web una página web.*
4. *El servidor web provee la página web a L con JavaScript que corre código WebRTC.*
5. *M decide comunicarse con L. El código JavaScript en M envía una oferta al servidor web a través de su Session Description Object.*
6. *El servidor web envía el Session Description Object de M al código JavaScript corriendo en L.*
7. *El código JavaScript que corre en L envía el Session Description Object de L al servidor Web.*
8. *El servidor web envía el Session Description Object de L al código JavaScript corriendo en M.*
9. *M y L comienzan el hole punching para determinar el mejor camino para llegar de un navegador a otro.*
10. *Cuando se completa el hole punching, M y L comienzan la negociación de claves para enviarse multimedia en forma segura.*
11. *M y L comienzan a intercambiar datos multimedia.*

Una comunicación triangular en WebRTC consiste en dos navegadores que se comunican directamente a través de una Peer Connection generada por un programa JavaScript descargado del mismo servidor *web*. Este escenario es representado en la Figura 53.

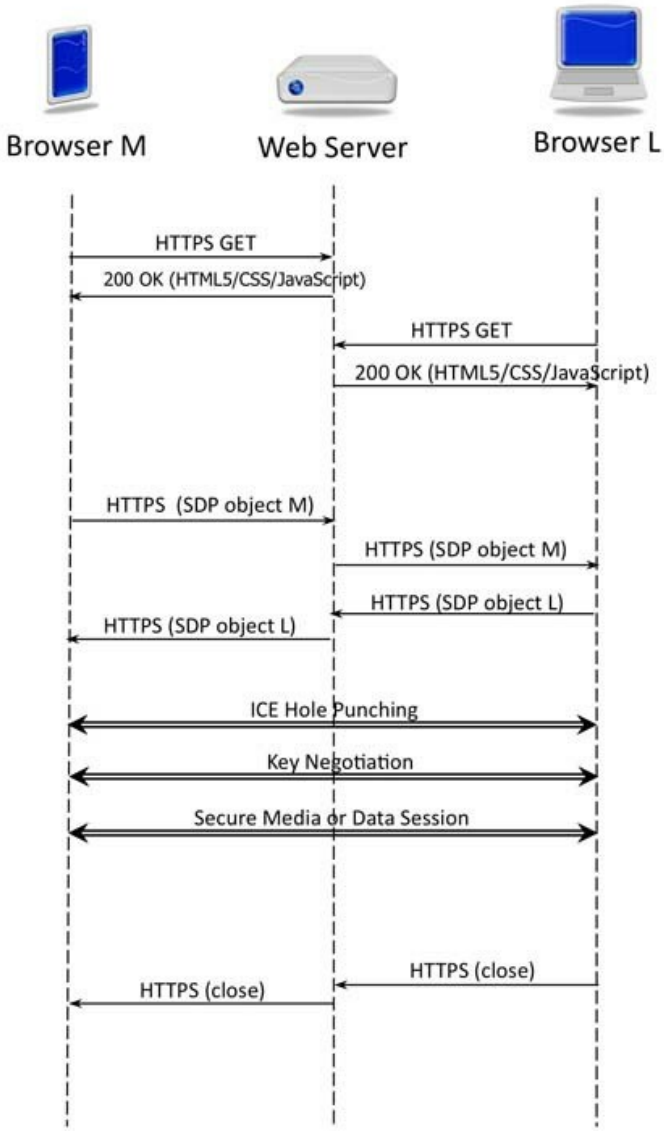

Figura 54. Flujo de una llamada triangular en WebRTC [25]

Según la Figura 53, los navegadores M y L corren la misma aplicación WebRTC descargada del servidor *web*. Cuando M desea comunicarse con L, se establece negociación *multimedia* entre los navegadores, a través del intercambio de *offer*/*answer*. Esta negociación se lleva a cabo mediante el envío de mensajes HTTP con datos en formato SDP, la comunicación *offer*/*answer* continúa entre los dos navegadores hasta que llegan a un acuerdo. Todos estos mensajes son transparentes al desarrollador, quien simplemente establece algunos parámetros de configuración.

Cada mensaje SDP describe un formato de reproducción. Para el ejemplo, el navegador M envía en forma de “oferta” un mensaje SDP al navegador L, describiendo qué tipos de formatos y capacidades soporta y en base a eso qué tipo de sesión le gustaría establecer. El navegador L en forma de “respuesta”, anuncia cuáles de las capacidades ofrecidas son soportadas y aceptadas para esa sesión.

Por último, M acepta o no las condiciones del L. Este proceso puede ser repetido varias veces para iniciar o modificar una sesión. Por lo tanto, el resultado de la negociación

termina siendo un acuerdo entre M y L sobre qué tipo de formato un cliente va a enviar y qué tipo de formato el otro peer espera recibir.

Como se muestra en la Figura 54, una vez que dicha negociación se resuelve, se inicia la etapa de Hole Punching para resolver el NAT Traversal a utilizando el protocolo ICE. Si esta etapa es exitosa, se intercambian claves para el intercambio seguro de datos. Luego de esto ambas entidades comienzan intercambiar flujos *multimedia*. Finalmente, tanto M como L pueden decidir cerrar la conexión con su contraparte.

9.7 Debates actuales sobre WebRTC

Dado que WebRTC es un proyecto que está en continuo desarrollo, muchos de los componentes que hoy en día están implementados de cierta manera son propensos a ser modificados. Existen un número de decisiones que no son finales con respecto al diseño del sistema o sobre las dependencias con ciertos protocolos. Existen cadenas de correo con discusiones sobre temas de lo más diversos y los borradores de algunas especificaciones son cambiadas frecuentemente. A continuación se detallan algunas de los debates que están teniendo lugar [25].

9.7.1 Utilización de SDP

Existen preocupaciones acerca del uso de SDP como el medio para generar acuerdos de conexiones *multimedia*. Mientras que algunos argumentan que es el navegador quien debería manejar todos los detalles, otros creen que las aplicaciones del “mundo real” necesitarán modificar esas representaciones para lograr un control preciso del comportamiento de la interacción. Además, se argumenta que las implementaciones de SDP realizadas en el navegador pueden diferir con la interpretación de los desarrolladores de aplicación, afectando el comportamiento.

El desafío de delegar esas decisiones a la aplicación es no alejar al público objetivo del *framework*: los desarrolladores *web*, un grupo que habitualmente no tiene experiencia con SDP. Esto puede presentar una barrera de aceptación desde la comunidad.

9.7.2 Soporte en diferentes navegadores

Existe el riesgo de que no todos los navegadores soporten WebRTC. En particular, Apple (creador de Safari) no se ha involucrado activamente en la creación del estándar, haciendo difícil de predecir si Safari eventualmente será capaz de soportar el *framework*.

Si bien Microsoft ha colaborado en la definición del estándar, se han manifestado preocupaciones acerca del rumbo que se está tomando, en particular acerca de la utilización de SDP. Por lo tanto, tampoco es fácil predecir el nivel de soporte que proveerá el navegador Internet Explorer.

9.7.3 Soporte de *codecs*

Hay un continuo desacuerdo en relación a los *codecs* que deberían ser soportados obligatoriamente por los navegadores que soporten WebRTC. Para garantizar un mínimo de interoperabilidad de audio y video entre dos nodos, ambas implementaciones deben soportar por lo menos un *codec* de audio y uno de video en común.

Para garantizar esto, la idea es que cada navegador soporte una lista de *codecs* de audio y video por defecto. De todas formas, hay una disputa para definir dicha lista y varias discusiones se están llevando a cabo en distintos grupos de desarrolladores para determinar las mejores opciones.

10 *Framework* de *Testing*

El *framework* de *testing* se generó con el objetivo de lograr ejecutar los casos de prueba de manera repetible y eliminando la mayor cantidad de error posible. Realizar las pruebas de manera enteramente manual presenta muchas complicaciones operacionales, por lo que un *framework* de estas características resulta adecuado. La construcción de este sistema brinda las siguientes ventajas:

- Definir los casos de prueba en archivos de configuración reutilizables.
- Correr y detener la ejecución de un determinado escenario de prueba de manera centralizada con un sólo comando.
- Controlar computadoras de manera remota, no solo a través de una LAN sino también incluso a través de Internet.
- Sincronizar los tiempos de ingreso al sistema de los *peers* remotos con un error mínimo.
- Habilidad de simular un *peer* en un ambiente sin interfaz gráfica, lo que brinda la posibilidad de instalar el *framework* en un *cluster* de máquinas virtuales.

El *framework* consta de los siguientes componentes: el Peer Simulator, el Gateway y el Gateway Client. Cada nodo de la infraestructura de *testing* tiene instalado uno o más de estos componentes. Las computadoras que simulan *peers* deben tener instalado el Peer Simulator, debe existir un Gateway por cada NAT utilizada en la infraestructura y finalmente la computadora que se encarga de iniciar la ejecución de los casos debe tener instalado el Gateway Client.

10.1 Peer Simulator

El Peer Simulator tiene la responsabilidad de simular uno o más *peers* en el sistema cada determinado intervalo de segundos. En términos técnicos esto se traduce a uno o más navegadores Chrome accediendo al sistema de manera automatizada. A nivel abstracto, el Peer Simulator debe ser un servidor escuchando una señal que le de la orden de levantar una cantidad X de pestañas de Chrome, una cada Y segundos. También debe tener la capacidad de dar de baja dichas pestañas mediante otra señal distinta.

Se utiliza Selenium[67] para levantar las instancias del navegador, un servidor Sinatra[68] que escucha las señales HTTP e implementa la lógica de negocio en lenguaje Ruby[69], y Redis[70] y Sidekiq[71] como cola de mensajes para implementar el asincronismo necesario que conlleva levantar instancias de Chrome en el contexto de una petición HTTP.

10.2 Gateway

El Gateway tiene la responsabilidad de hacer el *relay* de las señales para iniciar o detener la simulación de un peer a un nodo que tenga instalado el Peer Simulator. Por lo tanto, el Gateway conoce a todos los Peer Simulators que están corriendo dentro de su misma NAT. El mismo es un servidor que escucha una señal que le indica a qué Peer Simulators contactar, cuantas instancias de Chrome levantar y cada qué intervalos de tiempo hacerlo.

---

67 *Selenium*. http://www.seleniumhq.org/. Agosto de 2014.
68 *Sinatra*. http://www.sinatrarb.com/. Agosto de 2014.
69 *Ruby Programming Language*. https://www.ruby-lang.org/. Agosto de 2014
70 *Redis*. http://redis.io/. Agosto de 2014.
71 *Sidekiq*. http://sidekiq.org/. Agosto de 2014.

También tiene la capacidad de enviar un mensaje a otro Gateway, por ejemplo situado en otra NAT, a través de Internet. De esta forma dos NATs que tengan instaladas un conjunto de Peer Simulators pueden sincronizarse en la ejecución de un mismo caso de prueba a través de sus Gateways con tan solo una señal externa.

Se utiliza un servidor Sinatra, que escucha las señales HTTP e implementa la lógica de negocio en el lenguaje Ruby. También se necesita lidiar con problemas de asincronismo, por lo que también se utiliza Redis y Sidekiq como parte de la solución.

10.3 Gateway Client

El Gateway Client es un programa que tiene la capacidad de *parsear* un archivo de configuración, que es el que define a un caso de prueba. Básicamente este programa toma como entrada ese archivo y lo traduce en una petición HTTP con todos los parámetros *parseados*, para luego luego enviarlo a un Gateway. En el archivo se definen, los nodos Peers Simulator a contactar, la cantidad de pestañas a simular para cada nodo, el intervalo de tiempo entre cada contacto y los mensajes a enviar a un segundo Gateway dentro de otra NAT.

Para implementar este programa se utilizó el lenguaje Ruby.

Un flujo de uso de ejemplo posible es el siguiente:

- Se genera el archivo de configuración que define el caso de prueba.
- Se ejecuta el Gateway Client con ese archivo.
- El Gateway Client envía la señal al Gateway.
- El Gateway contacta a cada Peer Simulator utilizando los intervalos de tiempo configurados en el archivo.
- Cada Peer Simulator recibe la señal de levantar la cantidad correspondiente de pestañas de Chrome.
- Cada Peer Simulator utiliza el adaptador de Chrome de Selenium implementado en Ruby para levantar la cantidad de pestañas de Chrome solicitadas.
- El video empieza a reproducirse en cada una de las pestañas levantas, simulando *peers* en el sistema de manera distribuida.

11 Resultados de las pruebas realizadas

11.1 Porcentaje de *seeders* en el sistema

11.1.1 Cero *seeders* en el sistema

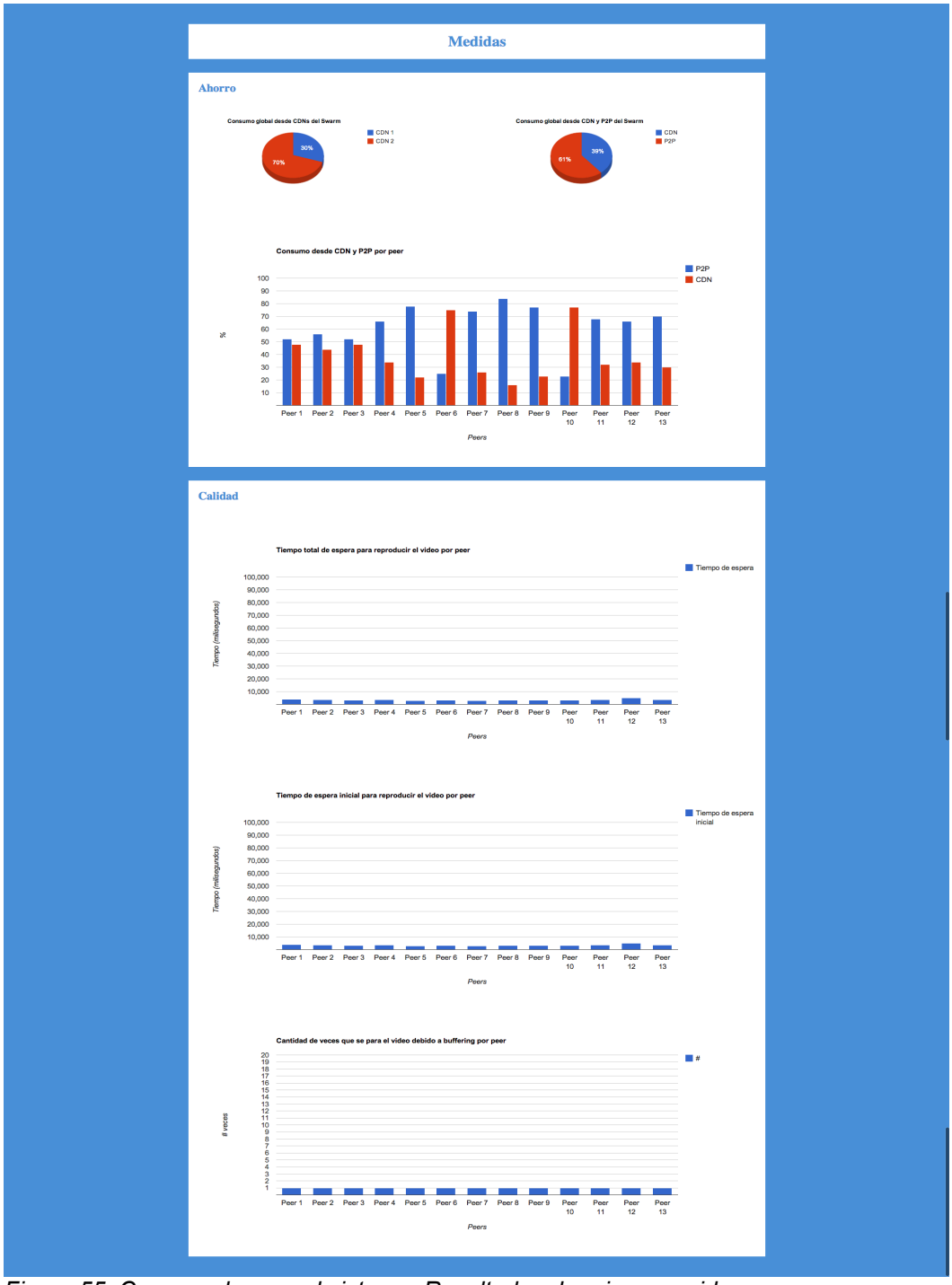

Figura 55: Cero seeders en el sistema. Resultados de primer corrida.

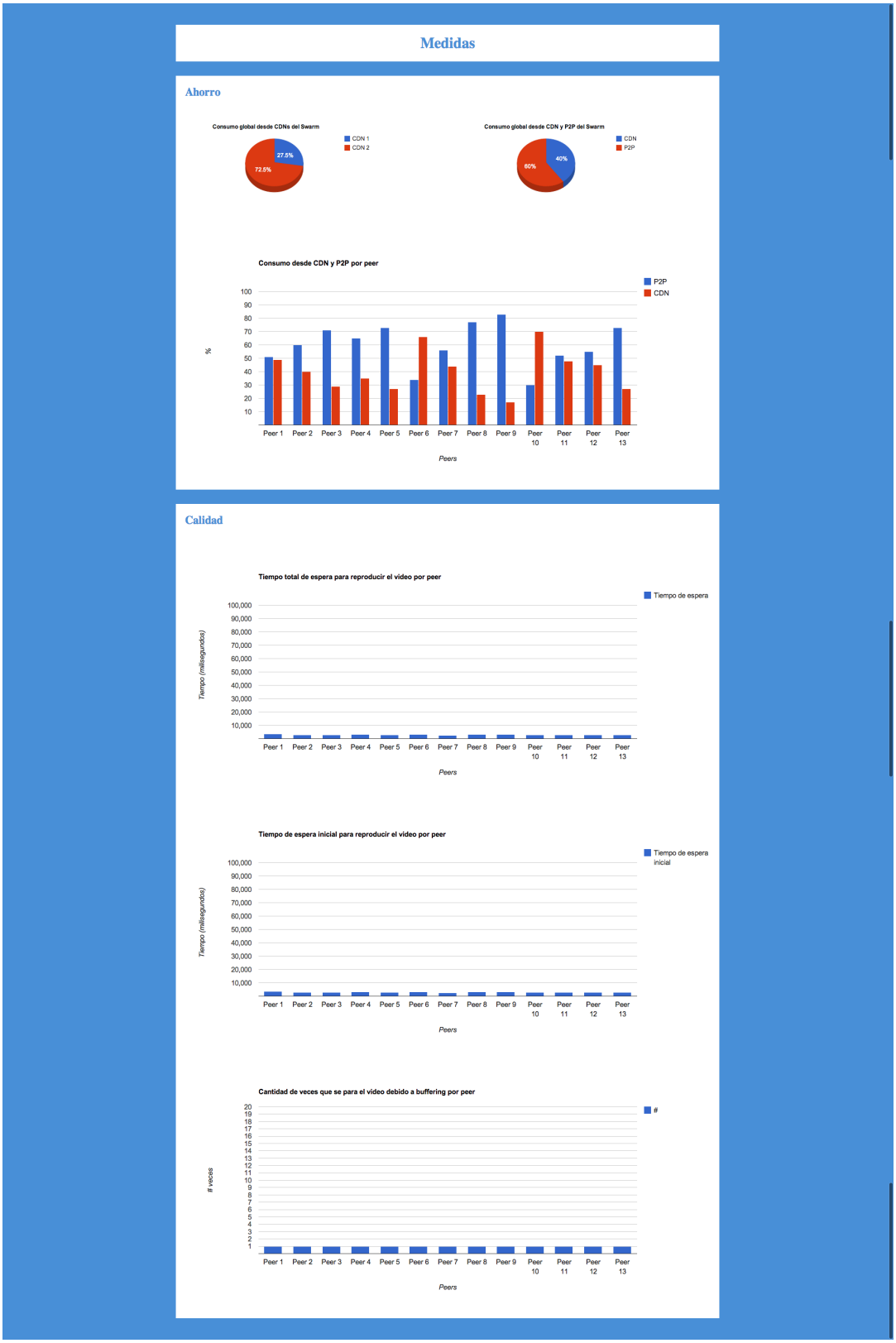

Figura 56: Cero seeders en el sistema. Resultados de segunda corrida.

Figura 57: Cero seeders en el sistema. Resultados de tercer corrida.

	General		CDNs		Peer 1		Peer 2		Peer 3		Peer 4		Peer 5		Peer 6		Peer 7		Peer 8		Peer 9		Peer 10		Peer 11		Peer 12		Peer 13		Prom.	
	P2P	CDN	CDN 1	CDN 2	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN		
Corrida #1																																
Consumo (%)	61	39	30	70	52	48	56	44	52	48	66	34	78	22	25	75	74	26	84	16	77	23	23	77	68	32	66	34	70	30	60.85	39.15
Espera total (s)					3.8		3.5		3.2		3.7		2.9		3.1		2.9		3.2		3.1		3.4		3.5		4.9		3.5		3.44	
Espera inicial (s)					3.8		3.5		3.2		3.7		2.9		3.1		2.9		3.2		3.1		3.4		3.5		4.9		3.5		3.44	
Cant. esperas					1		1		1		1		1		1		1		1		1		1		1		1		1		1	
Corrida #2																																
Consumo (%)	60	40	27.5	72.5	51	49	60	40	71	29	65	35	73	27	34	66	56	44	77	23	83	17	30	70	52	48	55	45	73	27	60	40
Espera total (s)					3.5		3		2.9		3.2		2.8		3.1		2.7		3.1		3.1		2.8		2.8		2.8		2.9		2.98	
Espera inicial (s)					3.5		3		2.9		3.2		2.8		3.1		2.7		3.1		3.1		2.8		2.8		2.8		2.9		2.98	
Cant. esperas					1		1		1		1		1		1		1		1		1		1		1		1		1		1	
Corrida #3																																
Consumo (%)	58	42	26.2	73.8	53	47	55	45	59	41	67	33	69	31	31	69	58	42	77	23	67	33	36	64	49	51	62	38	69	31	57.85	42.15
Espera total (s)					3.9		3.4		3.7		3.1		3		3.1		2.9		3.1		3.1		3.3		3.1		3.6		3.1		3.26	
Espera inicial (s)					3.9		3.4		3.7		3.1		3		3.1		2.9		3.1		3.1		3.3		3.1		3.6		3.1		3.26	
Cant. esperas					1		1		1		1		1		1		1		1		1		1		1		1		1		1	
Totales																																
P2P vs CDN	**P2P**	**CDN**																														
Consumo (%)	60	40																														
CDN1 vs CDN2	**CDN1**	**CDN2**																														
Consumo (%)	28	72																														
Calidad																																
Espera total (s)	3.2																															
Espera inicial (s)	3.2																															
Cant. esperas	1																															

Tabla 14: Cero seeders en el sistema. Resultados de las tres corridas.

11.1.2 Dos *seeders* en el sistema

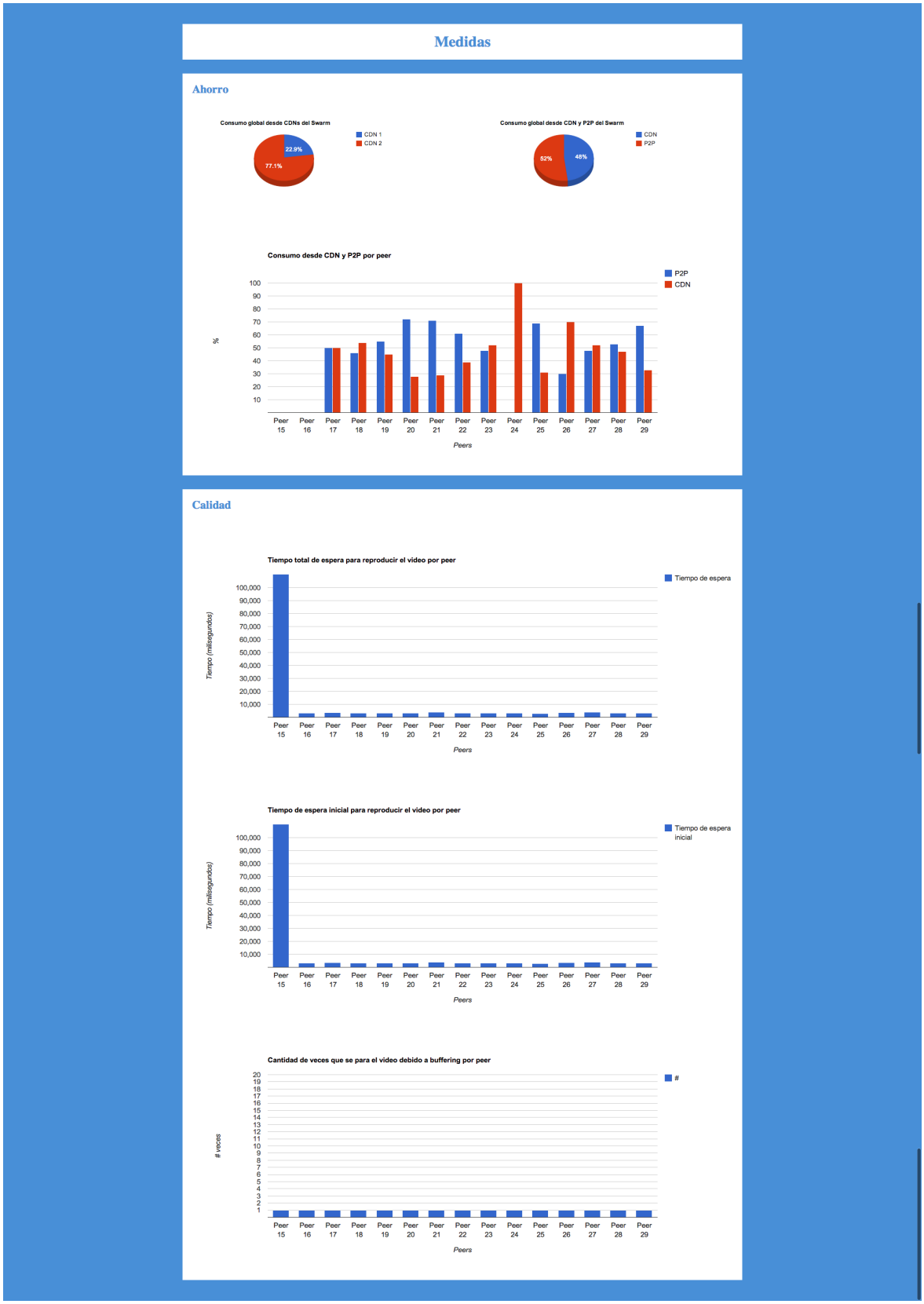

Figura 58: Dos seeders en el sistema. Resultados de primer corrida.

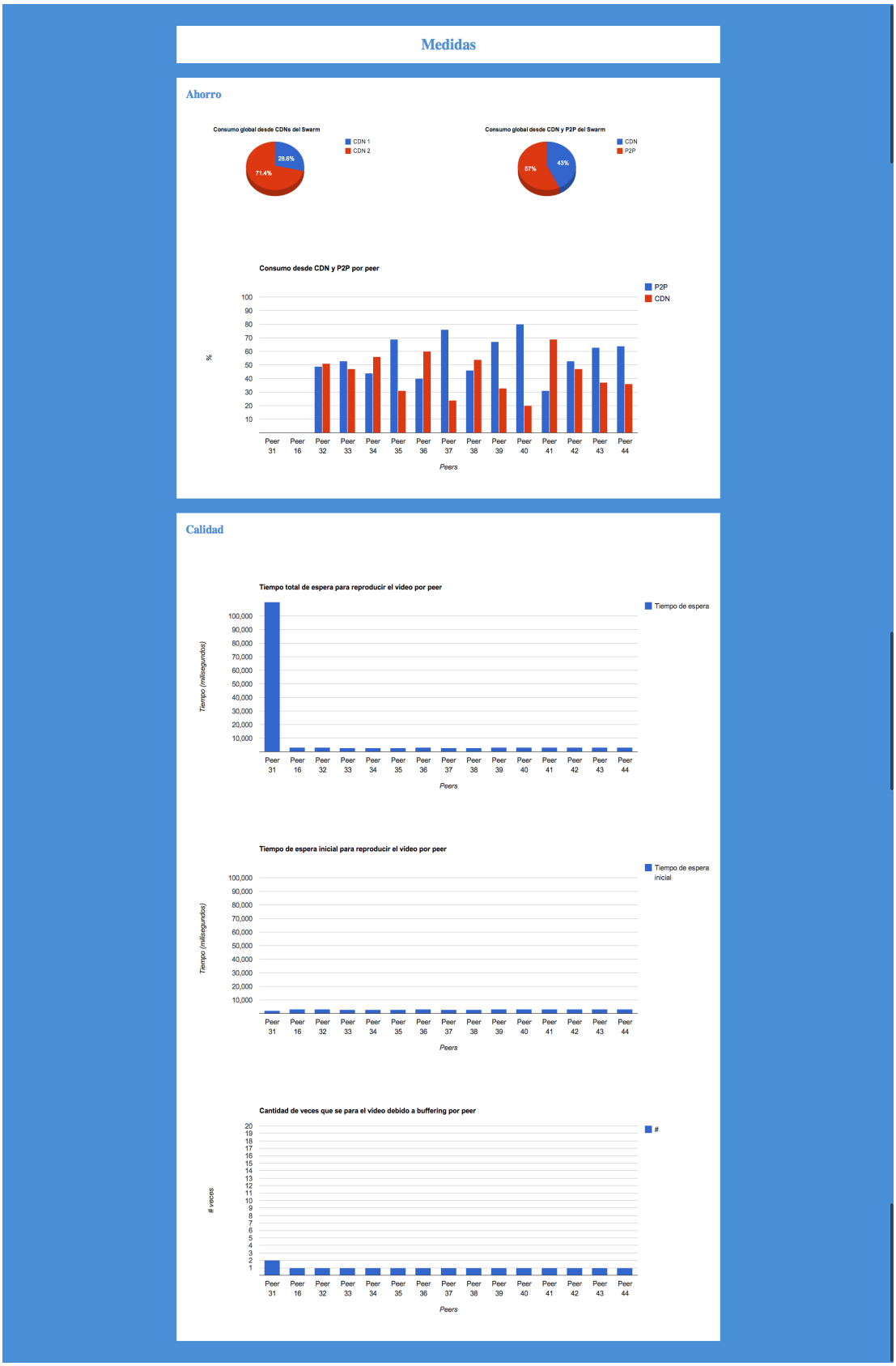

Figura 59: Dos seeders en el sistema. Resultados de tercer corrida.

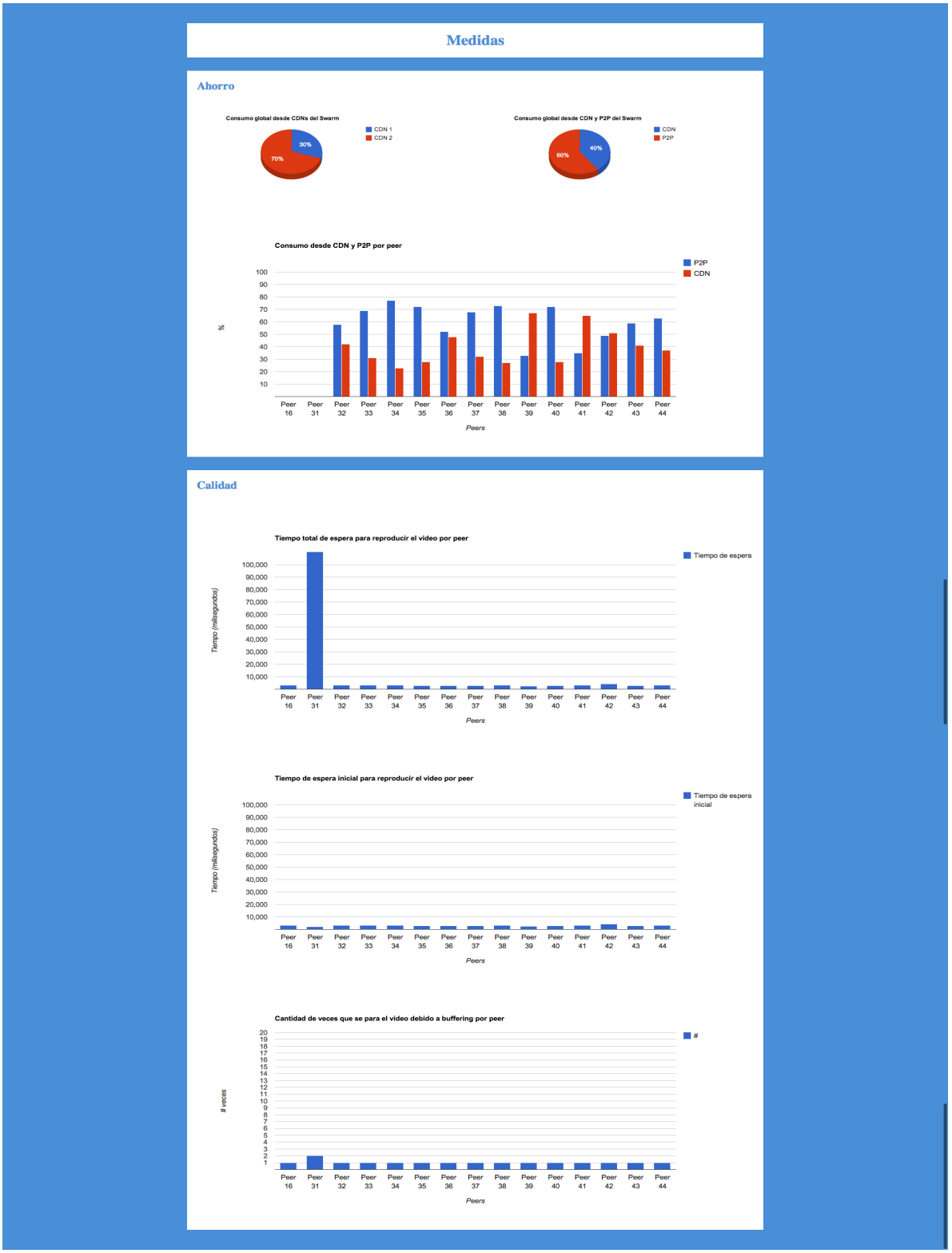

Figura 60: Dos seeders en el sistema. Resultados de segunda corrida.

	General		CDNs		Peer 1		Peer 2		Peer 3		Peer 4		Peer 5		Peer 6		Peer 7		Peer 8		Peer 9		Peer 10		Peer 11		Peer 12		Peer 13		Prom.	
	P2P	CDN	CDN 1	CDN 2	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN		
Corrida #1																																
Consumo (%)	52	48	22.9	77.1	50	50	46	54	55	45	72	28	71	29	61	39	48	52	0	100	69	31	30	70	48	52	53	47	67	33	51.54	48.46
Espera total (s)					3.7		3.3		3.3		3.3		3.9		3.1		3.1		3.1		3		3.4		3.9		3.3		3.3		3.36	
Espera inicial (s)					3.7		3.3		3.3		3.3		3.9		3.1		3.1		3.1		3		3.4		3.9		3.3		3.3		3.36	
Cant. esperas					1		1		1		1		1		1		1		1		1		1		1		1		1		1	
Corrida #2																																
Consumo (%)	60	40	30	70	58	42	69	31	77	23	72	28	52	48	68	32	73	27	33	67	72	28	35	65	49	51	59	41	63	37	60	40
Espera total (s)					3.3		3.3		3.2		3		2.8		3		3.2		2.6		3		3.2		4.2		3		3.3		3.16	
Espera inicial (s)					3.3		3.3		3.2		3		2.8		3		3.2		2.6		3		3.2		4.2		3		3.3		3.16	
Cant. esperas					1		1		1		1		1		1		1		1		1		1		1		1		1		1	
Corrida #3																																
Consumo (%)	57	43	28.6	71.4	49	51	53	47	44	56	69	31	40	60	76	24	46	54	67	33	80	20	31	69	53	47	63	37	64	36	56.54	43.46
Espera total (s)					3.2		2.8		2.8		3		3.1		3		2.9		3.1		3.3		3.2		3.3		3.3		3.3		3.1	
Espera inicial (s)					3.2		2.8		2.8		3		3.1		3		2.9		3.1		3.3		3.2		3.3		3.3		3.3		3.1	
Cant. esperas					1		1		1		1		1		1		1		1		1		1		1		1		1		1	
Totales																																
P2P vs CDN	**P2P**	**CDN**																														
Consumo (%)	56	44																														
CDN1 vs CDN2	**CDN1**	**CDN2**																														
Consumo (%)	27	73																														
Calidad																																
Espera total (s)	3.2																															
Espera inicial (s)	3.2																															
Cant. esperas	1																															

Tabla 15: Dos seeders en el sistema. Resultados de las tres corridas.

11.1.3 Cuatro *seeders* en el sistema

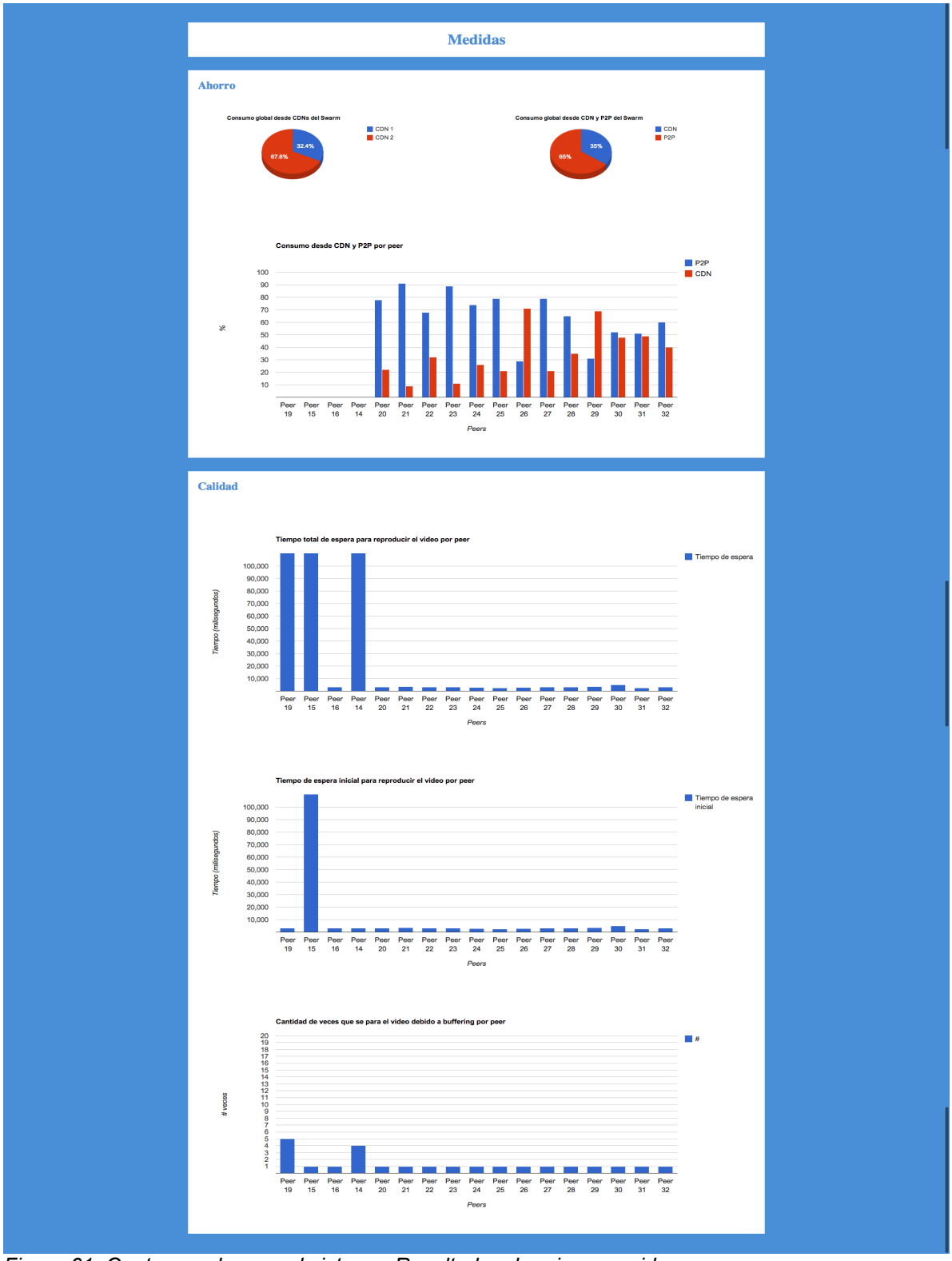

Figura 61: Cuatro seeders en el sistema. Resultados de primer corrida.

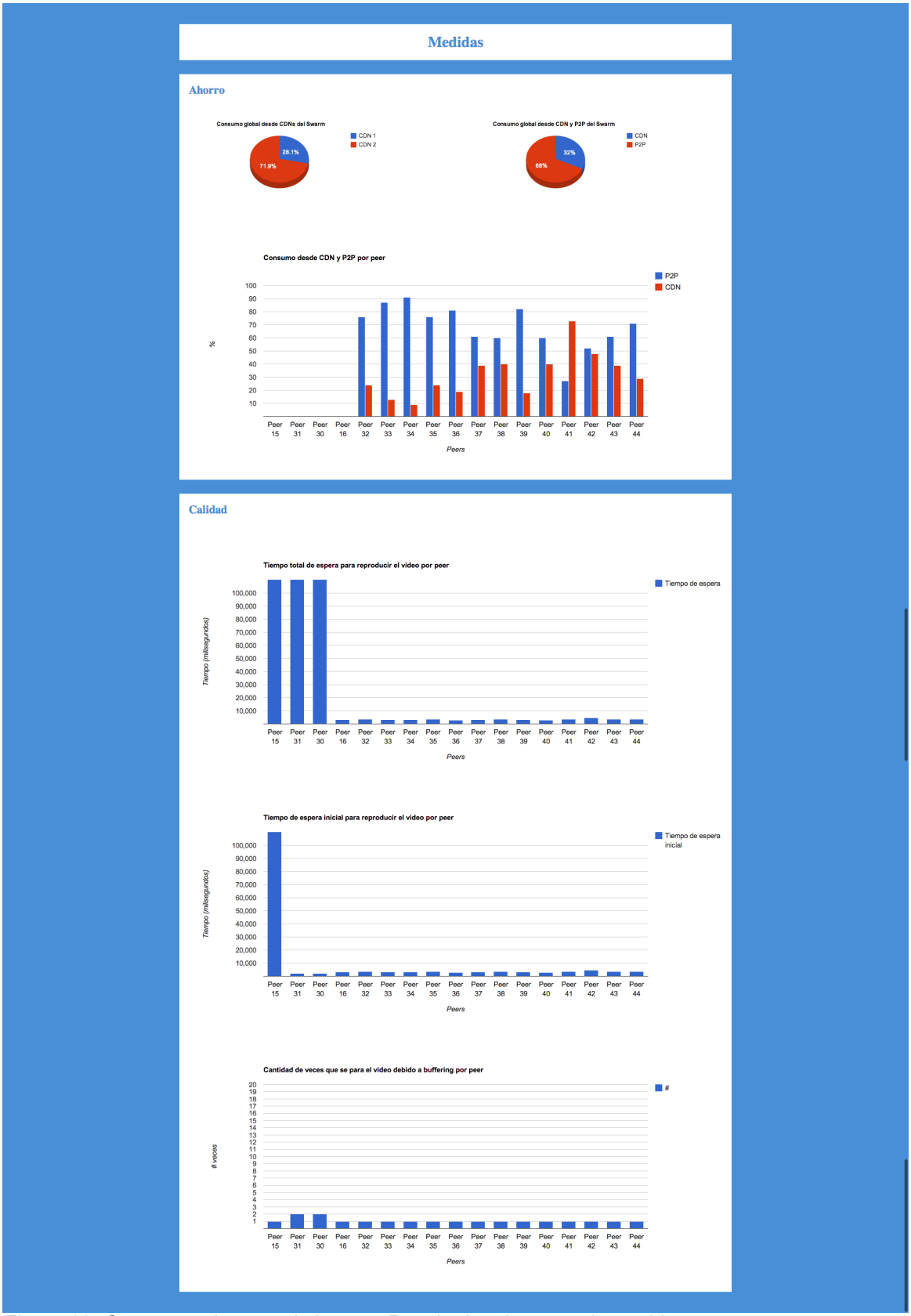

Figura 62: Cuatro seeders en el sistema. Resultados de segunda corrida.

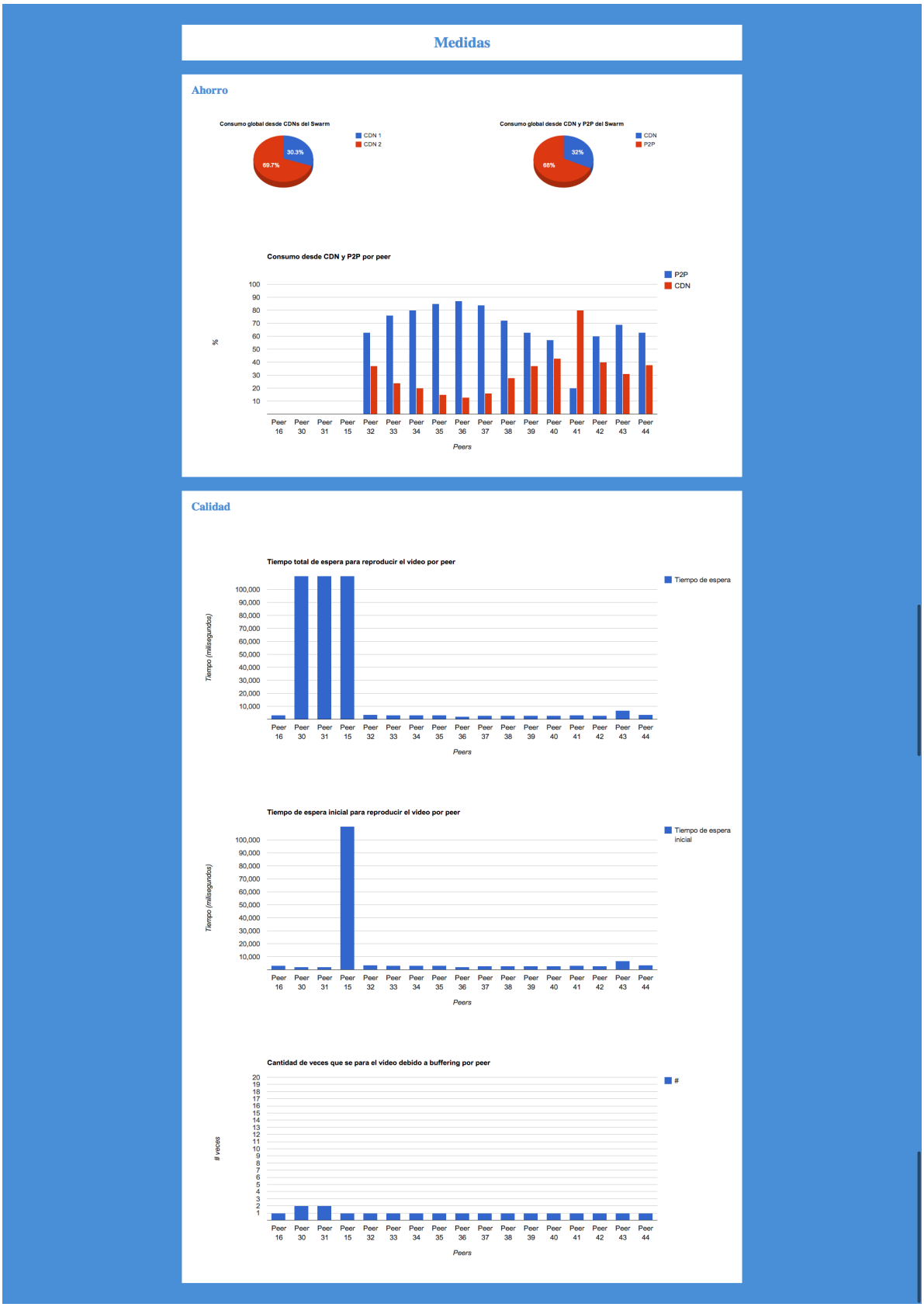

Figura 63: Cuatro seeders en el sistema. Resultados de tercer corrida.

	General		CDNs		Peer 1		Peer 2		Peer 3		Peer 4		Peer 5		Peer 6		Peer 7		Peer 8		Peer 9		Peer 10		Peer 11		Peer 12		Peer 13		Prom.	
	P2P	CDN	CDN 1	CDN 2	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN		
Corrida #1																																
Consumo (%)	65	35	32.4	67.7	78	22	91	21	68	32	89	11	74	24	79	21	29	71	79	29	65	35	31	69	58	48	51	49	60	40	65.54	36.31
Espera total (s)						3.3		3.4		3.4		3.4		2.9		2.5		2.8		3.3		3.3		3.4		4.9		2.3		3.3		3.25
Espera inicial (s)						3.3		3.4		3.4		3.4		2.9		2.5		2.8		3.3		3.3		3.4		4.9		2.3		3.3		3.25
Cant. esperas						1		1		1		1		1		1		1		1		1		1		1		1		1		1
Corrida #2																																
Consumo (%)	68	32	28.1	71.9	76	24	87	13	91	9	76	24	81	19	61	39	60	40	82	18	60	40	28	72	51	49	59	41	72	28	68	32
Espera total (s)						3.4		3.4		3		3.4		3.3		3.1		3.6		3.1		3		3.6		4.6		3.7		3.5		3.44
Espera inicial (s)						3.4		3.4		3		3.4		3.3		3.1		3.6		3.1		3		3.6		4.6		3.7		3.5		3.44
Cant. esperas						1		1		1		1		1		1		1		1		1		1		1		1		1		1
Corrida #3																																
Consumo (%)	68	32	30.3	69.7	63	37	76	24	80	20	85	15	87	13	84	16	72	28	63	37	57	43	20	80	60	40	69	31	63	37	67.62	32.38
Espera total (s)						3.6		3.3		3.1		3.3		2.2		3		2.8		2.9		3		3.4		3		6.7		3.5		3.37
Espera inicial (s)						3.6		3.3		3.1		3.3		2.2		3		2.8		2.9		3		3.4		3		6.7		3.5		3.37
Cant. esperas						1		1		1		1		1		1		1		1		1		1		1		1		1		1
Totales																																
P2P vs CDN	**P2P**	**CDN**																														
Consumo (%)	67	34																														
CDN1 vs CDN2	**CDN1**	**CDN2**																														
Consumo (%)	30	70																														
Calidad																																
Espera total (s)		3.4																														
Espera inicial (s)		3.4																														
Cant. esperas		1																														

Tabla 16: Cuatro seeders en el sistema. Resultados de las tres corridas.

11.2 Distribución de ingreso

11.2.1 *Ramp up* cero

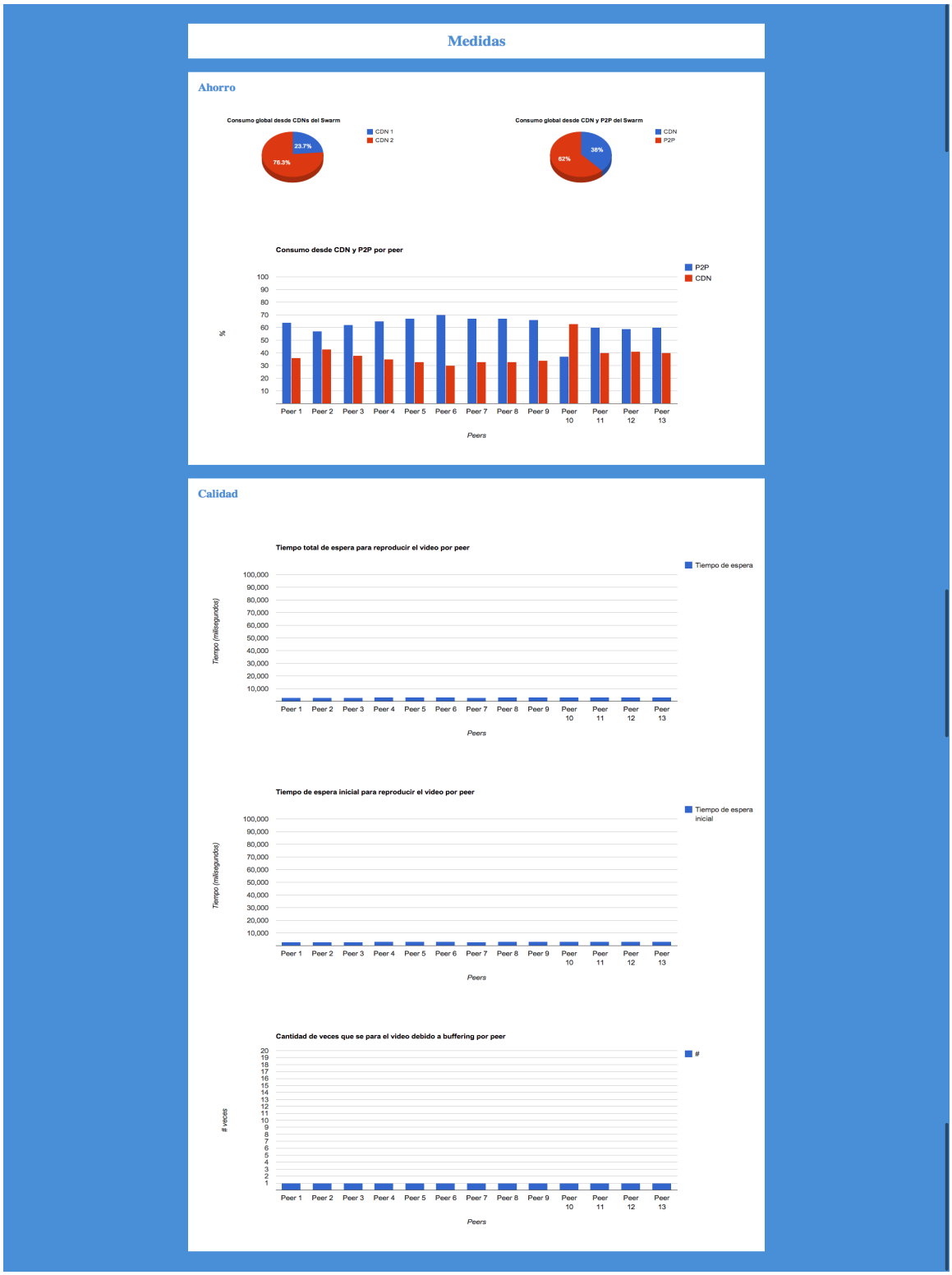

Figura 64: Ramp up cero. Resultados de primer corrida.

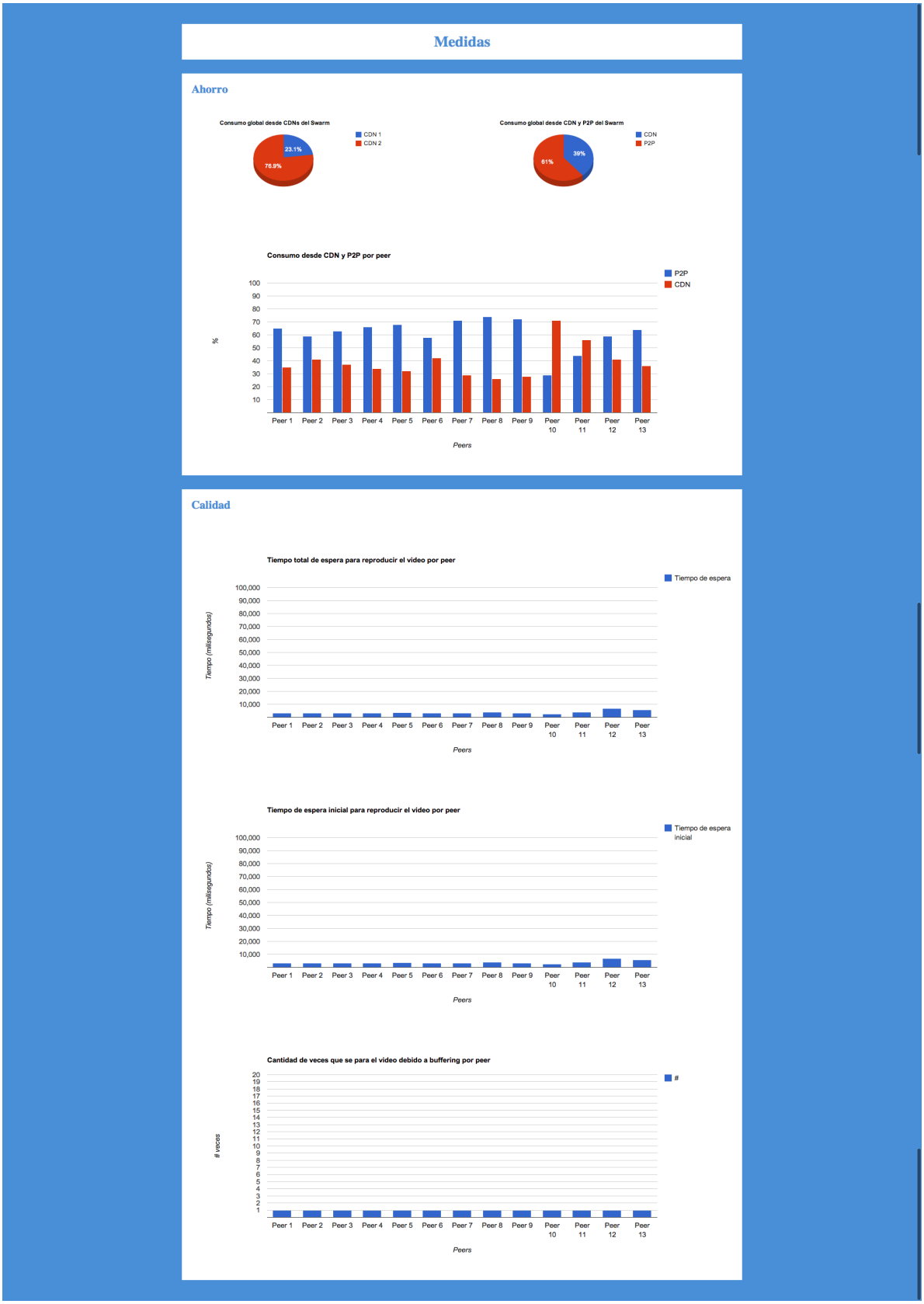

Figura 65: Ramp up cero. Resultados de segunda corrida.

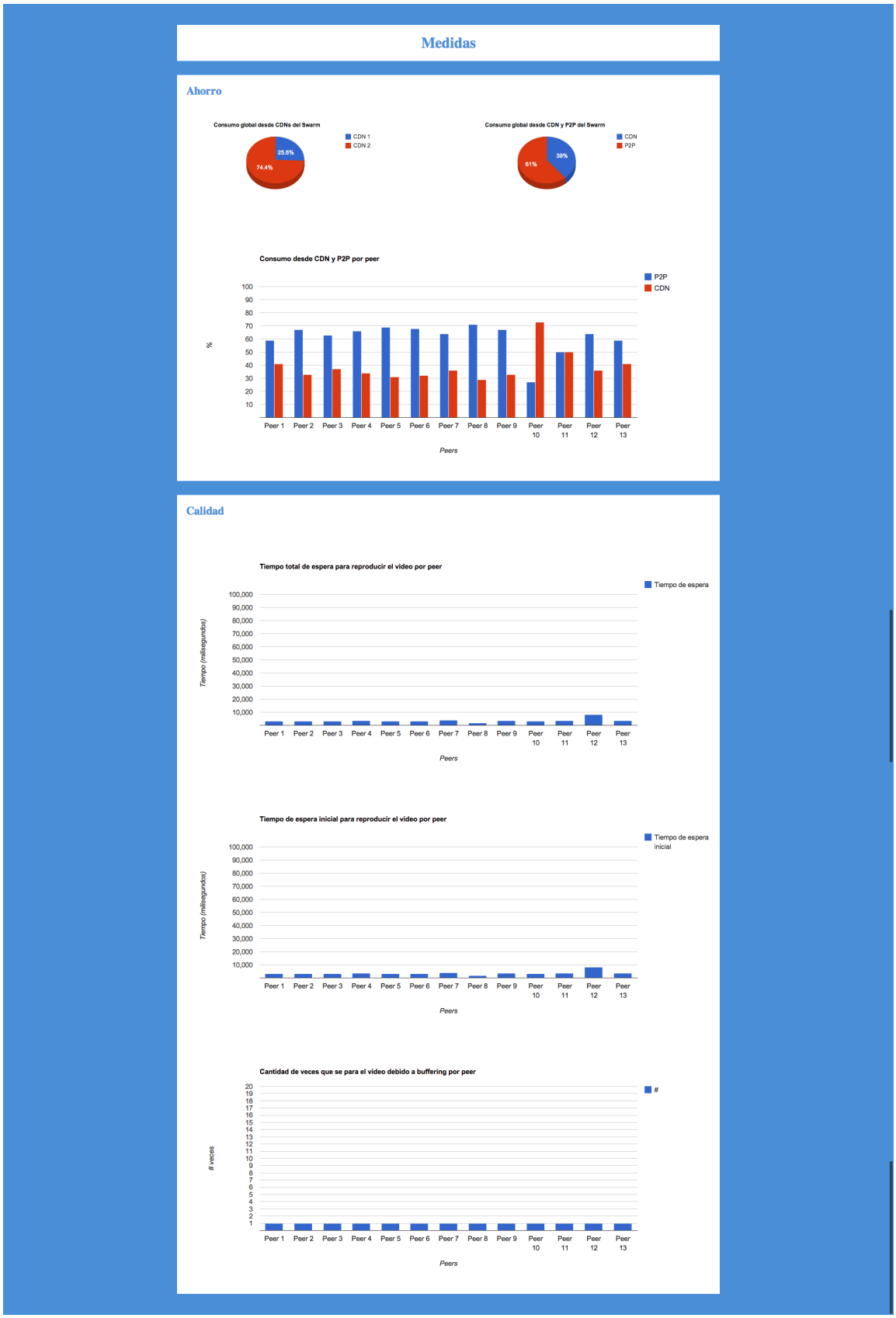

Figura 66: Ramp up cero. Resultados de tercer corrida.

	General			CDNs			Peer 1		Peer 2		Peer 3		Peer 4		Peer 5		Peer 6		Peer 7		Peer 8		Peer 9		Peer 10		Peer 11		Peer 12		Peer 13			Prom.	
	P2P	CDN		CDN 1	CDN 2		P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN			
Corrida #1																																			
Consumo (%)	62	38		23.7	76.3		64	36	57	43	62	38	65	35	67	33	70	30	67	33	67	33	66	34	37	63	60	40	59	41	60	40		61.62	38.38
Espera total (s)							2.9		3		2.9		3.1		3.1		3.1		2.8		3.1		3		3.3		3.2		3.2		3.4			3.08	
Espera inicial (s)							2.9		3		2.9		3.1		3.1		3.1		2.8		3.1		3		3.3		3.2		3.2		3.4			3.08	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #2																																			
Consumo (%)	61	39		23.1	76.9		65	35	59	41	63	37	66	34	68	32	58	42	71	29	74	26	72	28	29	71	44	56	59	41	64	36		60.92	39.08
Espera total (s)							3.1		3.3		3.2		3.2		3.4		3.1		3.3		3.9		3.1		2.3		4.1		6.7		5.8			3.73	
Espera inicial (s)							3.1		3.3		3.2		3.2		3.4		3.1		3.3		3.9		3.1		2.3		4.1		6.7		5.8			3.73	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #3																																			
Consumo (%)	61	39		25.6	74.4		59	41	67	33	63	37	66	34	69	31	68	32	64	36	71	29	67	33	27	73	50	50	64	36	59	41		61.08	38.92
Espera total (s)							3.2		3.3		3.3		3.4		3.1		3		3.8		1.9		3.6		3.2		3.4		8.2		3.5			3.61	
Espera inicial (s)							3.2		3.3		3.3		3.4		3.1		3		3.8		1.9		3.6		3.2		3.4		8.2		3.5			3.61	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Totales																																			
P2P vs CDN	**P2P**	**CDN**																																	
Consumo (%)	61	39																																	
CDN1 vs CDN2	**CDN1**	**CDN2**																																	
Consumo (%)	24	76																																	
Calidad																																			
Espera total (s)	3.5																																		
Espera inicial (s)	3.5																																		
Cant. esperas	1																																		

Tabla 17: Ramp up cero. Resultados de las tres corridas.

11.2.2 *Ramp up* constante

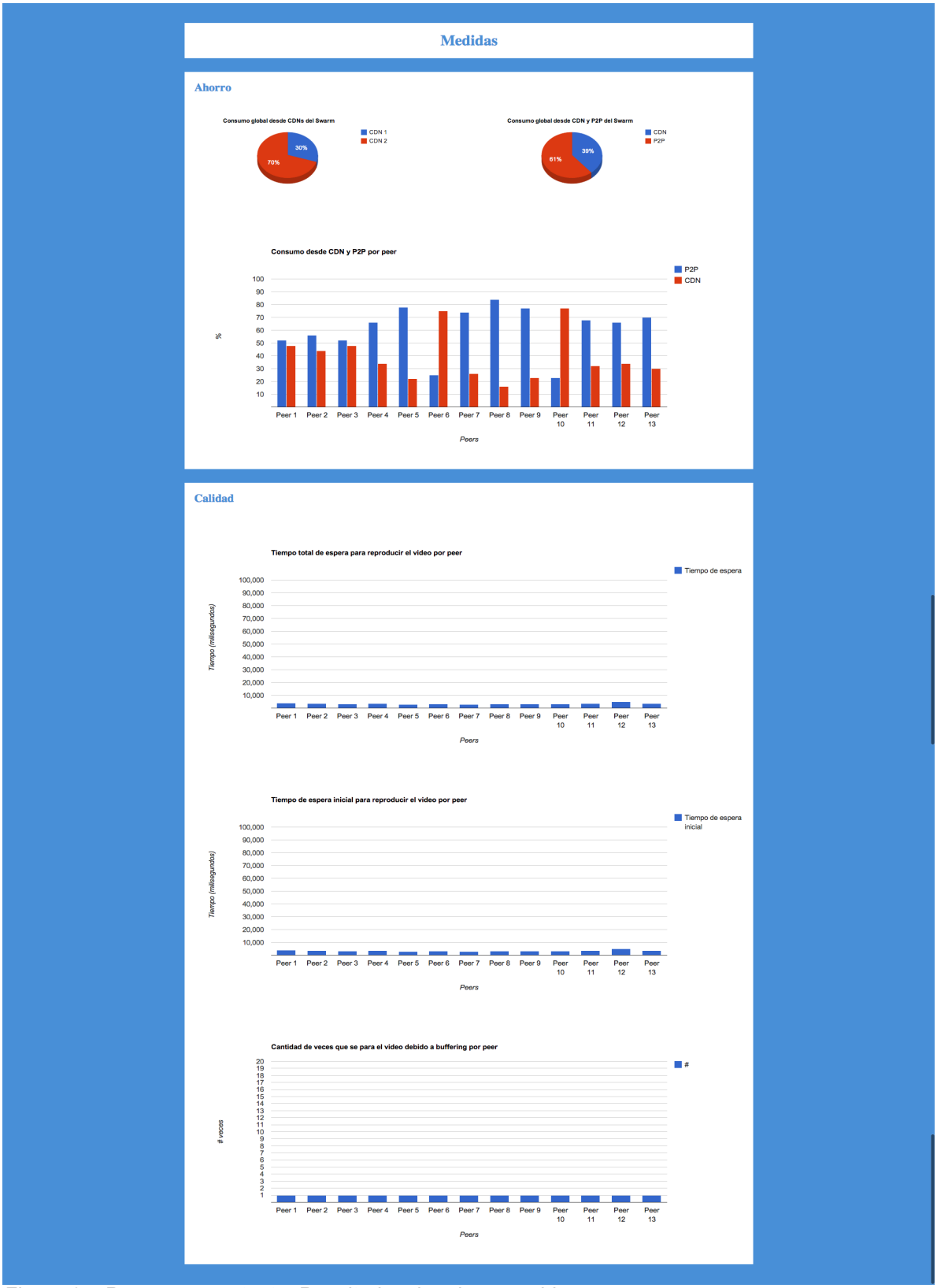

Figura 67: Ramp up constante. Resultados de primer corrida.

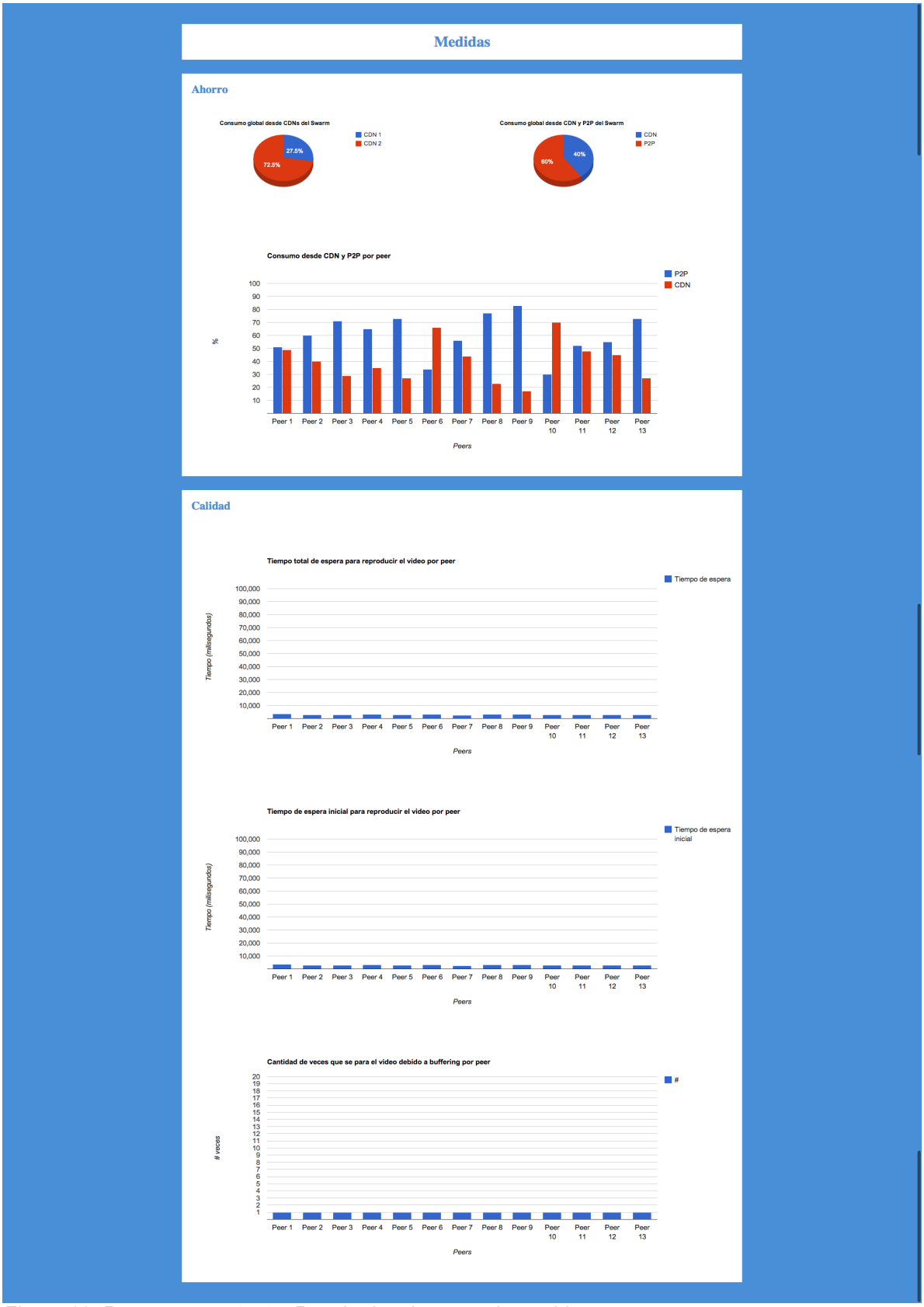

Figura 68: Ramp up constante. Resultados de segunda corrida.

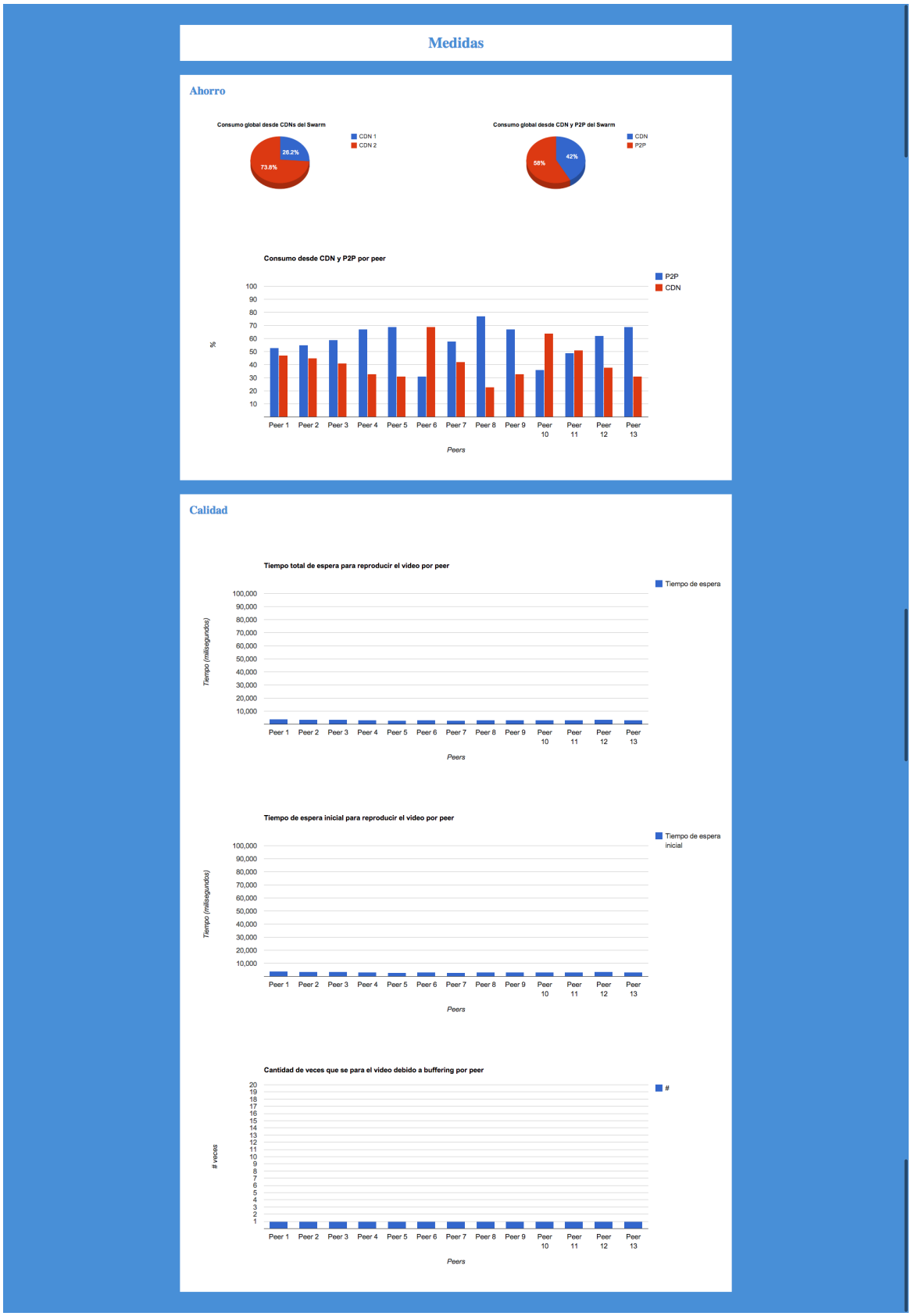

Figura 69: Ramp up constante. Resultados de tercer corrida.

	General			CDNs			Peer 1		Peer 2		Peer 3		Peer 4		Peer 5		Peer 6		Peer 7		Peer 8		Peer 9		Peer 10		Peer 11		Peer 12		Peer 13			Prom.	
	P2P	CDN		CDN 1	CDN 2		P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN			
Corrida #1																																			
Consumo (%)	61	39		30	70		52	48	56	44	52	48	66	34	78	22	25	75	74	26	84	16	77	23	23	77	68	32	66	34	70	30		60.85	39.15
Espera total (s)							3.8		3.5		3.2		3.7		2.9		3.1		2.9		3.2		3.1		3.4		3.5		4.9		3.5			3.44	
Espera inicial (s)							3.8		3.5		3.2		3.7		2.9		3.1		2.9		3.2		3.1		3.4		3.5		4.9		3.5			3.44	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #2																																			
Consumo (%)	60	40		27.5	72.5		51	49	60	40	71	29	65	35	73	27	34	66	56	44	77	23	83	17	30	70	52	48	55	45	73	27		60	40
Espera total (s)							3.5		3		2.9		3.2		2.8		3.1		2.7		3.1		3.1		2.8		2.8		2.8		2.9			2.98	
Espera inicial (s)							3.5		3		2.9		3.2		2.8		3.1		2.7		3.1		3.1		2.8		2.8		2.8		2.9			2.98	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #3																																			
Consumo (%)	58	42		26.2	73.8		53	47	55	45	59	41	67	33	69	31	31	69	58	42	77	23	67	33	36	64	49	51	62	38	69	31		57.85	42.15
Espera total (s)							3.9		3.4		3.7		3.1		3		3.1		2.9		3.1		3.1		3.3		3.1		3.6		3.1			3.26	
Espera inicial (s)							3.9		3.4		3.7		3.1		3		3.1		2.9		3.1		3.1		3.3		3.1		3.6		3.1			3.26	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Totales																																			
P2P vs CDN	**P2P**	**CDN**																																	
Consumo (%)	60	40																																	
CDN1 vs CDN2	**CDN1**	**CDN2**																																	
Consumo (%)	28	72																																	
Calidad																																			
Espera total (s)	3.2																																		
Espera inicial (s)	3.2																																		
Cant. esperas	1																																		

Tabla 18: Ramp up constante. Resultados de las tres corridas.

11.2.3 *Ramp up* random

Figura 70: Ramp up random. Resultados de primer corrida.

Figura 71: Ramp up random. Resultados de segunda corrida.

Figura 72: Ramp up random. Resultados de tercer corrida.

	General		CDNs		Peer 1		Peer 2		Peer 3		Peer 4		Peer 5		Peer 6		Peer 7		Peer 8		Peer 9		Peer 10		Peer 11		Peer 12		Peer 13		Prom.	
	P2P	CDN	CDN 1	CDN 2	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN		
Corrida #1																																
Consumo (%)	54	46	30.4	69.6	0	100	30	70	63	37	71	29	71	29	57	43	52	48	66	34	20	80	60	40	64	36	64	36	82	18	53.85	46.15
Espera total (s)					3.6		3.5		3.2		3.1		3.2		2.9		3.1		3		4.5		3.1		3		7.9		3.1		3.63	
Espera inicial (s)					3.6		3.5		3.2		3.1		3.2		2.9		3.1		3		4.5		3.1		3		7.9		3.1		3.63	
Cant. esperas					1		1		1		1		1		1		1		1		1		1		1		1		1		1	
Corrida #2																																
Consumo (%)	50	50	29.4	70.6	0	100	9	91	57	43	58	42	76	24	67	33	60	40	70	30	42	58	20	80	63	37	61	39	62	38	49.62	50.38
Espera total (s)					3.6		1.9		2.9		3		3.1		3.1		3		3.2		23.9		3.5		2.8		2.8		22.2		6.08	
Espera inicial (s)					3.6		1.9		2.9		3		3.1		3.1		3		3.2		23.9		3.5		2.8		2.8		22.2		6.08	
Cant. esperas					1		1		1		1		1		1		1		1		1		1		1		1		1		1	
Corrida #3																																
Consumo (%)	53	47	39.1	60.9	32	68	62	38	60	40	59	41	70	30	68	32	58	42	37	63	57	43	18	82	49	51	68	32	56	44	53.38	46.62
Espera total (s)					3.6		3		3.3		3		1.8		3.1		3		3.4		2.4		3.3		3		4.4		2.9		3.09	
Espera inicial (s)					3.6		3		3.3		3		1.8		3.1		3		3.4		2.4		3.3		3		4.4		2.9		3.09	
Cant. esperas					1		1		1		1		1		1		1		1		1		1		1		1		1		1	
Totales																																
P2P vs CDN	**P2P**	**CDN**																														
Consumo (%)	52	48																														
CDN1 vs CDN2	**CDN1**	**CDN2**																														
Consumo (%)	33	67																														
Calidad																																
Espera total (s)	4.3																															
Espera inicial (s)	4.3																															
Cant. esperas	1																															

Tabla 19: Ramp up aleatorio. Resultados de las tres corridas.

11.3 Ráfagas de *peers*

11.3.1 Ráfagas de dos *peers*

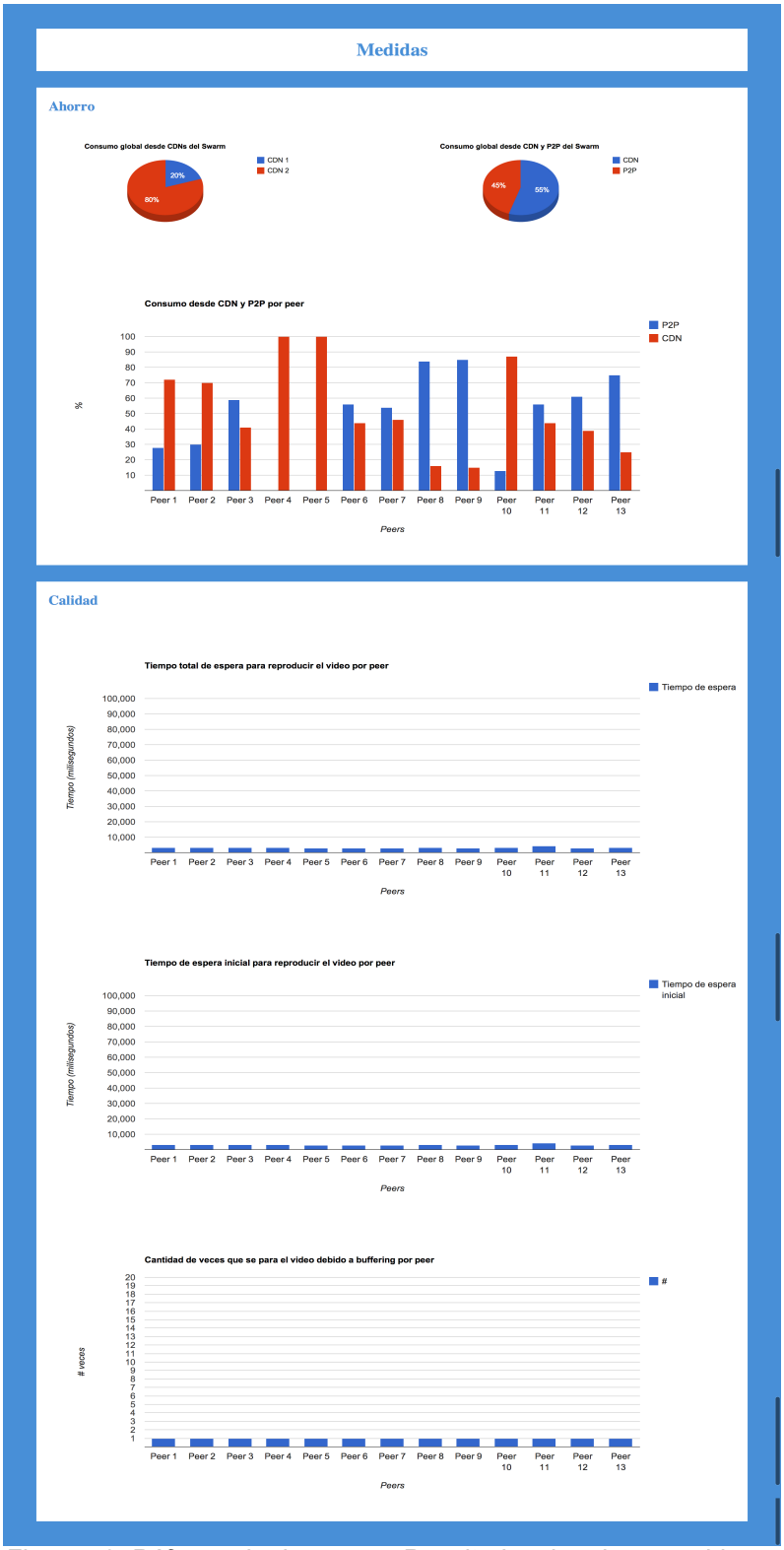

Figura 73: Ráfagas de dos peers. Resultados de primer corrida.

Figura 74: Ráfagas de dos peers. Resultados de segunda corrida.

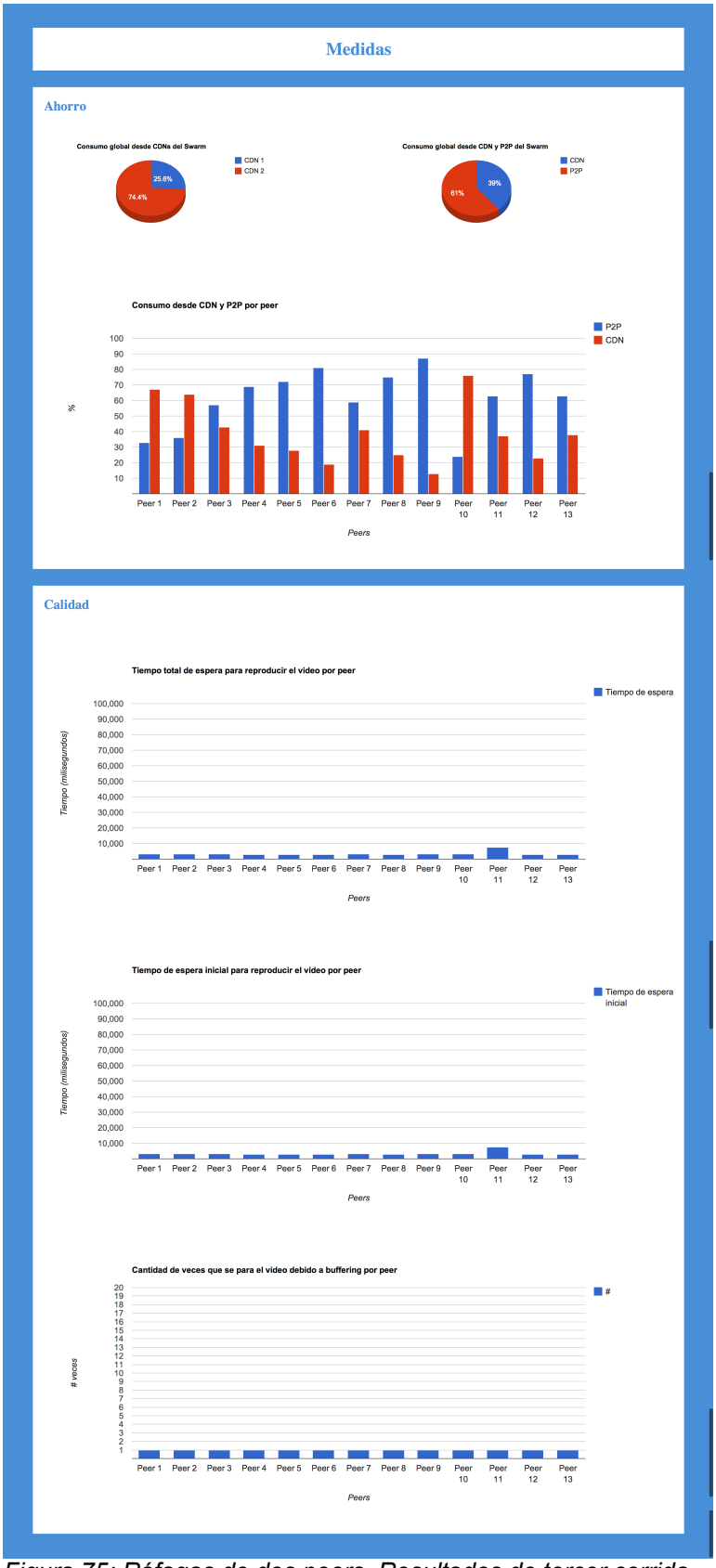

Figura 75: Ráfagas de dos peers. Resultados de tercer corrida.

	General			CDNs			Peer 1		Peer 2		Peer 3		Peer 4		Peer 5		Peer 6		Peer 7		Peer 8		Peer 9		Peer 10		Peer 11		Peer 12		Peer 13			Prom.	
	P2P	CDN		CDN 1	CDN 2		P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN			
Corrida #1																																			
Consumo (%)	45	55		20	80		28	72	30	70	59	41	0	100	0	100	56	44	54	46	84	16	85	15	12	88	57	43	62	38	68	32		45.77	54.23
Espera total (s)							3.3		3.3		3		3.2		3		2.9		3		3		3		3.3		4.2		2.9		4.1			3.25	
Espera inicial (s)							3.3		3.3		3		3.2		3		2.9		3		3		3		3.3		4.2		2.9		4.1			3.25	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #2																																			
Consumo (%)	59	41		26.8	73.2		34	66	37	63	57	43	72	28	85	15	56	44	60	40	75	25	78	22	9	91	67	33	70	30	63	37		58.69	41.31
Espera total (s)							3.2		3.3		3.4		3		3		3		3.1		3.1		3.3		3.2		10.1		2.8		2.9			3.65	
Espera inicial (s)							3.2		3.3		3.4		3		3		3		3.1		3.1		3.3		3.2		10.1		2.8		2.9			3.65	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #3																																			
Consumo (%)	61	39		25.6	74.4		33	67	36	64	57	43	69	31	72	28	81	19	59	41	75	25	87	13	24	76	63	37	77	23	63	38		61.23	38.85
Espera total (s)							3.1		3.1		3.2		3		2.9		3		3		2.8		3.2		3.2		7.4		2.9		2.9			3.36	
Espera inicial (s)							3.1		3.1		3.2		3		2.9		3		3		2.8		3.2		3.2		7.4		2.9		2.9			3.36	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Totales																																			
P2P vs CDN	**P2P**	**CDN**																																	
Consumo (%)	55	45																																	
CDN1 vs CDN2	**CDN1**	**CDN2**																																	
Consumo (%)	24	76																																	
Calidad																																			
Espera total (s)	3.4																																		
Espera inicial (s)	3.4																																		
Cant. esperas	1																																		

Tabla 20: Ráfagas de dos peers. Resultados de las tres corridas.

11.3.2 Ráfagas de cuatro *peers*

Figura 76: Ráfagas de cuatro peers. Resultados de primer corrida.

Figura 77: Ráfagas de cuatro peers. Resultados de segunda corrida.

Figura 78: Ráfagas de cuatro peers. Resultados de tercer corrida.

	General			CDNs			Peer 1		Peer 2		Peer 3		Peer 4		Peer 5		Peer 6		Peer 7		Peer 8		Peer 9		Peer 10		Peer 11		Peer 12		Peer 13			Prom.	
	P2P	CDN		CDN 1	CDN 2		P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN			
Corrida #1																																			
Consumo (%)	57	43		30.2	69.8		54	46	54	46	50	50	52	48	67	33	73	27	62	38	63	37	73	27	41	57	41	59	41	59	67	33		56.77	43.08
Espera total (s)							3.3		3.2		3.1		3.1		3.2		3		2.9		12.6		3		3.2		3		3.6		4			3.94	
Espera inicial (s)							3.3		3.2		3.1		3.1		3.2		3		2.9		12.6		3		3.2		3		3.6		4			3.94	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #2																																			
Consumo (%)	57	43		25.6	74.4		51	49	49	51	49	51	55	45	68	32	65	35	62	38	64	36	69	31	45	55	44	56	59	41	58	42		56.77	43.23
Espera total (s)							3.3		3.1		3.1		3.5		2.6		3		2.9		3.2		22.4		3.2		3		3.2		3.2			4.59	
Espera inicial (s)							3.3		3.1		3.1		3.5		2.6		3		2.9		3.2		22.4		3.2		3		3.2		3.2			4.59	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #3																																			
Consumo (%)	59	41		24.4	75.6		53	47	51	49	57	43	56	44	59	41	70	30	72	28	66	34	75	25	56	44	64	36	41	59	41	59		58.54	41.46
Espera total (s)							3		3		3		3		2.9		3.3		12.3		3		2.9		4.3		10.9		3.1		3.1			4.45	
Espera inicial (s)							3		3		3		3		2.9		3.3		12.3		3		2.9		4.3		10.9		3.1		3.1			4.45	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Totales																																			
P2P vs CDN	**P2P**	**CDN**																																	
Consumo (%)	57	43																																	
CDN1 vs CDN2	**CDN1**	**CDN2**																																	
Consumo (%)	27	73																																	
Calidad																																			
Espera total (s)	4.3																																		
Espera inicial (s)	4.3																																		
Cant. esperas	1																																		

Tabla 21: Ráfagas de cuatro peers. Resultados de las tres corridas.

11.3.3 Ráfagas de seis *peers*

Figura 79: Ráfagas de seis peers. Resultados de primer corrida.

Figura 80: Ráfagas de seis peers. Resultados de segunda corrida.

Figura 81: Ráfagas de seis peers. Resultados de tercer corrida.

	General			CDNs			Peer 1		Peer 2		Peer 3		Peer 4		Peer 5		Peer 6		Peer 7		Peer 8		Peer 9		Peer 10		Peer 11		Peer 12		Peer 13			Prom.	
	P2P	CDN		CDN 1	CDN 2		P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN	P2P	CDN			
Corrida #1																																			
Consumo (%)	43	57		13	45		52	48	52	48	55	45	55	45	56	44	56	44	59	41	52	48	64	36	0	100	7	93	35	65	14	86		42.85	57.15
Espera total (s)							3.4		3.2		3.2		3.5		3.4		3.2		3.2		3.6		3		3.4		3.2		3.1		3.1			3.27	
Espera inicial (s)							3.4		3.2		3.2		3.5		3.4		3.2		3.2		3.6		3		3.4		3.2		3.1		3.1			3.27	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #2																																			
Consumo (%)	55	45		23.9	76.1		61	39	60	40	60	40	71	29	59	41	62	38	68	32	31	69	45	55	53	47	43	57	49	51	46	54		54.46	45.54
Espera total (s)							3		2.9		3		5.3		3.2		3		5.3		3		3.1		3.3		3.4		3.2		3.3			3.46	
Espera inicial (s)							3		2.9		3		5.3		3.2		3		5.3		3		3.1		3.3		3.4		3.2		3.3			3.46	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Corrida #3																																			
Consumo (%)	60	40		27.5	72.5		59	41	60	40	59	41	64	36	61	39	60	40	67	33	79	21	84	16	50	50	49	51	52	48	39	61		60.23	39.77
Espera total (s)							3		3.2		3		3.3		3.2		3.2		3.2		3		3.1		3.2		3.2		3		3.2			3.14	
Espera inicial (s)							3		3.2		3		3.3		3.2		3.2		3.2		3		3.1		3.2		3.2		3		3.2			3.14	
Cant. esperas							1		1		1		1		1		1		1		1		1		1		1		1		1			1	
Totales																																			
P2P vs CDN	**P2P**	**CDN**																																	
Consumo (%)	53	47																																	
CDN1 vs CDN2	**CDN1**	**CDN2**																																	
Consumo (%)	21	65																																	
Calidad																																			
Espera total (s)	3.3																																		
Espera inicial (s)	3.3																																		
Cant. esperas	1																																		

Tabla 22: Ráfagas de seis peers. Resultados de las tres corridas.