_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Distribución de contenidos audiovisuales a través de Internet y su reproducción usando HTML 5

Autor: David Martínez Urrea

Tutor: Juan Carlos Guerri Cebollada

Cotutor: Francisco José Martínez Zaldívar

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2013-14

Valencia, 5 de septiembre de 2014

Resumen

Esta memoria se centra en la descripción y desarrollo del Trabajo Fin de Grado “Distribución de contenidos audiovisuales a través de Internet y su reproducción usando HTML 5”, que consiste en explicar y desarrollar un servidor web empleando APIs liberadas por Google para lograr establecer videollamadas entre dos clientes sin la necesidad de instalar nada, únicamente ejecutando JavaScript en el navegador. Para llegar a ello previamente se ha hecho un estudio del funcionamiento de WebRTC y algunos protocolos que intervienen en la comunicación. El servidor web instalado de Apache alberga diferentes archivos HTML5 con CSS para mejorar el aspecto visual, además de los archivos JavaScript que los propios HTML deben llamar para realizar las funciones de videollamada. Por último, se ha empleado un servidor de señalización cuya función es poner en contacto a dos clientes mediante tokens que se acoplan a una URL donde se establecerá la videollamada y se ha debido configurar correctamente el router para hacer accesible estos dos servidores a través del NAT.

Resum

Aquesta memòria es centra en la descripció i desenvolupament del Treball Fi de Grau "Distribució de continguts audiovisuals a través d'Internet i la seva reproducció usant HTML 5", que consisteix a explicar i desenvolupar un servidor web emprant APIs alliberades per Google per aconseguir establir videotrucades entre dos clients sense la necessitat d'instal·lar res, únicament executant JavaScript al buscador. Per arribar a això prèviament s'ha fet un estudi del funcionament WebRTC i alguns protocols que intervenen en la comunicació. El servidor web instal·lat d'Apache alberga diferents arxius HTML5 amb CSS per millorar l'aspecte visual, a més dels arxius JavaScript que els mateixos HTML han de cridar per a realitzar les funcions de videotrucada. Finalment, s'ha emprat un servidor de senyalització, la seva funció és posar en contacte a dos clients mitjançant tokens que s'acoplen a una URL on s'establirà la videotrucada i s'ha hagut de configurar correctament el router per fer accessible aquests dos servidors a través del NAT.

Abstract

This memory focuses on the description and development of the Final Project "Distribution of audiovisual content over the Internet and its playback using HTML 5", that is to explain and develop a web server using APIs released by Google to achieve establishing video calls between two clients without the need to install anything, just running JavaScript in the browser. To get there we have previously made a study of how WebRTC works and explain some protocols involved in communication. The Apache web server installed hosts some HTML5 with CSS files to improve the visual appearance, in addition to this the HTML files must call other

JavaScript files to make video call functions. Finally, we have employed a signaling server whose function is to connect two clients using tokens which are coupled to a URL where the video call is established and because it has properly configured the router to make these two servers accessible through the NAT.

Índice

Contenido

Capítulo 1. Introducción

Uno de los grandes retos actuales para las webs es poder establecer comunicaciones mediante voz y video: Real Time Communication. RTC debería ser algo tan natural como mandar emails, ya que estamos continuamente innovando y desarrollando nuevas formas de comunicación entre las personas como aplicaciones para móviles y tabletas. Web-RTC quiere llevar este concepto de comunicación instantánea a través de los navegadores, independientemente incluso del sistema operativo.

1.1 Historia y actualidad

Típicamente, las tecnologías para comunicarnos en tiempo real siempre han llegado primero a las empresas, ya que son las que más capital tienen, y mayor comunicación necesitan. Pero los problemas también surgen al intentar integrar algunas de estas tecnologías emergentes con las ya existentes, y en nuestro caso, que implica "tocar" código en la web, puede ser bastante tedioso.

Dando un repaso a los últimos años, y a las grandes marcas, Google incorporó videollamadas en 2008, lo cual resultó bastante popular, aunque también eran los años en los que Messenger estaba de moda y nos permitía el mismo servicio con mayores animaciones incluso. Más adelante en 2011, Google introduce Hangouts, que usa "Google Talk service", una extensión en Chrome de rápida instalación, y también compra GIPS, empresa dedicada a las RTC, con software valorado en 65 millones de € [1], que incluye códecs de video y audio además de técnicas de cancelación de ruido. Lo curioso de este caso, es que Google liberó la tecnología de GIPS y se juntó con el IETF y el W3C para conseguir consenso en cuanto a WebRTC. En Mayo de 2011, Ericsson programó la primera implementación, y grandes firmas como Microsoft y un gran desarrollador de Navegadores como es Mozilla (y ahora sistemas operativos para smartphones) se han sumado al desarrollo. [2]

Actualmente, WebRTC se implementa con estándares abiertos para la comunicación en tiempo real de video, audio y datos, además no necesita plugins ya que se integra en el navegador, lo cual es una gran ventaja ya que los plugins pueden tener problemas a la hora de descargarse, actualizarse, instalarse, también necesitan mantenimiento, puede que no se integren bien según la versión del navegador, etc. Y también es difícil persuadir a las personas para que quieran instalarlos por miedo a comprometer su seguridad.

En la figura siguiente podemos ver cómo han ido evolucionando las webs hasta el uso de WebRTC. La gran potencia de esta tecnología hace que desde cualquier dispositivo que tenga un navegador compatible podamos realizar una videollamada y ahorrar tiempo enviando unos cuantos emails, y yendo más allá, cuando se desarrollen algunos estándares se podría integrar con SIP, o incluso la red pública de telefonía (PSTN).

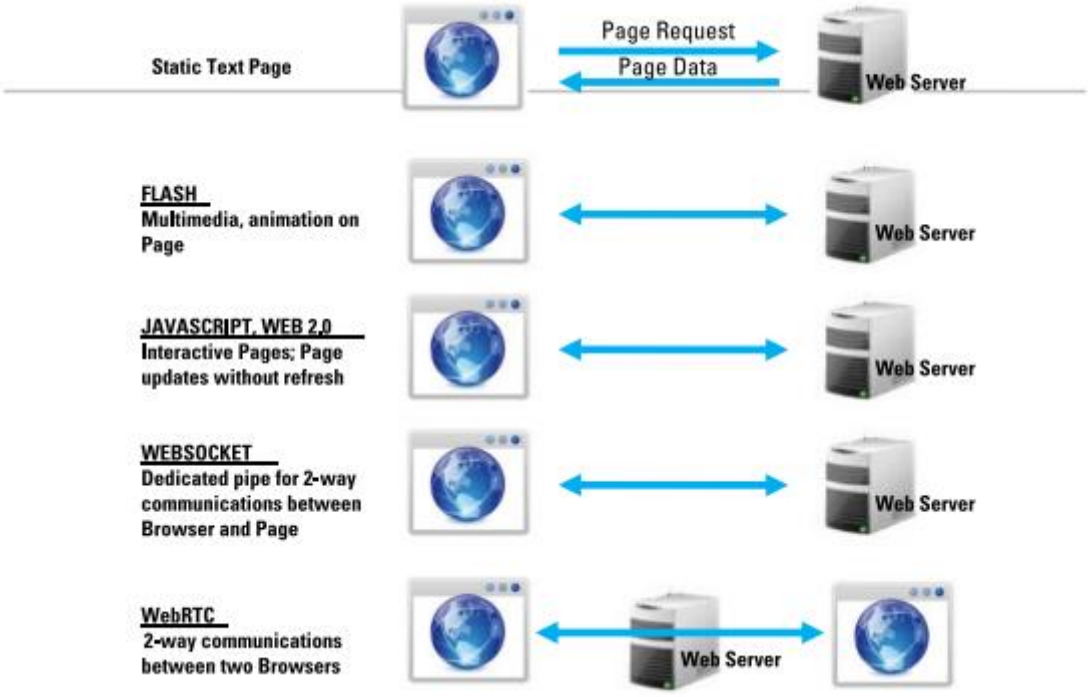


Figura 1: Evolución de la Web

Capítulo 2. Objetivos del TFG

El objetivo de este trabajo final de grado es comprender como funciona WebRTC, todos sus componentes y futuras aplicaciones, además de lograr una implementación práctica de esta tecnología en un servidor web que bien podría ser usado por una empresa, ya que nos ofrece una alternativa real a servicios como Hangouts o Skype con gran facilidad de integración en la web.

Para conseguir este objetivo se ha procedido primero documentándose sobre WebRTC, conociendo su funcionamiento teórico antes que nada. Después hemos probado alguna Demo disponible en Internet y capturado los paquetes para ver su funcionamiento.

Para darle una orientación profesional al trabajo final de grado, se ha montado un pequeño servidor web que también hace de señalizador, empleando HTML 5, CSS y JavaScript. Tras lograr el funcionamiento correcto de la aplicación, se ha procedido capturando los paquetes que intervienen en nuestro servidor, e incluso los logs de señalización para su explicación a lo largo de esta memoria.

Capítulo 3. Metodología de trabajo

3.1 Gestión del proyecto

El proyecto se ha desarrollado con la ayuda de bibliografía aportada por los tutores, dando éstos las referencias de las tareas que se querían desarrollar y trasladar a esta memoria. Con el montaje del servidor web en casa es de esperar que no pueda soportar gran cantidad de usuarios, no obstante, es un primer paso hacia lo que se podría llegar a conseguir si se trasladara a una empresa.

3.2 Distribución de Tareas

En este proyecto se han realizado cinco tareas principalmente: documentación, testeo de algunas aplicaciones en Internet, desarrollo del servidor, desarrollo del cliente y por último la fase de comprobación y testeo.

3.2.1 Documentación

En esta primera etapa se realizan las consultas oportunas en la bibliografía proporcionada, para entender el funcionamiento de WebRTC. También se aprende sobre HTML5 y CSS para poder desarrollar la parte Web. Se opta por usar un servidor Apache, Node.js para el uso algunas librerías y ejecución de JavaScript, y Sublime Text para hacer la programación más sencilla.

3.2.2 Testeo de aplicaciones

Se prueban 2 aplicaciones para investigar acerca del funcionamiento, la primera de ellas establece videoconferencias sencillas y es capaz de descubrir sesiones activas en la misma red donde nos encontramos: https://www.webrtc-experiment.com/video-conferencing/ [3]

La segunda de ellas es más cercana a lo que queremos hacer, establece videoconferencias con un identificador que se acopla a la URL: https://talky.io [4]

3.2.3 Desarrollo del cliente

En esta parte el objetivo era diseñar una web “bonita”, la página principal contiene un pequeño script que simula una sombra del cuadro principal. Luego disponemos de un botón de videollamada que nos redirige al servidor de señalización y por ultimo un formulario de contacto.

3.2.4 Desarrollo del servidor

En el servidor web Apache [5] tenemos los códigos de las páginas de navegación que el cliente demandara cuando entre. Luego además el servidor de señalización es un JavaScript que se deja encendido y para el cual nos hace falta la plataforma Node.js.

3.2.5 Comprobación y testeo

Se comprueba que efectivamente la web y la videollamada se pueden establecer tanto dentro de la red local como con una red ajena.

Capítulo 4. Desarrollo teórico del trabajo.

Web Real Time Communication es una API (Application Programming Interface) que en este caso va a utilizar JavaScript, lo cual evita dependencias con el sistema operativo. [6]

El funcionamiento básico seria:

1. El desarrollador web usa los códigos JavaScript (que son libres) y modifica su web para que por ejemplo aparezca un botón de comunicación.
2. El cliente, o usuario que está navegando por la web, tiene interés en contactar con un experto, con soporte técnico, o quiere compartir opiniones con otros usuarios conectados en ese momento, por lo que hace "click" en ese botón, imagen, o enlace.
3. En ese momento se pasa a la página que va a ejecutar todo el proceso de llamada y transmisión multimedia.
4. El receptor aceptará la llamada y empezará la transmisión de audio y video entre ambos navegadores (o más, si por ejemplo se trata de un foro de discusión).

4.1 Elementos que intervienen

4.1.1 APIs

Las hemos comentado antes, son las partes fundamentales que hacen que WebRTC funcione, cada una hace una cosa, pero son imprescindibles al menos en lo básico, en el futuro se pueden desarrollar muchas más.

4.1.2 MediaStream

MediaStream hace uso del método "getUserMedia()"[7], cuya función es obtener acceso al dispositivo local, que podría ser la webcam y el micrófono, y para que esto sea posible el usuario debe dar permiso. El mecanismo para obtener este permiso, su duración y cualquier otro detalle lo decide el navegador según lo especifiquen sus desarrolladores. El método devuelve "successCallback" si todo ha ido bien, o "errorCallback" si ha habido algún problema, éste último además devuelve el error PERMISSION_DENIED cuyo valor es 1.

La unidad funcional de la API es "MediaStreamTrack", que básicamente representa el un tipo de medio de un único dispositivo. Por ejemplo podría ser una pista de audio estéreo, o una pista de 6 canales de audio surround, pueden tener diferente número de canales, pero en cambio se trata como un único MediaStreamTrack, ya que aquí no se trata a nivel de canal. Al final según las especificaciones el medio se codificará y se transmitirá como un RTP y su tipo se definirá en el Payload Type. Aunque este objeto todavía no se usa como tal, ahora mismo están especificados el AudioMediaStreamTrack y el VideoMediaStreamTrack como subclases de MediaStreamTrack con su propio constructor.

El MediaStream finalmente es un conjunto de objetos MediaStreamTrack, cada uno debe llevar por tanto un ID para poder identificarlos, y poder hacer todas las operaciones de codificación, transmisión y decodificación por separado según el tipo de medio que estemos tratando. De la sincronización se puede encargar perfectamente RTP, que incluye el tiempo en que se generó la muestra y números de secuencia para que no haya ningún problema.

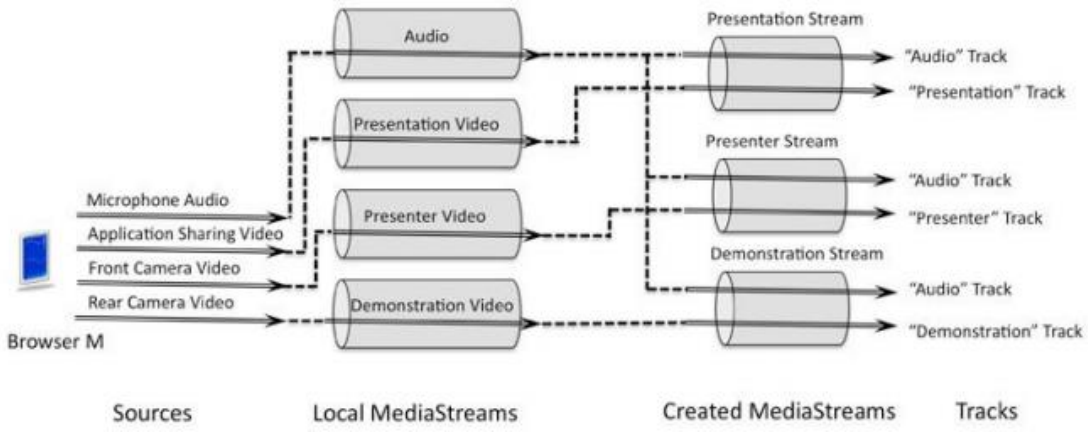


Figura 2: Obtención y transmisión de los diferentes Tracks en Streams

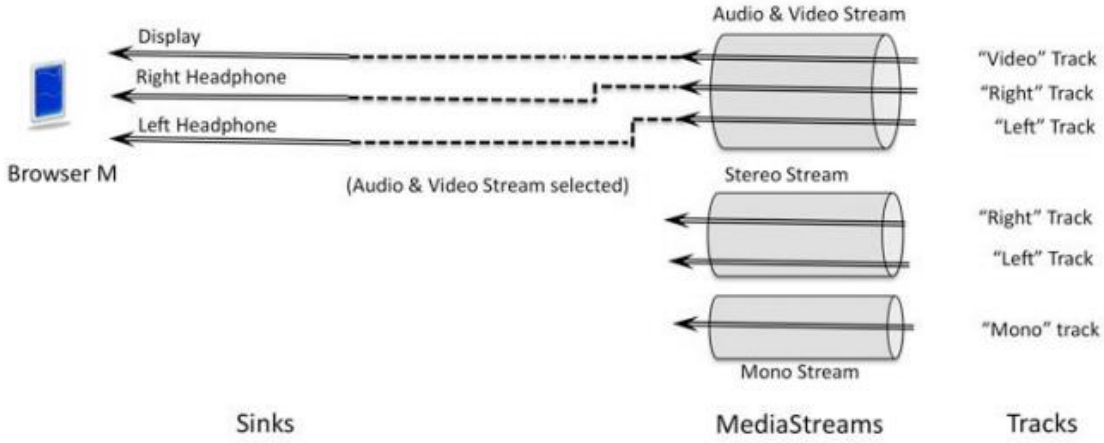


Figura 3: Recepción de un Media Stream con sus diferentes Tracks

4.1.3 RTCPeerConnection

Esta API es la principal para el correcto funcionamiento de WebRTC, ya que se encarga de establecer la conexión multimedia entre los navegadores. La API hace uso del JavaScript Session Establishment Protocol (JSEP) para la negociación, pero todo ello es manejado por el navegador, por lo que no nos concierne. [7]

El constructor del objeto debe contener las direcciones de los servidores que ayudan a establecer la sesión entre NATs y firewalls (STUN & TURN servers).

El establecimiento de la conexión se consigue tras una negociación SDP, que en esta API se define de la siguiente manera:

1. Se ejecuta el método "createOffer()" por el navegador, que genera un objeto RTCSessionDescription (al cual llamamos oferta).
2. Se llama al método "setLocalDescription()" con el objeto RTCSessionDescription para rellenar los datos.
3. Se envía el SDP generado para esta sesión al otro cliente por el canal de señalización.

Luego obviamente desde el otro extremo se necesita crear una respuesta, que se hará con el método "createAnswer()" del objeto RTCSessionDescription. Y cuando finaliza la negociación, la aplicación debe llamar al método "setRemoteDescription()" para que se aplique la configuración en el navegador y así poder finalmente empezar a transmitir y recibir.

Con toda negociación se establecen aspectos importantes como los códecs de video y audio, tipos de cifrado y compresión o ancho de banda, de tal forma que se puedan entender los videos y audios desde ambas partes, y según las capacidades de cada máquina, ya que un smartphone difícilmente puede llegar a las capacidades de un Ultrabook (como ejemplo).

4.1.4 DataChannel

Otra API fundamental, la cual es capaz de abrir un canal bidireccional para su uso sobre Peer Connection. Al ser un canal exclusivo de datos, podemos mandar cualquier cosa y puede aprovechar el hecho de que Peer Connection nos provea de una baja latencia para trabajar con un alto caudal de información, pudiéndose utilizar en aplicaciones de escritorio remoto, chats de texto, transferencia de archivos directa o el caso con mayor visión quizás, los videojuegos.

Los videojuegos online generan gran cantidad de ingresos cada año, y el hecho de que los jugadores esperen poco tiempo entre partidas es algo muy valorable. Por norma general, los usuarios se conectan a un servidor que les hace de nexo entre ellos, poniéndonos en un caso peor, una partida de 32 jugadores, necesita 32 jugadores enviando su información al servidor, y éste la de todos los demás jugadores a cada uno, por lo que está siendo sobrecargado. Con DataChannel, podríamos unir a los jugadores entre ellos, y seguramente conseguiríamos bajar latencias y tener una mayor estabilidad en el juego.

Dejando aparte este tema, el DataChannel se crea con el método RTCPeerConnection.createDataChannel(), que también incorpora un objeto llamado "dataChannelDict" a modo de configuración. ¿Y qué podemos configurar? La entrega fiable o no de los datos, en un entorno como por ejemplo los videojuegos, tener alguna pérdida no influye, pero transmitiendo un archivo, deben llegar el 100% de los paquetes. Se pueden configurar los parámetros "maxRetransmitTime y "maxRetransmits" para que aun poniendo entregas fiables, no perjudiquemos en caso de tener altas pérdidas aunque sea por un pequeño contratiempo. [7]

Otras funcionalidades que puede aportar son la priorización si se usa más de 1 canal, seguridad mediante DTLS (Datagram Transport Layer Security), control de congestión (controlando las pérdidas y ajustando los parámetros antes mencionados), y además se puede utilizar con audio y video o sin ellos.

Usar RTCDataChannels es mucho más rápido que WebSockets incluso si necesitamos un servidor TURN para atravesar NATs y firewalls, por lo que se puede convertir en una herramienta muy potente cuando cualquier milisegundo en una transmisión es importante.

4.1.5 Web Servers

Aunque sea una parte obvia no hay que descuidarla, el servidor web contendrá las WebRTC APIs, puede ser el mismo que contenga la web ya existente de la empresa o proveedor de servicios o estar aparte. Lo importante es que cuando el usuario haga click en una URL para realizar la videollamada u otra tarea de WebRTC, el servidor este bien configurado para responder. [6]

4.1.6 Navegadores

Hasta la fecha los principales navegadores tanto en sus versiones para PC como para terminales móviles lo soportan.

- Google Chrome
- Mozilla Firefox
- Opera

También estamos a la espera de que el gigante Microsoft lo incorpore a Internet Explorer, quizá en alguna nueva versión.

4.1.7 Códecs

Los códecs como bien sabemos son especificaciones para transformar flujos de datos y transmitirlos en forma de señal, de tal forma que al recuperar la señal y aplicar la decodificación volvamos a tener los datos originales. Esto no es del todo cierto, nunca se llega a obtener una señal al 100% original, pero hacen que las pérdidas sean imperceptibles (por ejemplo en el caso del mp3). Con los códecs podemos comprimir la información y transmitirla ahorrando ancho de banda, que en el caso de una videoconferencia es muy importante.

En WebRTC, los códecs involucrados de forma oficial son [6]:

- Audio: Opus, códec dinámico, liberado por Google, con 1 o 2 canales, 48.000 Hz de frecuencia de muestreo, tamaño de muestra de 2,5 a 60 ms que por defecto se deja a 20ms.
- Audio: G.711, con 1 canal, 8000 Hz de frecuencia de muestreo, usado en telefonía, con un tamaño de muestra de 20 ms, puede usarse PCM (pulse code modulation) con ley A, o ley μ.

- Video: VP8, códec dinámico, propiedad de Google, aunque liberado, con frecuencia de muestreo de 90.000 Hz.

4.1.8 Gateways

Muchas empresas usan sistemas de comunicación propios como una pequeña central telefónica o servicios más avanzados como SIP. Si estas empresas quieren integrar WebRTC con sus sistemas actuales son necesarios los gateways. [6]

Los gateways son de cierta manera "puentes" que unen varias tecnologías, y que pueden ser instalados en la misma organización que emplea SIP o PBX, o en el proveedor de servicios si es externo. En cualquier caso, este gateway proporciona diversas funciones:

- Translación entre SIP y WebRTC
- Transcodificación y ajuste del bitrate del tráfico.
- Interconectividad.

4.1.9 Session Border Controller

El Session Border Controller es un elemento corporativo empleado en el extremo donde se conecta la empresa con el proveedor de redes VoIP, desde donde se sale al resto de redes. Tiene diversas funciones, algunas como el propio gateway, lo cual no implica que lo sustituya. Las explico a continuación [6]:

- Seguridad en la red: este elemento está diseñado para proteger de una variedad de ataques como DoS (Denial of Service), spoofing (suplantación o robo de identidades), y cualquier ataque basado en la web que alberga WebRTC, de tal forma que la red interna de VoIP esté completamente segura.
- Proporciona interconectividad: tal como el gateway, es capaz de ayudar en el proceso de conversión del tráfico VoIP de WebRTC a SIP o a otro sistema VoIP, además de arreglar las pequeñas diferencias de protocolos cuando existen algunas incompatibilidades para que las comunicaciones funcionen.
- Control de la admisión de llamadas a la red: su propio nombre lo indica, podríamos decir que actúa como un firewall, pero en este caso determina que llamadas o sesiones puede entrar a nuestra red. Se podrían incluso crear "black lists" y "white lists" para facilitar este proceso.
- Transcodificación y "transrating": igual que el gateway, es capaz de convertir entre diferentes códecs para que la comunicación sea entendible extremo a extremo, y ajustar el bitrate requerido por la red.
- Supera la topología de la red: si usáramos WebRTC únicamente a nivel interno, alcanzaríamos a todos los usuarios, pero como bien sabemos, queremos interconectar usuarios de todo el mundo. Las empresas usan IPs privadas y mediante un NAT con una

o varias IPs públicas salen al resto del mundo, por lo que el SBC es capaz de atravesar los NATs y alcanzar cualquier parte del mundo. Otra de sus ventajas es que se oculta la topología de la red local al resto del mundo, hoy en día muy importante para la seguridad.

4.2 Escenarios posibles

4.2.1 WebRTC con SIP

Hemos hablado de los elementos necesarios para integrar WebRTC con SIP, y como actuarían, sobre todo con la transcodificación y "transrating". Ahora bien, ¿Cómo es el proceso que siguen para establecer la conexión? [7]

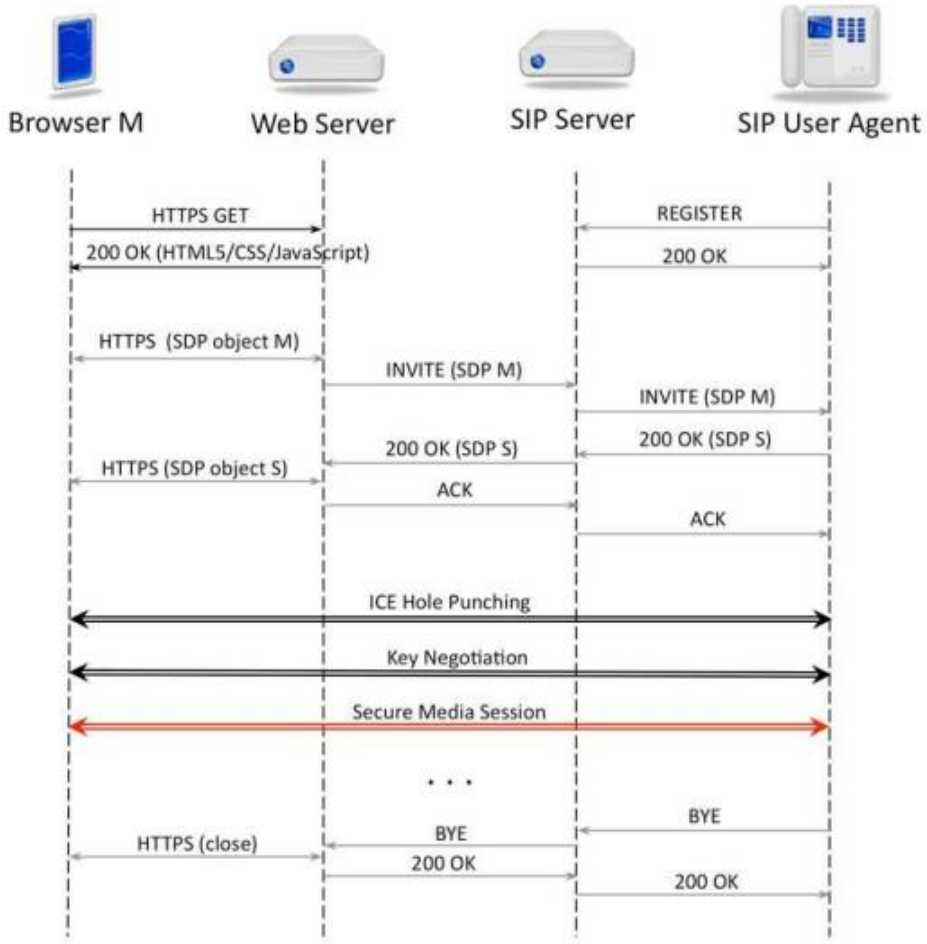


Figura 4: WebRTC con SIP

Como podemos ver es bastante sencillo, el cliente se conecta al servidor web como haría normalmente, intentaría llamar a un cliente SIP, para lo cual el servidor web se comunicara con el servidor SIP donde este registrado el cliente y se empezará una negociación con SDP (Session Description Protocol) para establecer los parámetros necesarios para entenderse.

Esta podría ser la topología empleada, donde vemos el Media Gateway (SBC) que hace posible el entendimiento entre ambos extremos.

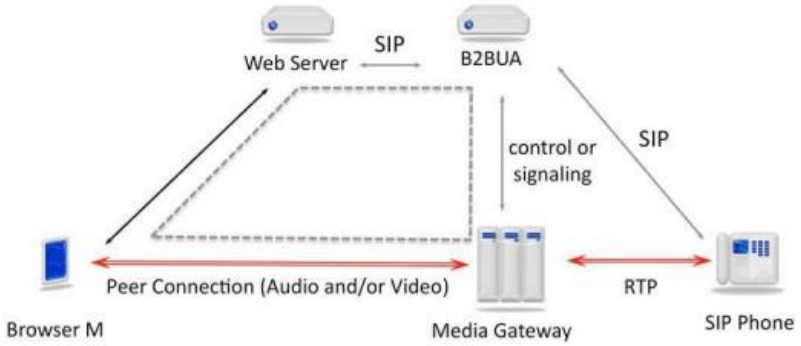


Figura 5: Topología empleada en una llamada WebRTC-SIP

4.2.2 WebRTC con la red pública de telefonía

Se podría decir que es un escenario parecido a SIP, donde solo sería posible establecer llamadas con audio. El servidor web y el PSTN Gateway deberán utilizar algún protocolo de señalización y control para mantener la llamada y además emplear el códec G.711 (PCM) para evitar cualquier transcodificación de la señal de audio (que provocaría más retardos). [7]

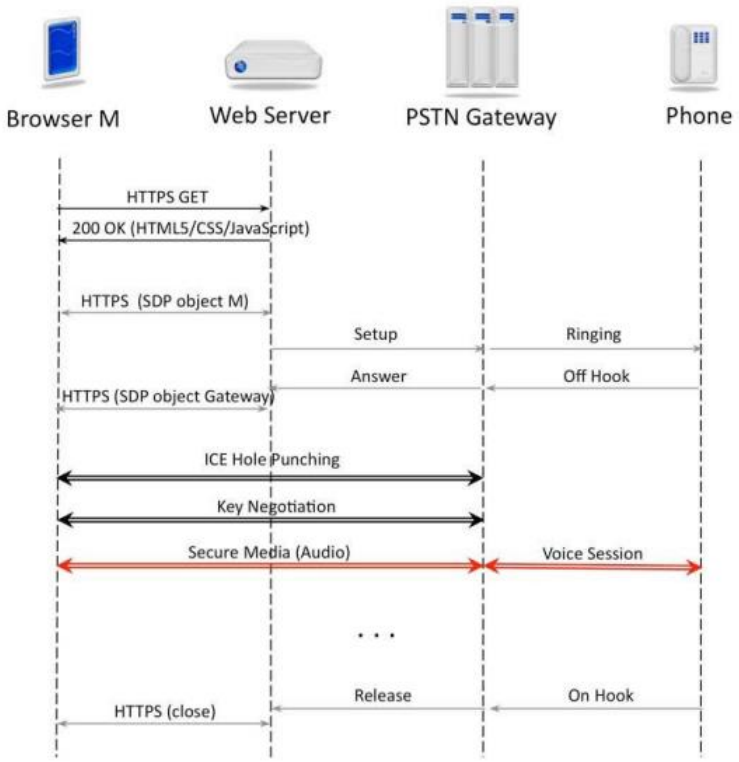


Figura 6: WebRTC con PSTN

4.3 Firewall

Para que las comunicaciones sean posibles es necesario en se habilite el paso de información de señalización a través del firewall, para lo cual tenemos diversas opciones.

Interactive connectivity establishment (ICE) es un estándar con un procedimiento seguro para determinar la IP pública y e información de los puertos a usar para los usuarios tanto externos como internos, para ello se usan dos protocolos [6]:

- Session Transversal utilities for NAT (STUN) determina aquellas restricciones que pueden impedir la conexión entre dos clientes mediante uno o varios routers. El cliente envía una petición a un servidor STUN de internet y éste le responde con su IP pública y si el otro cliente es accesible o no a través de su NAT.
- Traversal Using Relay around NAT (TURN) se usa cuando tenemos restricciones en el acceso mediante el router, consiste en que toda la información pasa a través de él, encargándose de redireccionar los paquetes, en cierto modo es parecido a un Proxy, y obviamente sobrecargamos con un poco de overhead y provocamos retardos, por lo que se recomienda que solo se use como última alternativa. Aquí debajo podemos ver un pequeño grafico de cómo se establecería la conexión usando STUN y TURN.

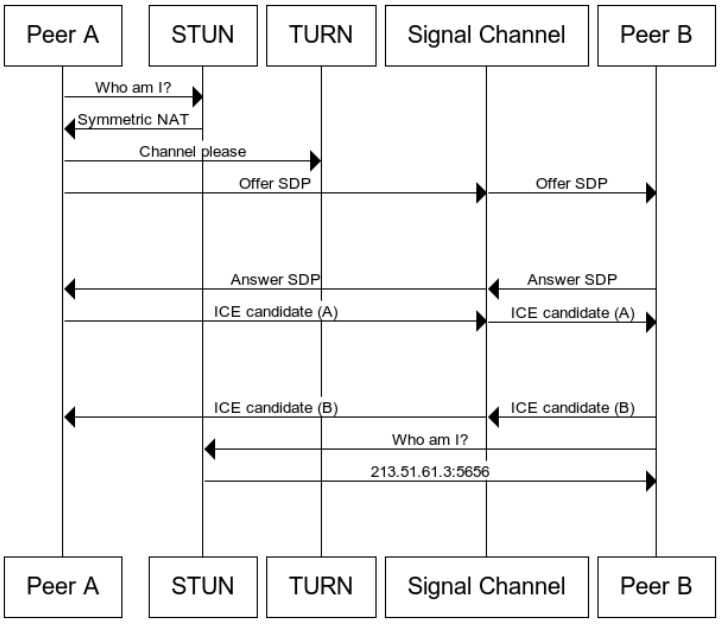


Figura 7: Grafico establecimiento de conexión con STUN y TURN

Si ninguno de los extremos está detrás de un firewall no se debería requerir ICE, pero en general estamos detrás de NATs, por lo que el uso de STUN es casi obligatorio.

4.4 Política de accesos

Como en todas las redes, los recursos no son limitados, sobre todo las conexiones con nuestro proveedor de servicios, por lo que es importante definir políticas para determinar quién puede usar qué recursos y cuándo [6].

Antes hablé del Session Border Controller, y sus funciones entre las cuales podía restringir el acceso a las llamadas que quisiéramos, no obstante, en este caso la política a definir tiene que ver con otros aspectos relevantes como:

- Ancho de banda utilizado y su limitación.
- Menor coste de enrutamiento para llamadas de larga distancia.
- Disponibilidad de la aplicación (de 9.00 a 17.00 por ejemplo).
- Caminos y enrutamiento de flujos multimedia.

WebRTC, por su naturaleza peer-to-peer puede ocupar mucho ancho de banda en las videollamadas por ejemplo, mediante políticas podemos manejar esto para no ver afectados otros servicios, sobre todo cuando integramos con redes SIP. Además también podemos requerir SLAs (Service Level Agreements) para mantener un mínimo de calidad en las comunicaciones.

Básicamente, las políticas se definen para restringir el tráfico y prevenir una sobrecarga de la red, o si lo preferimos, podemos dar libertad al uso de aplicaciones y solo restringir en casos de sobrecarga.

4.5 Seguridad

Hoy por hoy, lo que más preocupa al usuario habitual es la seguridad de sus datos, pero también toda la información que se transmite en tiempo real, todos deseamos que nuestras contraseñas viajen de forma segura por la red cuando por ejemplo accedemos a nuestro banco online, ya que actualmente es un claro ejemplo de donde más atacan los ciberdelincuentes.

No obstante, también deseamos que nadie pueda escuchar una conversación por Skype o por WebEx, ya sea una conversación personal o empresarial, que podría llevar pérdidas económicas por intercepción de información privilegiada.

Además, se ha visto que algunas aplicaciones (como hasta hace un tiempo WhatsApp) o plugins que se instalan en nuestro navegador comprometen continuamente nuestra información o directamente la roban sin que el usuario se dé cuenta.

Por lo tanto, WebRTC, que se integra en el navegador, proporciona algunos mecanismos para evitar cualquier intento de descifrar la información [6].

- Implementa protocolos seguros como DTLS (Datagram Transport Layer Security) y SRTP (Secure Real-Time Transport Protocol).

- Con los protocolos seguros se proporciona encriptación para todos los componentes WebRTC incluyendo los mecanismos de señalización.
- Como WebRTC va integrado en el navegador y no es un plugin, no es un proceso separado y por lo tanto es más difícil de penetrar, además de actualizarse cada vez que el navegador se actualiza si así lo deciden los desarrolladores, proceso que en Chrome se hace automáticamente.
- Por último, el acceso a la cámara y el micrófono lo da el usuario explícitamente para evitar capturas no deseadas, que además se muestran en la interfaz para ver en todo momento como se nos ve.

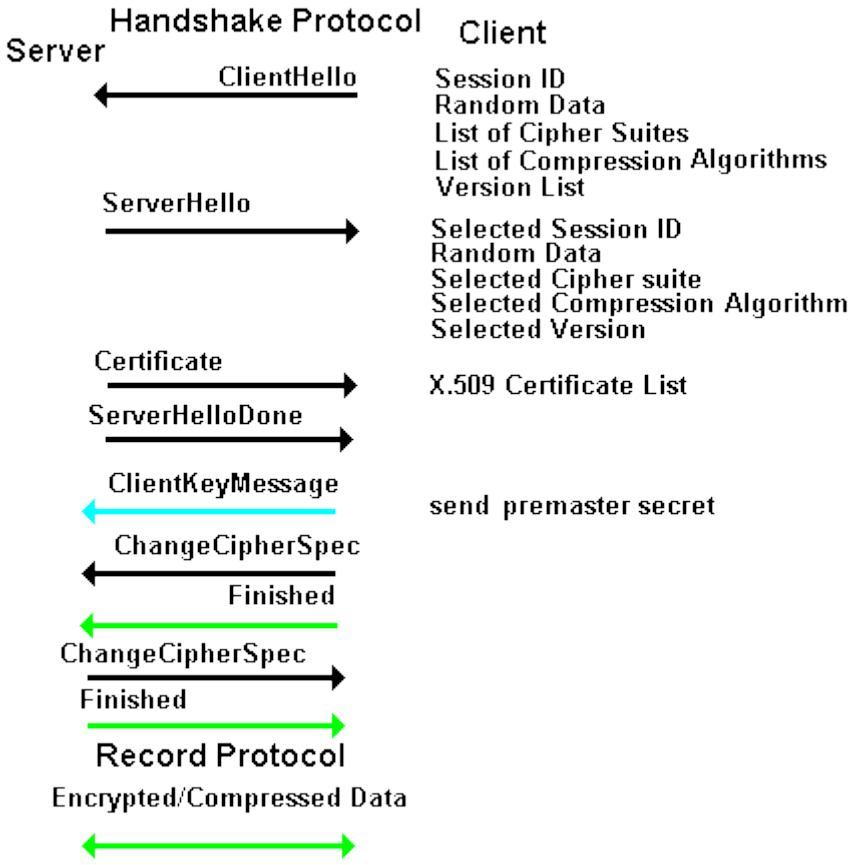


Figura 8: Establecimiento de una sesión segura mediante Handshake Protocol

Capítulo 5. Desarrollo práctico del trabajo

5.1 Servidor

Uno de los primeros pasos es descargar Node.js [8], situarnos en el directorio donde tendremos el servidor y ejecutar el comando:"`npm install websocket`", ya que en esta implementación no vamos a usar DataChannels, sino WebSockets. [9]

La primera parte del servidor son diversas variables, entre las que se encuentra el puerto local a emplear, en este caso el "1234" (obligado abrirlo y redirigirlo en el router). Luego la variable http_server se encarga de crear el servidor que nos va a cargar la página de la videollamada.

```
var http = require("http");
var fs = require("fs");
var websocket = require("websocket").server;
var port = 1234;
var webrtc_clients = [];
var webrtc_discussions = {};
var http_server = http.createServer(function(request, response) {
  var matches = undefined;
  if (matches = request.url.match("^/images/(.*)")) {
    var path = process.cwd()+"/images/"+matches[1];
    fs.readFile(path, function(error, data) {
      if (error) {
        log_error(error);
      } else {
        response.end(data);
      }
    });
  } else {
    response.end(page);
  }
});
```

A partir de aquí, el servidor se pone a la escucha en el puerto local definido para señalización, y se le pasa la página "videocall.html" que contiene toda la parte que se ejecuta en el navegador del cliente.

```
http_server.listen(port, function() {
  log_comment("server listening (port "+port+")");
});
var page = undefined;
fs.readFile("videocall.html", function(error, data) {
  if (error) {
    log_error(error);
  } else {
    page = data;
  }
});
```

Esta parte del servidor se dedica a procesar las peticiones, básicamente, cuando entra una petición al cliente se le devuelve el "videocall.html", que más adelante explicaré, y en cuyo código se define un "token" que le es asignado cuando entra (depende de la hora, por lo que va cambiando continuamente). Se supone que el primer cliente, el llamante, debe enviar un enlace al receptor de la llamada, y éste entrar a través de él. Entonces, lo que está programado aquí en cierto modo es la validación del enlace (que contiene el token), de tal forma que el token una vez se cierra la conexión se borra y pasa a no ser válido.

```
var websocket_server = new websocket({httpServer: http_server});
websocket_server.on("request", function(request) {
  log_comment("new request ("+request.origin+")");

  var connection = request.accept(null, request.origin);
  log_comment("new connection ("+connection.remoteAddress+")");

  webrtc_clients.push(connection);
  connection.id = webrtc_clients.length-1;

  connection.on("message", function(message) {
    if (message.type === "utf8") {
      log_comment("got message "+message.utf8Data);

      var signal = undefined;
      try { signal = JSON.parse(message.utf8Data); } catch(e){ };
      if (signal) {
        if (signal.type === "join" && signal.token !== undefined){
          try {
            if (webrtc_discussions[signal.token] === undefined) {
              webrtc_discussions[signal.token] = {};
            }
          } catch(e) { };
          try {
            webrtc_discussions[signal.token][connection.id]     =
true;
          } catch(e) { };
        } else if (signal.token !== undefined) {
          try {

Object.keys(webrtc_discussions[signal.token]).forEach(function
(id) {
              if (id != connection.id) {
                webrtc_clients[id].send(message.utf8Data,
log_error);
              }
            });
          } catch(e) { };
        } else {
          log_comment("invalid signal: "+message.utf8Data);
        }
      } else {
        log_comment("invalid signal: "+message.utf8Data);
      }
```

```
    }
  });
```

Aunque leyendo el código se puede entender, esta parte se encarga de finalmente cerrar la conexión y eliminar el token cuando los clientes se desconectan, de tal forma que es imposible acceder al mismo token (si por ejemplo un cliente pierde la conexión durante un momento, tienen que volver a establecer la videollamada porque el token se ha borrado)

```
  connection.on("close", function(reason_code, description) {
    log_comment("connection  closed  ("+this.remoteAddress+"  -
"+reason_code+" - "+description+")");
    Object.keys(webrtc_discussions).forEach(function(token) {

Object.keys(webrtc_discussions[token]).forEach(function(id) {
        if (id === connection.id) {
          delete webrtc_discussions[token][id];
        }
      });
    });
  });
});
```

Y aquí dos pequeñas funciones para tener logs de los errores que se producen y guardar los comentarios que se envían.

```
function log_error(error) {
  if (error !== "Connection closed" && error !== undefined) {
    log_comment("ERROR: "+error);
  }
}
function log_comment(comment) {
  console.log((new Date())+" "+comment);
}
```

5.2 Cliente

El grueso del código está en el cliente, por lo que únicamente voy a explicar las partes más importantes, ya que son más de 600 líneas de código. [9]

Es un documento HTML5 con CSS que incluye el JavaScript necesario para realizar la videollamada, con apoyo del servidor de señalización. [10] [11] [12]

El primer script inicia variables de WebRTC, nos proporciona una IP de un servidor STUN, y se encarga de la función "getUserMedia" que comenté en la parte teórica. Como podemos ver tiene 3 implementaciones según el navegador que usemos, la primera es para Chrome, la segunda para Mozilla Firefox y la última que emplea webkit se usa en Opera.

```
   var webrtc_capable = true;
   var rtc_peer_connection = null;
   var rtc_session_description = null;
   var rtc_ice_candidate = null
   var get_user_media = null;
   var connect_stream_to_src = null;
   var stun_server = "stun.l.google.com:19302";
if (navigator.getUserMedia) {
      rtc_peer_connection = RTCPeerConnection;
      rtc_session_description = RTCSessionDescription;
      rtc_ice_candidate = RTCIceCandidate;
      get_user_media = navigator.getUserMedia.bind(navigator);
      connect_stream_to_src     =     function     (media_stream,
media_element) {
            media_element.srcObject = media_stream;
            media_element.play();
      };
} else if (navigator.mozGetUserMedia) {
      rtc_peer_connection = mozRTCPeerConnection;
      rtc_session_description = mozRTCSessionDescription;
      rtc_ice_candidate = mozRTCIceCandidate;
      get_user_media = navigator.mozGetUserMedia.bind(navigator);
      connect_stream_to_src     =     function     (media_stream,
media_element) {
            media_element.mozSrcObject = media_stream;
            media_element.play();
      };
      stun_server = "23.21.150.121"; // Mozilla STUN server
} else if (navigator.webkitGetUserMedia) {
      rtc_peer_connection = webkitRTCPeerConnection;
      rtc_session_description = RTCSessionDescription;
      rtc_ice_candidate = RTCIceCandidate;
      get_user_media                                          =
navigator.webkitGetUserMedia.bind(navigator);
      connect_stream_to_src     =     function     (media_stream,
media_element) {
            media_element.src                                 =
webkitURL.createObjectURL(media_stream);
            media_element.play();
```

```
      };
} else {
      alert("This browser does not support WebRTC / Este navegador
no soporta WebRTC");
      webrtc_capable = false;
}
```

El siguiente script es mucho más largo, por lo que intentare explicarlo en pequeñas dosis. Lo primero que tenemos son diversas variables para la conexión, incluido el servidor de señalización y el token que se usa en la llamada. Luego tenemos la función start(), que va a manejar toda la creación de la videollamada y la unión del cliente que la recibe.

```
   var call_token;
   var signaling_server;
   var peer_connection;
   var file_store = [];
   var local_stream_added = false;

   function start() {
      peer_connection = new rtc_peer_connection({
      "iceServers": [
                { "url": "stun:"+stun_server }, ]
      });
```

Esta función se encarga e interconectar los diferentes peers gracias al servidor de señalización.

```
      peer_connection.onicecandidate = function (ice_event) {

            console.log("new ice candidate");
            if (ice_event.candidate) {
                  signaling_server.send(
                        JSON.stringify({
                              token:call_token,
                              type: "new_ice_candidate",
                              candidate: ice_event.candidate ,
                        })
                  );
            }
      };
```

Esta función se encarga de mostrar el video cuando nos llega el elemento “remote_video” con la etiqueta <video> de HTML5.

```
      peer_connection.onaddstream = function (event) {
            console.log("new remote stream added");
            connect_stream_to_src(event.stream,
document.getElementById("remote_video"));

      document.getElementById("loading_state").style.display    =
"none";
            console.log("updating UI to open_call_state");
```

```
      document.getElementById("open_call_state").style.display  =
"block";
      };
```

Esta función está definida más adelante, obtiene el video y audio local.

```
      setup_video();
```

Establece la conexión con el servidor de señalización mediante WebSocket, en este caso he dejado puesto “localhost”, pero en la implementación práctica debemos poner la dirección IP pública donde este alojado este servidor.

```
      console.log("setting up connection to signaling server");
      signaling_server = new WebSocket("ws://localhost:1234");
```

Esta parte del código intenta localizar nuestro token, lo normal es que si hemos creado nosotros la videollamada no tengamos token, por lo que en este caso generaríamos uno, haciendo uso de la fecha actual para que difícilmente se creen 2 tokens a la vez con el mismo contenido.

```
if (document.location.hash === "" || document.location.hash ===
undefined) {
           console.log("you are the Caller");

           var token = Date.now();
           call_token = "#"+token;

           document.location.hash = token;
```

Y aquí le comunicamos al servidor que hemos creado una videollamada y le decimos cual es el token que se nos ha asignado.

```
           signaling_server.onopen = function() {

                signaling_server.onmessage                    =
caller_signal_handler;

                console.log("sending  'join'  signal  for  call
token:"+call_token);
                signaling_server.send(
                     JSON.stringify({
                          token:call_token,
                          type:"join",
                     })
                );
           }
```

Además generamos una pequeña página HTML donde le facilitamos al cliente que llama el enlace que debe pasar al otro peer para conectarse.

```
           document.title = "WebRTC - Llamada enviada";
           console.log("updating UI to loading_state");
           document.getElementById("loading_state").innerHTML   =
"Videollamada     preparada...dale     este     link     a     tu
amig@:<br/><br/>"+document.location;
```

Ahora nos ponemos en el caso contrario, somos el peer que tiene el enlace y nuestro amigo nos está esperando. Accedemos al enlace y esta parte del código lo que hace es enviar el token (que forma parte del enlace) al servidor de señalización, para que éste nos ponga en contacto directo y podamos establecer la videollamada.

```
	} else {
		console.log("you are the Callee");

		call_token = document.location.hash;

		signaling_server.onopen = function() {

		signaling_server.onmessage = callee_signal_handler;

		console.log("sending 'join' signal for call
token:"+call_token);
			signaling_server.send(
				JSON.stringify({
					token:call_token,
					type:"join",
				})
			);

		console.log("sending 'callee_arrived' signal for call
token:"+call_token);
			signaling_server.send(
				JSON.stringify({
					token:call_token,
					type:"callee_arrived",
				})
			);
		}

		document.title = "WebRTC - Llamada recibida";
		console.log("updating UI to loading_state");
		document.getElementById("loading_state").innerHTML =
"Conectando la videollamada...";
	}
```

El resto de elementos de esta función están destinados al chat de mensajes y a compartir imagines, por lo que no los voy a explicar.

La función que viene a continuación se encarga de procesar los SDP que se usan para establecer los parámetros básicos para la comunicación.

```
function new_description_created(description) {
	peer_connection.setLocalDescription(
		description,
		function () {
			signaling_server.send(
				JSON.stringify({
					token:call_token,
					type:"new_description",
					sdp:description
				})
			);
```

```
            },
            log_error
      );
}
```

Esta función se encarga de manejar los diferentes eventos cuando nosotros somos la persona llamante, básicamente, si llega el receptor de la llamada, le debemos crear una oferta SDP, si llega un nuevo "ice_candidate", lo añadiremos, y por ultimo si llega un SDP y lo aceptamos debemos establecerlo y responder que lo hemos aceptado, en caso contrario seguiríamos negociando.

```
function caller_signal_handler(event) {
      var signal = JSON.parse(event.data);
      console.log(signal.type);
      if (signal.type === "callee_arrived") {
            create_offer();
      } else if (signal.type === "new_ice_candidate") {
            peer_connection.addIceCandidate(
                  new rtc_ice_candidate(signal.candidate)
            );
      } else if (signal.type === "new_description") {
            peer_connection.setRemoteDescription(
                  new rtc_session_description(signal.sdp),
                  function () {
                        if  (peer_connection.remoteDescription.type
== "answer") {                    }
                  },
                  log_error
            );
      }
}
```

La función que viene a continuación es bastante parecida a la anterior, en esta se tratan los eventos cuando nosotros hemos sido los receptores, que básicamente manejamos cuando llega un nuevo "ice_candidate" y cuando nos llega un SDP para responder.

```
function callee_signal_handler(event) {
      var signal = JSON.parse(event.data);
      console.log(signal.type);
      if (signal.type === "new_ice_candidate") {
            peer_connection.addIceCandidate(
                  new rtc_ice_candidate(signal.candidate)
            );
      } else if (signal.type === "new_description") {
            peer_connection.setRemoteDescription(
                  new rtc_session_description(signal.sdp),
                  function () {
                        if  (peer_connection.remoteDescription.type
== "offer") {
                              create_answer();
                        }
                  },
                  log_error
            );

      }
```

```
}
```

La siguiente función crea las ofertas SDP, en caso de no haber todavía flujo local del cual crear un SDP, se establece un timeout y se intenta de nuevo.

```
function create_offer() {
      if (local_stream_added) {
            console.log("creating offer");
            peer_connection.createOffer(
                  new_description_created,
                  log_error
            );
      } else {
            console.log("local stream has not been added yet -
delaying creating offer");
            setTimeout(function() {
                  create_offer();
            }, 1000);
      }
}
```

Esta es igual que la anterior, pero se usa en las respuestas a las ofertas SDP, tienen diferente nombre porque se usan en casos distintos, pero hacen exactamente lo mismo.

```
function create_answer() {
      if (local_stream_added) {
            console.log("creating answer");
            peer_connection.createAnswer(new_description_created,
log_error);
      } else {
            console.log("local stream has not been added yet -
delaying creating answer");
            setTimeout(function() {
                  create_answer();
            }, 1000);
      }
}
```

Esta función se encarga de obtener el acceso local al video y audio, lo muestra en la pantalla local gracias al elemento “local_video” que se mete en una etiqueta <video> de HTML5 y por ultimo lo conecta a local_stream para luego enviarlo mediante el método addStream de peerConnection.

```
function setup_video() {
      console.log("setting up local video stream");
      get_user_media(
            {
                  "audio": true,
                  "video": true        },
            function (local_stream) {
                  console.log("new local stream added");
                  connect_stream_to_src(local_stream,
document.getElementById("local_video"));
                  document.getElementById("local_video").muted   =
true;
```

```
                console.log("local      stream      added      to
peer_connection to send to remote peer");
                peer_connection.addStream(local_stream);
                local_stream_added = true;
            },
            log_error
      );
}
```

Y esto es todo, el resto de código empleado se usa para mensajes de chat y compartir imágenes, pero la parte básica para realizar una videollamada esta explicada. Luego ya la parte visual se puede modificar al gusto de cada uno mediante CSS.

La figura representa la captura de pantalla de cómo se ven las videollamadas, a la izquierda arriba tenemos el local_video, y en grande a nuestro amigo cuyo elemento es el remote_video. Si nos fijamos bien en la URL aparece la IP del servidor, el puerto 1234 y después una / y un # seguidos de un gran número que representa el token empleado.

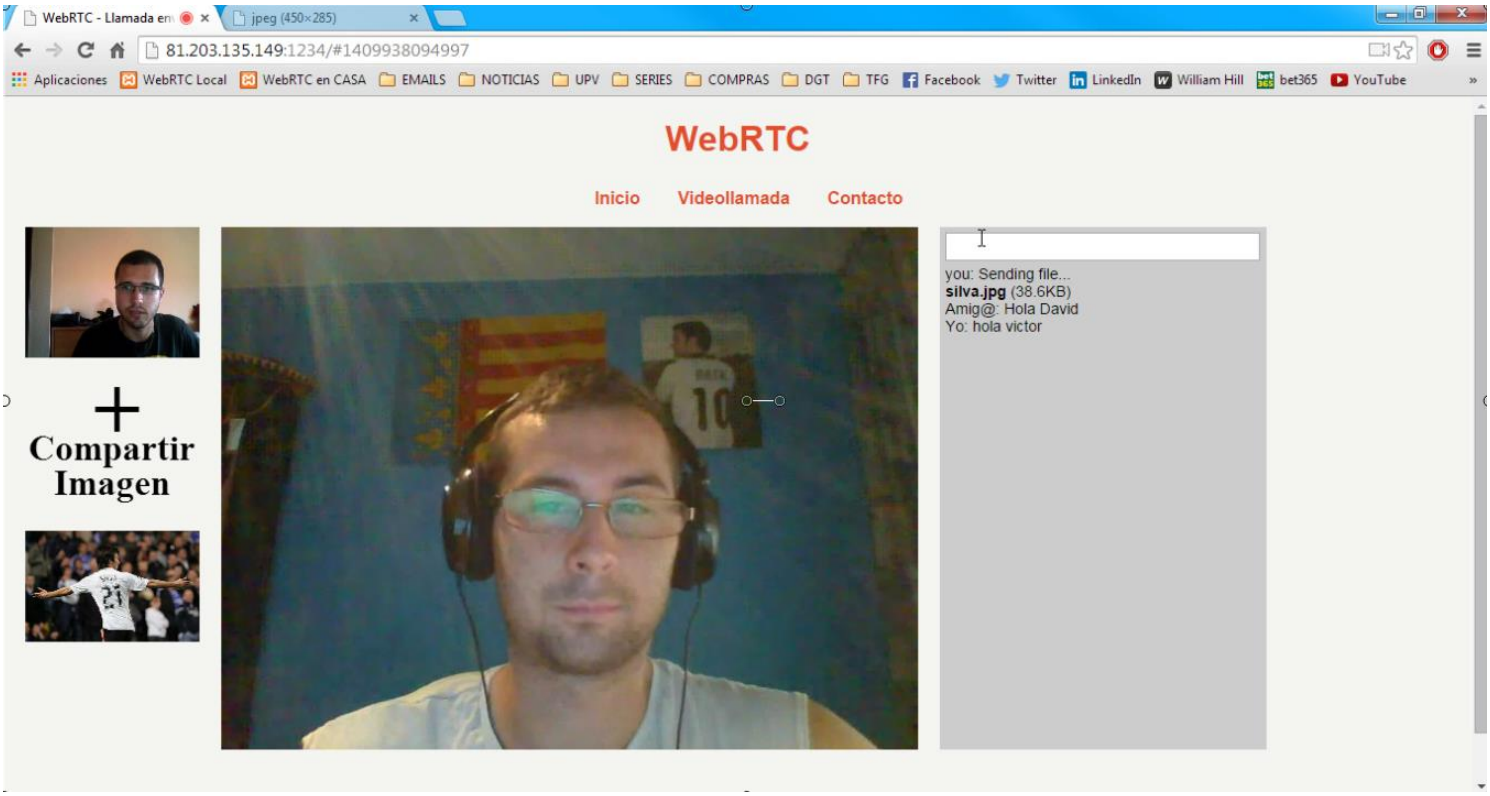
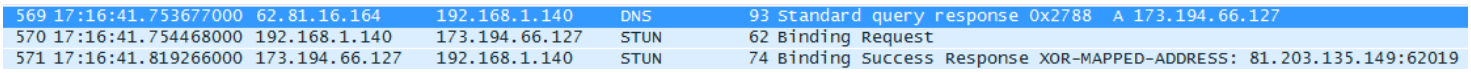


Figura 9: Vista del servidor web en una videollamada

Capítulo 6. Testeo WebRTC

Uno de los objetivos del trabajo final de grado es entender todo el proceso de unión de los peers para luego transmitir datos, ya sea una videollamada tal como la aplicación que hemos instalado y testeado o simplemente una transferencia de archivos.

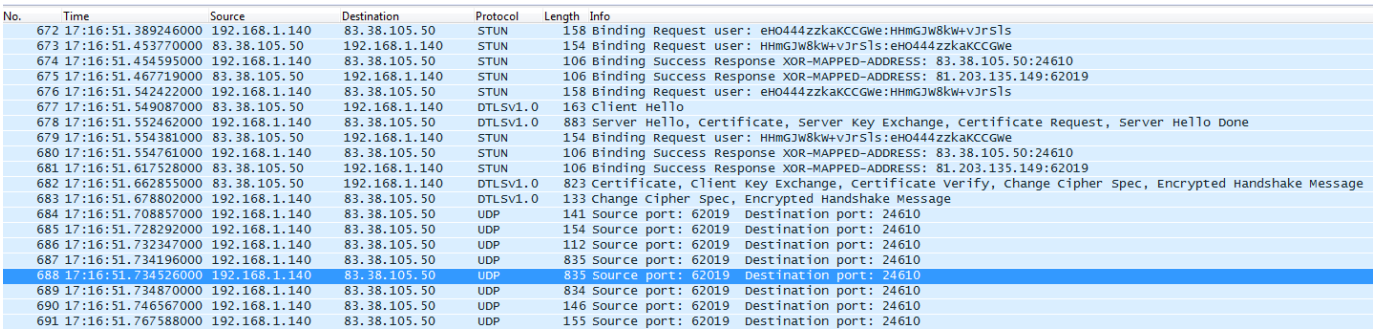
Para capturar el proceso se ha utilizado Wireshark [13] y la conexión se ha establecido entre 2 peers en diferentes redes para atravesar así dos NATs y comprobar que funciona correctamente.



569	17:16:41.753677000	62.81.16.164	192.168.1.140	DNS	93 Standard query response 0x2788 A 173.194.66.127
570	17:16:41.754468000	192.168.1.140	173.194.66.127	STUN	62 Binding Request
571	17:16:41.819266000	173.194.66.127	192.168.1.140	STUN	74 Binding Success Response XOR-MAPPED-ADDRESS: 81.203.135.149:62019

Figura 10: Consulta DNS, mapeo con servidor STUN

Esta primera figura es bastante simple, el cliente que ha iniciado la aplicación soy yo, con dirección IP 192.168.1.140, y he recibido la respuesta del DNS acerca del servidor STUN 173.194.66.127 perteneciente a Google y cuya función es mapearnos la IP pública de nuestro router con un puerto que se usará en el tráfico de la videollamada.



No.	Time	Source	Destination	Protocol	Length	Info
672	17:16:51.389246000	192.168.1.140	83.38.105.50	STUN	158	Binding Request user: eHO444zzkaKCCGWe:HHmGJW8kW+vJrSls
673	17:16:51.453770000	83.38.105.50	192.168.1.140	STUN	154	Binding Request user: HHmGJW8kW+vJrSls:eHO444zzkaKCCGWe
674	17:16:51.454595000	192.168.1.140	83.38.105.50	STUN	106	Binding Success Response XOR-MAPPED-ADDRESS: 83.38.105.50:24610
675	17:16:51.467719000	83.38.105.50	192.168.1.140	STUN	106	Binding Success Response XOR-MAPPED-ADDRESS: 81.203.135.149:62019
676	17:16:51.542422000	192.168.1.140	83.38.105.50	STUN	158	Binding Request user: eHO444zzkaKCCGWe:HHmGJW8kW+vJrSls
677	17:16:51.549087000	83.38.105.50	192.168.1.140	DTLSv1.0	163	Client Hello
678	17:16:51.552462000	192.168.1.140	83.38.105.50	DTLSv1.0	883	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
679	17:16:51.554381000	83.38.105.50	192.168.1.140	STUN	154	Binding Request user: HHmGJW8kW+vJrSls:eHO444zzkaKCCGWe
680	17:16:51.554761000	192.168.1.140	83.38.105.50	STUN	106	Binding Success Response XOR-MAPPED-ADDRESS: 83.38.105.50:24610
681	17:16:51.617528000	83.38.105.50	192.168.1.140	STUN	106	Binding Success Response XOR-MAPPED-ADDRESS: 81.203.135.149:62019
682	17:16:51.662855000	83.38.105.50	192.168.1.140	DTLSv1.0	823	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message
683	17:16:51.678802000	192.168.1.140	83.38.105.50	DTLSv1.0	133	Change Cipher Spec, Encrypted Handshake Message
684	17:16:51.708857000	192.168.1.140	83.38.105.50	UDP	141	Source port: 62019 Destination port: 24610
685	17:16:51.728292000	192.168.1.140	83.38.105.50	UDP	154	Source port: 62019 Destination port: 24610
686	17:16:51.732347000	192.168.1.140	83.38.105.50	UDP	112	Source port: 62019 Destination port: 24610
687	17:16:51.734196000	192.168.1.140	83.38.105.50	UDP	835	Source port: 62019 Destination port: 24610
688	17:16:51.734526000	192.168.1.140	83.38.105.50	UDP	835	Source port: 62019 Destination port: 24610
689	17:16:51.734870000	192.168.1.140	83.38.105.50	UDP	834	Source port: 62019 Destination port: 24610
690	17:16:51.746567000	192.168.1.140	83.38.105.50	UDP	146	Source port: 62019 Destination port: 24610
691	17:16:51.767588000	192.168.1.140	83.38.105.50	UDP	155	Source port: 62019 Destination port: 24610

Figura 11: Protocolo STUN, Negociación DTLS

Aquí se ha unido el otro peer, entonces se produce un intercambio de información acerca de la conexión mediante el protocolo STUN, donde para empezar se hace uso de unas etiquetas para identificar a los usuarios, que como podemos ver son un montón de caracteres sin mucho sentido.

En los siguientes paquetes se intercambia el mapeo de direcciones públicas y puertos que se van a usar en la conexión en cada extremo, que como podemos ver, uno es el 24610 y el otro 62019, lo cual quiere decir que son aleatorios.

Una vez mapeada la conexión, los extremos hacen uso del protocolo DTLS para establecer los mecanismos de cifrado, compresión, claves o certificados a emplear para conseguir dar confidencialidad a la sesión.

Ahora vamos a profundizar a ver cada uno de los paquetes que se usan en el establecimiento de la sesión con mayor detalle.

Este primer paquete que nos aparecía, era un “Binding Request User”, para empezar usa STUN para iniciar el establecimiento de la conexión, y añade atributos como USERNAME para identificar al cliente, y otros como MESSAGE-INTEGRITY y FINGERPRINT para preservar la integridad del mensaje y así aportar algo de seguridad.

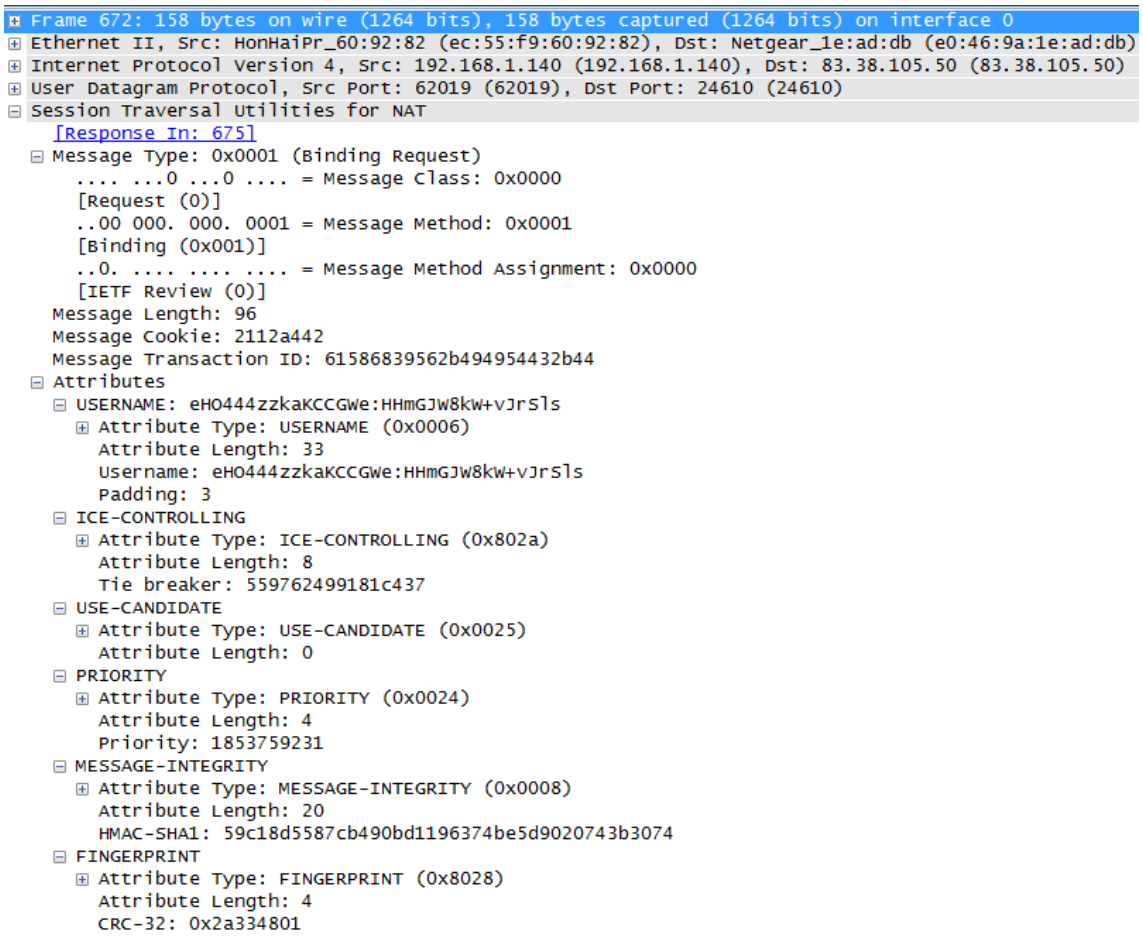


Figura 12: Paquete STUN, Binding Request User

A continuación recibimos (si todo ha ido bien) un paquete del tipo “Binding Success Response”, con los atributos ya conocidos como MESSAGE-INTEGRITY y FINGERPRINT. El atributo XOR-MAPPED-ADDRESS es el que nos proporciona la IP y puerto destino al que nos vamos a conectar.

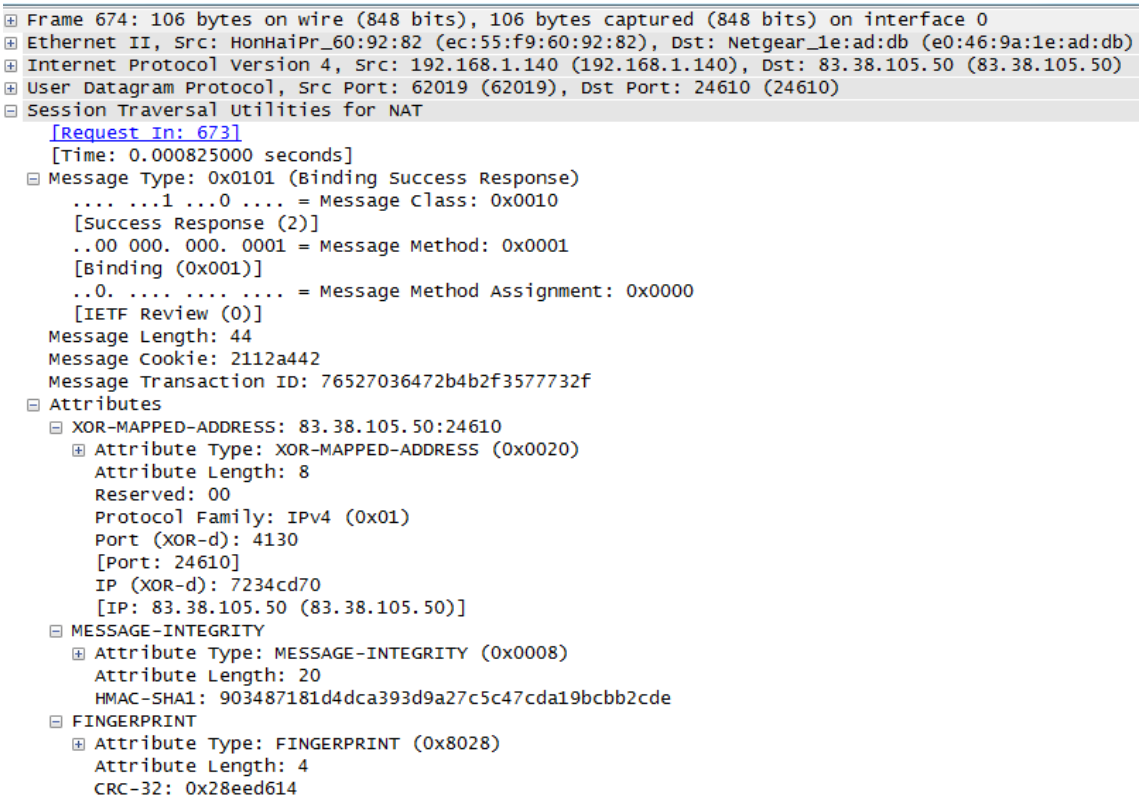


Figura 13: Paquete STUN, Binding Success Response

El siguiente paquete importante es del tipo DTLS, un “Client Hello” de Handshake Protocol cuya finalidad es establecer parámetros de seguridad (Cipher Suite, Compression, Extensions…) para asegurar la confidencialidad de la comunicación.

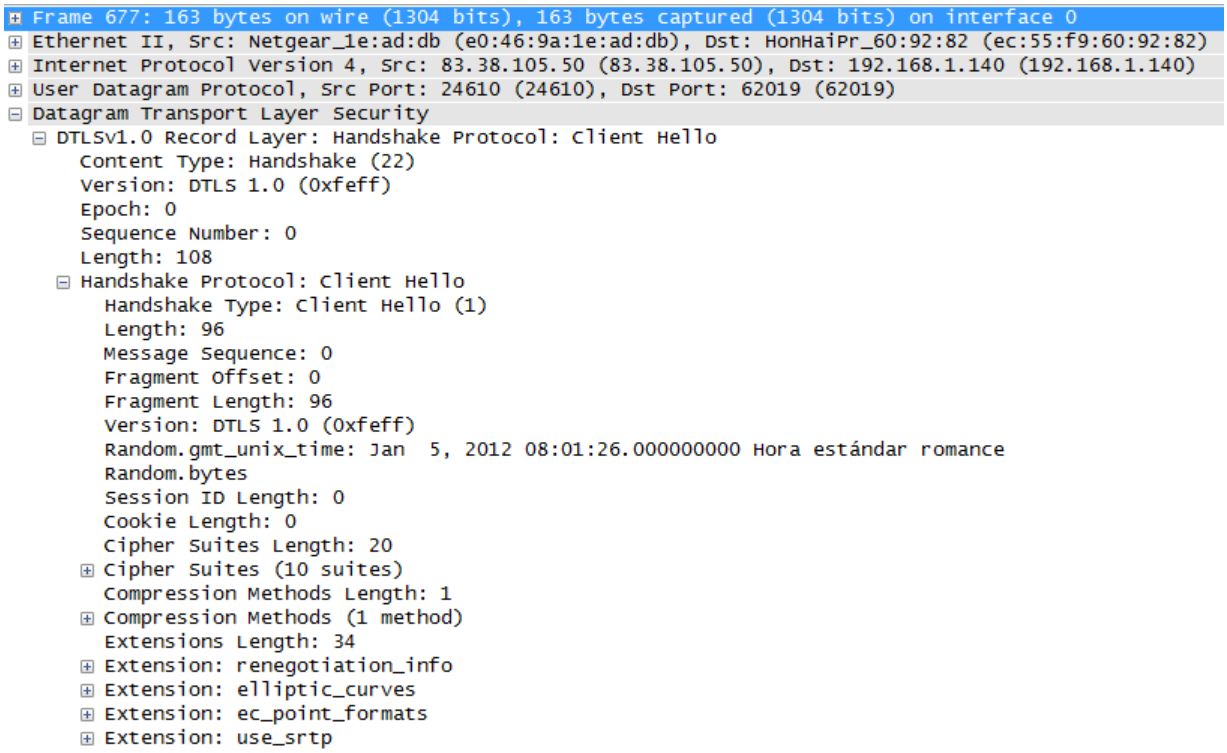


Figura 14: Paquete DTLS, Client Hello

La respuesta al anterior paquete es un “Server Hello”, donde usando el Handshake Protocol de nuevo, el servidor decide que algoritmos de cifrado, compresión, u otros parámetros utiliza y sus versiones.

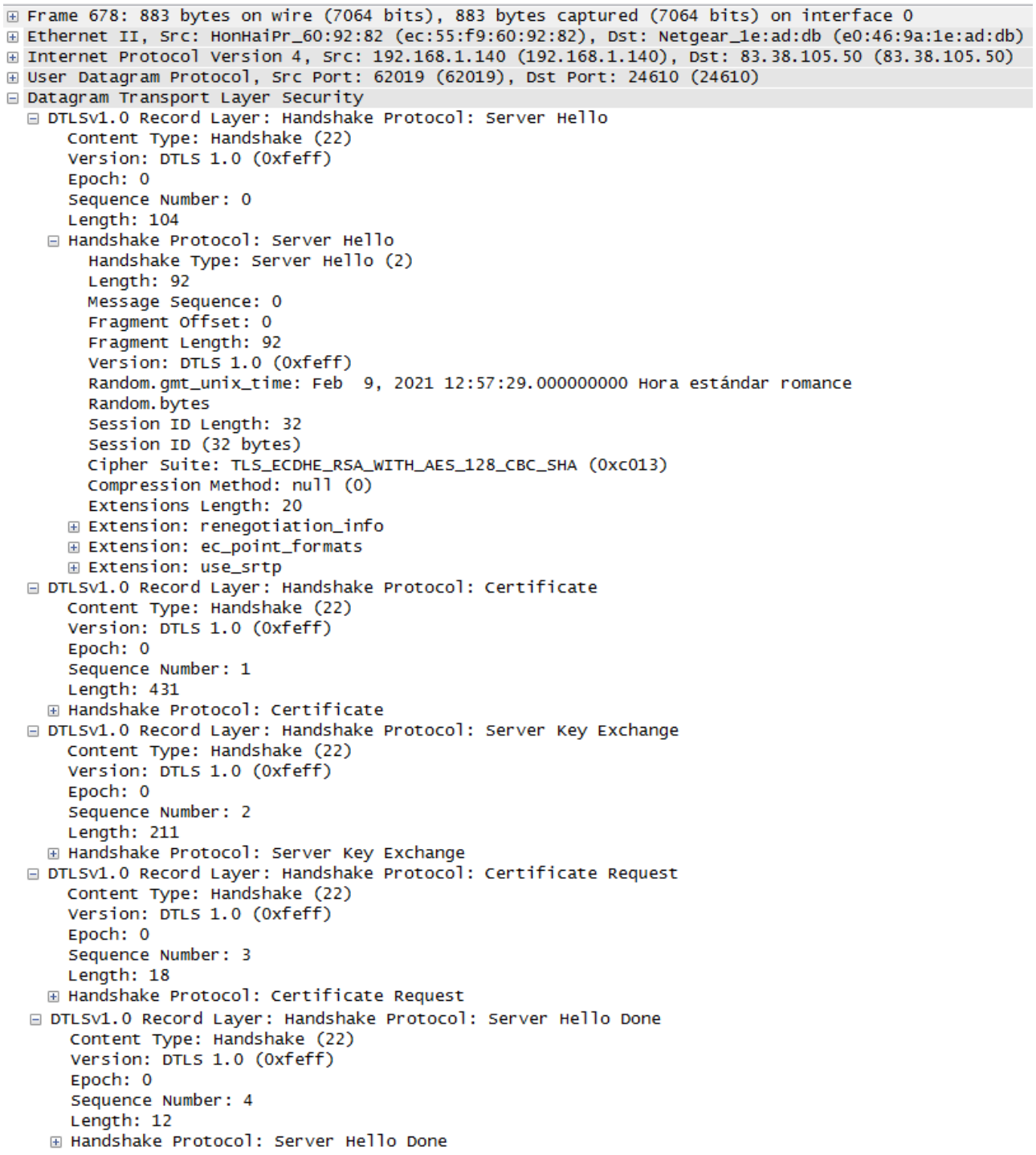


Figura 15: Paquete DTLS, Server Hello

De nuevo, el que inició la conversación con el “Client Hello” envía otro paquete para reclamar un cambio en el algoritmo de cifrado, y verifica el resto de parámetros dándolos por válidos.

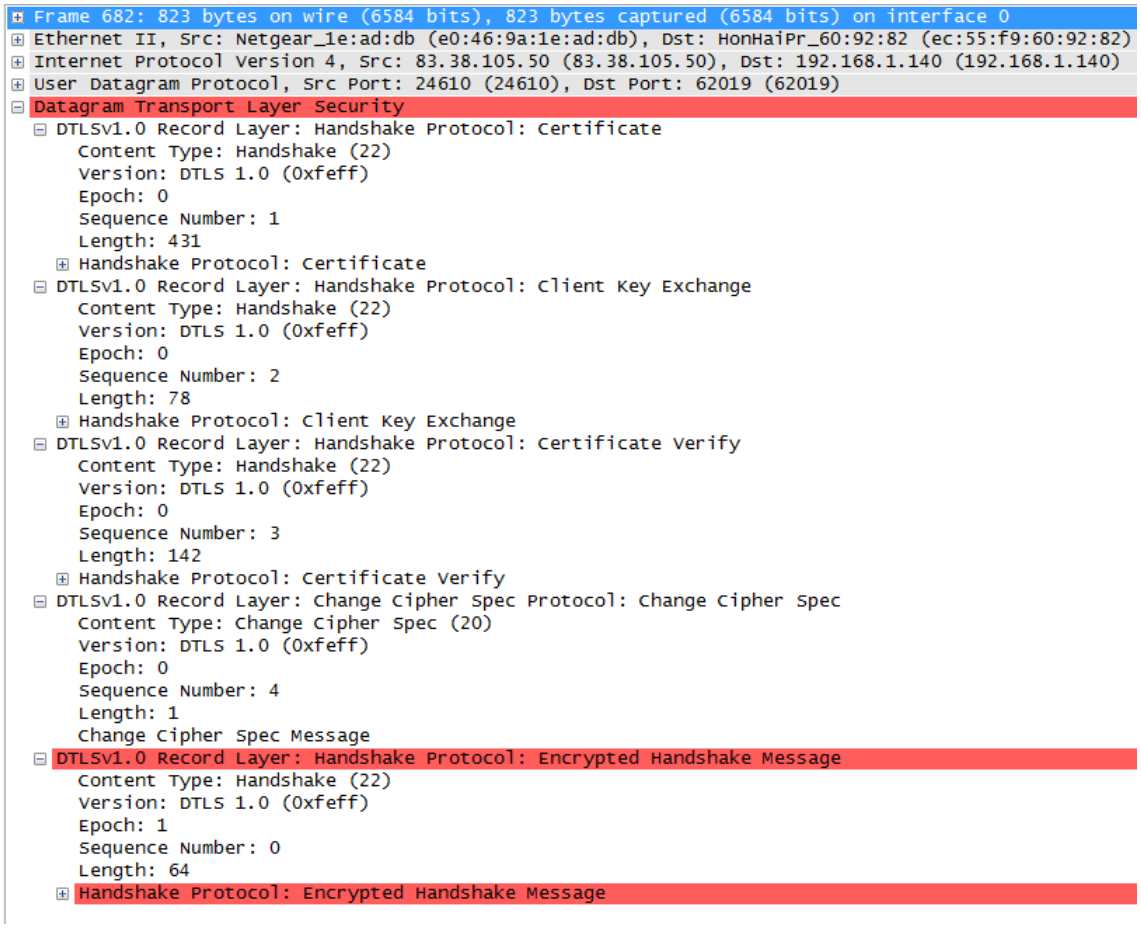


Figura 16: Paquete DTLS, Certificate, Verify, Change Cipher Spec

Y por último se acepta y se cambia el algoritmo de cifrado para por fin poder empezar a transmitir datos.

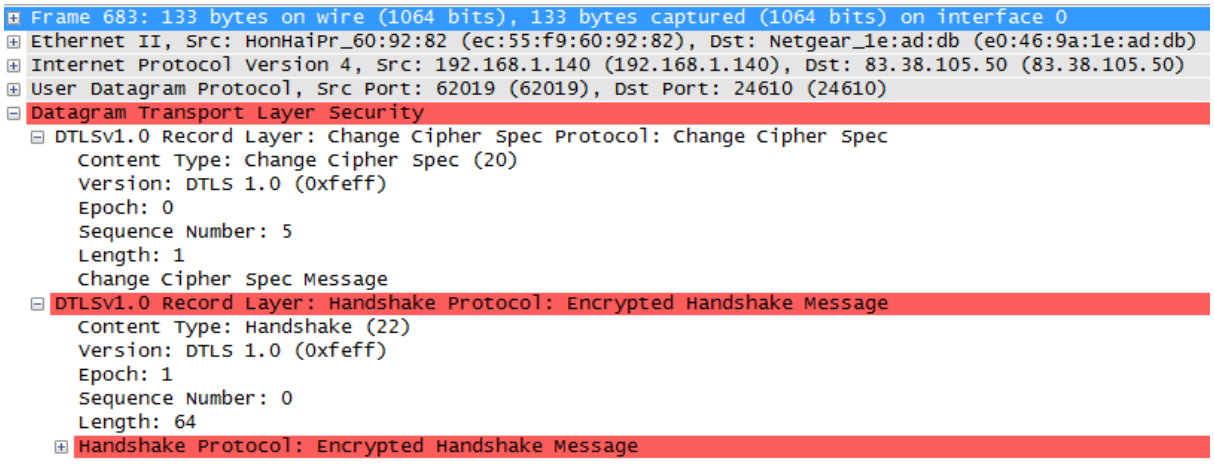


Figura 17: Paquete DTLS, Change Cipher Spec

Cabe destacar que todo este proceso es completamente transparente para el usuario, y en este caso mirando los tiempos de los paquetes nos damos cuenta que en menos de 1 segundo se ha negociado y se ha empezado a transmitir.

Ya que en nuestro caso se ha pedido un cambio de cifrado, hemos desplegado los algoritmos disponibles y tenemos:

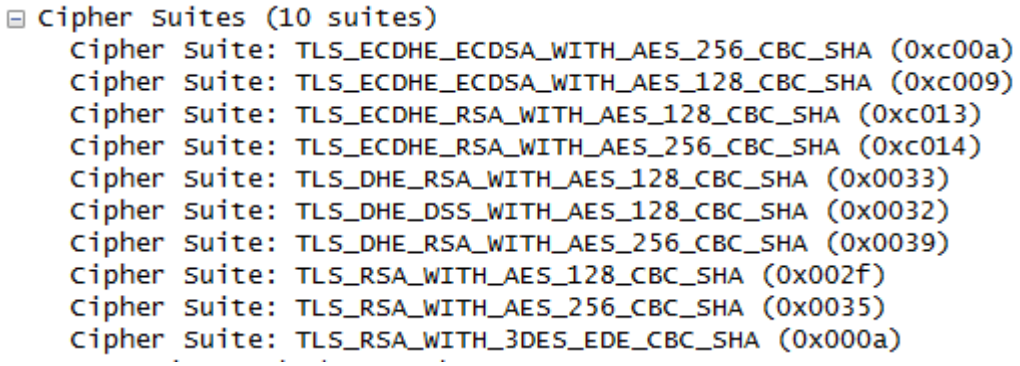


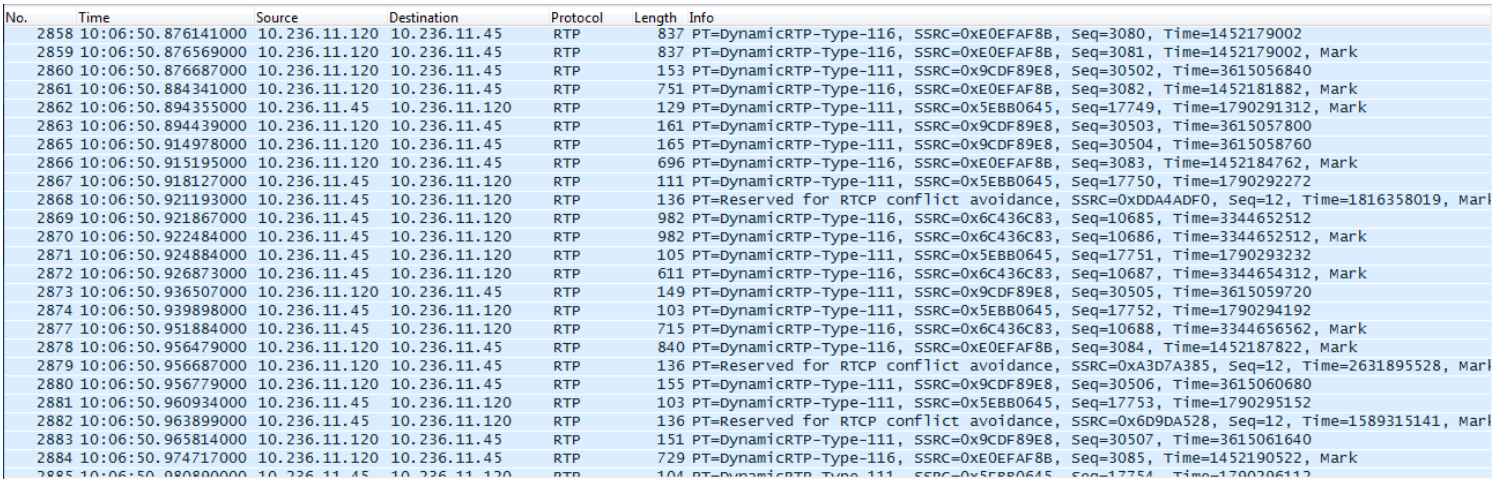
Figura 18: Tipos de Cifrado disponibles

ECDHE: Elliptic Curve Diffie-Hellman Exchange
ECDSA: Elliptic Curve Digital Signature Algorithm
AES: Advanced Encryption Standard
CBC: Cipher-block chaining
SHA: Secure Hash Algorithm
RSA: Rivest, Shamir y Adleman
DHE: Diffie–Hellman key exchange
DSS: Digital Signature Standard
3DES: Data Encryption Standard

Tenemos diferentes métodos de criptografía de clave pública con diferentes algoritmos de cifrado y al final con el mismo algoritmo de resumen o "hash" (con SHA mantener la integridad es bastante sencillo, pues alterar cualquier BYTE del paquete altera el "hash", en cambio para mantener privacidad es bueno poder cambiar los métodos de selección de claves y los tipos de cifrado).

Después de todo el proceso de establecimiento de la sesión, al utilizar la opción de WireShark “decode as” y poner RTP en lugar de UDP, podemos ver más datos todavía. [14]

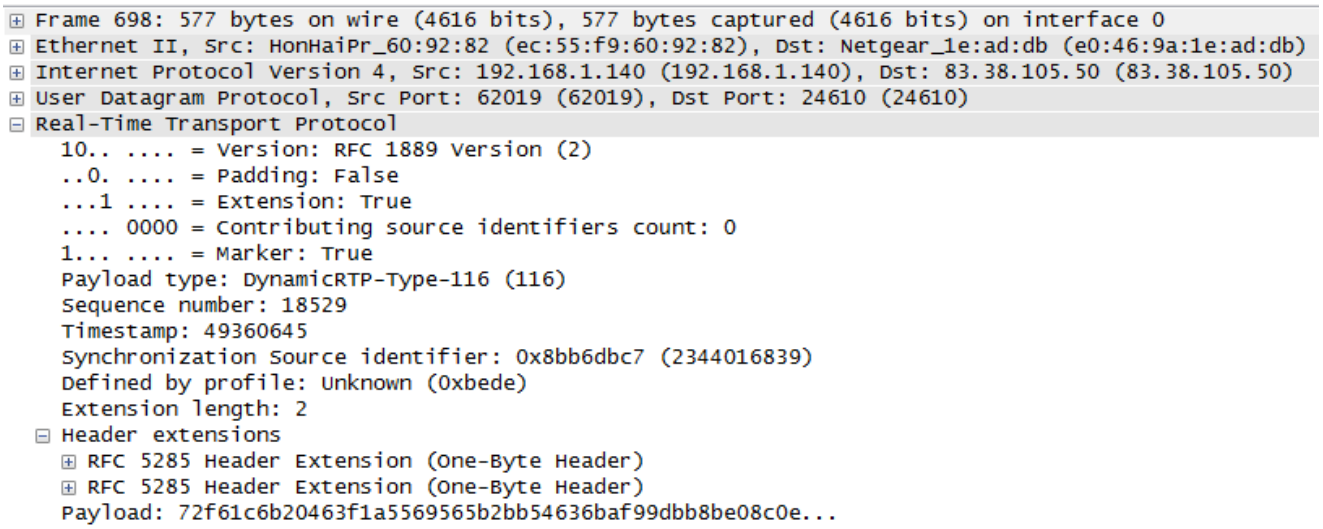
Los datos más relevantes son el PT (Payload Type) que básicamente define el códec empleado, ya sea de audio o video; el SSRC (Synchronization Source identifier), que identifica la fuente del flujo de audio o video; el Sequence Number se emplea para reordenar y reproducir los paquetes y por ultimo un Timestamp que marca cuando se generó el paquete (el primero de los paquetes lleva un numero aleatorio y el resto van a continuación).



No.	Time	Source	Destination	Protocol	Length	Info
2858	10:06:50.876141000	10.236.11.120	10.236.11.45	RTP	837	PT=DynamicRTP-Type-116, SSRC=0xE0EFAF8B, Seq=3080, Time=1452179002
2859	10:06:50.876569000	10.236.11.120	10.236.11.45	RTP	837	PT=DynamicRTP-Type-116, SSRC=0xE0EFAF8B, Seq=3081, Time=1452179002, Mark
2860	10:06:50.876687000	10.236.11.120	10.236.11.45	RTP	153	PT=DynamicRTP-Type-111, SSRC=0x9CDF89E8, Seq=30502, Time=3615056840
2861	10:06:50.884341000	10.236.11.120	10.236.11.45	RTP	751	PT=DynamicRTP-Type-116, SSRC=0xE0EFAF8B, Seq=3082, Time=1452181882, Mark
2862	10:06:50.894355000	10.236.11.45	10.236.11.120	RTP	129	PT=DynamicRTP-Type-111, SSRC=0x5EBB0645, Seq=17749, Time=1790291312, Mark
2863	10:06:50.894439000	10.236.11.120	10.236.11.45	RTP	161	PT=DynamicRTP-Type-111, SSRC=0x9CDF89E8, Seq=30503, Time=3615057800
2865	10:06:50.914978000	10.236.11.120	10.236.11.45	RTP	165	PT=DynamicRTP-Type-111, SSRC=0x9CDF89E8, Seq=30504, Time=3615058760
2866	10:06:50.915195000	10.236.11.120	10.236.11.45	RTP	696	PT=DynamicRTP-Type-116, SSRC=0xE0EFAF8B, Seq=3083, Time=1452184762, Mark
2867	10:06:50.918127000	10.236.11.45	10.236.11.120	RTP	111	PT=DynamicRTP-Type-111, SSRC=0x5EBB0645, Seq=17750, Time=1790292272
2868	10:06:50.921193000	10.236.11.45	10.236.11.120	RTP	136	PT=Reserved for RTCP conflict avoidance, SSRC=0xDDA4ADF0, Seq=12, Time=1816358019, Mark
2869	10:06:50.921867000	10.236.11.45	10.236.11.120	RTP	982	PT=DynamicRTP-Type-116, SSRC=0x6C436C83, Seq=10685, Time=3344652512
2870	10:06:50.922484000	10.236.11.45	10.236.11.120	RTP	982	PT=DynamicRTP-Type-116, SSRC=0x6C436C83, Seq=10686, Time=3344652512, Mark
2871	10:06:50.924884000	10.236.11.45	10.236.11.120	RTP	105	PT=DynamicRTP-Type-111, SSRC=0x5EBB0645, Seq=17751, Time=1790293232
2872	10:06:50.926873000	10.236.11.45	10.236.11.120	RTP	611	PT=DynamicRTP-Type-116, SSRC=0x6C436C83, Seq=10687, Time=3344654312, Mark
2873	10:06:50.936507000	10.236.11.120	10.236.11.45	RTP	149	PT=DynamicRTP-Type-111, SSRC=0x9CDF89E8, Seq=30505, Time=3615059720
2874	10:06:50.939898000	10.236.11.45	10.236.11.120	RTP	103	PT=DynamicRTP-Type-111, SSRC=0x5EBB0645, Seq=17752, Time=1790294192
2877	10:06:50.951884000	10.236.11.45	10.236.11.120	RTP	715	PT=DynamicRTP-Type-116, SSRC=0x6C436C83, Seq=10688, Time=3344656562, Mark
2878	10:06:50.956479000	10.236.11.120	10.236.11.45	RTP	840	PT=DynamicRTP-Type-116, SSRC=0xE0EFAF8B, Seq=3084, Time=1452187822, Mark
2879	10:06:50.956687000	10.236.11.120	10.236.11.45	RTP	136	PT=Reserved for RTCP conflict avoidance, SSRC=0xA3D7A385, Seq=12, Time=2631895528, Mark
2880	10:06:50.956779000	10.236.11.120	10.236.11.45	RTP	155	PT=DynamicRTP-Type-111, SSRC=0x9CDF89E8, Seq=30506, Time=3615060680
2881	10:06:50.960934000	10.236.11.45	10.236.11.120	RTP	103	PT=DynamicRTP-Type-111, SSRC=0x5EBB0645, Seq=17753, Time=1790295152
2882	10:06:50.963899000	10.236.11.45	10.236.11.120	RTP	136	PT=Reserved for RTCP conflict avoidance, SSRC=0x6D9DA528, Seq=12, Time=1589315141, Mark
2883	10:06:50.965814000	10.236.11.120	10.236.11.45	RTP	151	PT=DynamicRTP-Type-111, SSRC=0x9CDF89E8, Seq=30507, Time=3615061640
2884	10:06:50.974717000	10.236.11.120	10.236.11.45	RTP	729	PT=DynamicRTP-Type-116, SSRC=0xE0EFAF8B, Seq=3085, Time=1452190522, Mark

Figura 19: Flujo de paquetes RTP

Luego también si vemos un paquete más a fondo todavía podemos ver más datos como la versión que se emplea, si lleva Padding (relleno) o si aporta alguna extensión. El bit Marker es uso exclusivo de algunas aplicaciones y el CSRC Count indica el número de identificadores de fuentes que siguen a la cabecera fija del paquete que en nuestro caso es 0.



```
⊞ Frame 698: 577 bytes on wire (4616 bits), 577 bytes captured (4616 bits) on interface 0
⊞ Ethernet II, Src: HonHaiPr_60:92:82 (ec:55:f9:60:92:82), Dst: Netgear_1e:ad:db (e0:46:9a:1e:ad:db)
⊞ Internet Protocol Version 4, Src: 192.168.1.140 (192.168.1.140), Dst: 83.38.105.50 (83.38.105.50)
⊞ User Datagram Protocol, Src Port: 62019 (62019), Dst Port: 24610 (24610)
⊟ Real-Time Transport Protocol
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...1 .... = Extension: True
    .... 0000 = Contributing source identifiers count: 0
    1... .... = Marker: True
    Payload type: DynamicRTP-Type-116 (116)
    Sequence number: 18529
    Timestamp: 49360645
    Synchronization Source identifier: 0x8bb6dbc7 (2344016839)
    Defined by profile: Unknown (0xbede)
    Extension length: 2
  ⊟ Header extensions
    ⊞ RFC 5285 Header Extension (One-Byte Header)
    ⊞ RFC 5285 Header Extension (One-Byte Header)
    Payload: 72f61c6b20463f1a5569565b2bb54636baf99dbb8be08c0e...
```

Figura 20: Paquete RTP

Extrayendo estadísticas de una sesión que apenas duró un minuto, podemos analizar que apenas se consume ancho de banda señalizando y estableciendo la conexión, es casi al completo todo para el transporte de datos, y que éste además ha ocupado 1,243 Mbit/s, una velocidad bastante bien soportada por cualquier conexión en el hogar.

Protocol	% Packets	Packets	% Bytes	Bytes	Mbit/s	End Packets	End Bytes	End Mbit/s
Frame	100.00 %	7843	100.00 %	5105612	1.243	0	0	0.000
Ethernet	100.00 %	7843	100.00 %	5105612	1.243	0	0	0.000
Internet Protocol Version 4	100.00 %	7843	100.00 %	5105612	1.243	0	0	0.000
User Datagram Protocol	100.00 %	7843	100.00 %	5105612	1.243	0	0	0.000
Session Traversal Utilities for NAT	2.52 %	198	0.51 %	26124	0.006	198	26124	0.006
Datagram Transport Layer Security	0.05 %	4	0.04 %	2002	0.000	4	2002	0.000
Data	97.42 %	7641	99.45 %	5077486	1.236	7641	5077486	1.236

Figura 21: Estadísticas

Por último, quisiera mostrar el fichero de logs que se muestra en el servidor, una vez puesto en marcha, se queda escuchando en el puerto 1234; después recibe una primera conexión del tipo "join" a la que se le asigna un token; después recibe otra conexión con un token, que al analizarlo es válido y determina pues que la segunda conexión es el receptor de la llamada.

C:\xampp\htdocs\WebRTC>node webrtc_signal_server.js
Warning: Native modules not compiled. XOR performance will be degraded.
Warning: Native modules not compiled. UTF-8 validation disabled.
Fri Sep 05 2014 17:11:33 GMT+0200 (Hora de verano romance) server listening (port 1234)
Fri Sep 05 2014 17:16:21 GMT+0200 (Hora de verano romance) new request (http://81.203.135.149:1234)
Fri Sep 05 2014 17:16:21 GMT+0200 (Hora de verano romance) new connection (81.203.135.149)
Fri Sep 05 2014 17:16:21 GMT+0200 (Hora de verano romance) got message {"token":"#1409930181829","type":"join"}
Fri Sep 05 2014 17:16:41 GMT+0200 (Hora de verano romance) new request (http://81.203.135.149:1234)
Fri Sep 05 2014 17:16:41 GMT+0200 (Hora de verano romance) new connection (83.38.105.50)
Fri Sep 05 2014 17:16:41 GMT+0200 (Hora de verano romance) got message {"token":"#1409930181829","type":"join"}
Fri Sep 05 2014 17:16:41 GMT+0200 (Hora de verano romance) got message {"token":"#1409930181829","type":"callee_arrived"}

Y por último, podemos ver a continuación 2 logs de mensajes de texto enviados por el chat, y como se ha cerrado la conexión.

Fri Sep 05 2014 17:17:03 GMT+0200 (Hora de verano romance) got message {"token":"#1409930181829","type":"new_chat_message","message":"waittt"}
Fri Sep 05 2014 17:17:15 GMT+0200 (Hora de verano romance) got message {"token":"#1409930181829","type":"new_chat_message","message":"valep, cortamos"}
Fri Sep 05 2014 17:17:20 GMT+0200 (Hora de verano romance) connection closed (81.203.135.149 - 1006- Connection dropped by remote peer.)

Fri Sep 05 2014 17:17:23 GMT+0200 (Hora de verano romance) connection closed (83.38.105.50 - 1001 -Remote peer is going away)

Esta es la pinta que tiene un paquete SDP [15] que pasa por el servidor de señalización, como es tan complicado de entender por estar todo tan junto, he usado colores para distinguir los siguientes atributos:

- v (amarillo): versión del protocolo.
- o (amarillo): origen e identificador de sesión (en este caso es una oferta, por eso el origen es localhost).
- s (amarillo): Nombre de la sesión (no tiene)
- t (amarillo): tiempo durante el que la sesión estará activa (está a 0 porque es indefinido)
- a (gris): atributos de sesión.
- m (verde fosforescente): Nombre de medio y dirección de transporte.
- c (amarillo): información de conexión
- a (rojo): atributos de medios.
- a (turquesa): atributos de medios, en este caso son códecs, primero de audio y luego de video.
- a (verde oscuro): atributos de RTP, podemos ver diferentes SSRC, que se usan para identificar flujos RTP.

Fri Sep 05 2014 17:16:41 GMT+0200 (Hora de verano romance) got message {"token":"#1409930181829","type":"new_description","sdp":{"sdp":"**v=0** \r\n o=-7512300338533284817 2 IN IP4 127.0.0.1 \r\n s=- \r\n t=00 \r\n a=group:BUNDLE audio video \r\n a=msid-semantic: WMS Nx8K9PN3fk7DvH1VbAIN4izR1Vun54R1BsCO \r\n m=audio 1 RTP/SAVPF 111 103 104 0 8 106 105 13 126 \r\n c=IN IP4 0.0.0.0 \r\n a=rtcp:1 IN IP4 0.0.0.0 \r\n a=ice-ufrag:HHmGJW8kW+vJrSls \r\n a=ice-pwd:STM3FsFrbMQbTWA5xssnpfwY \r\n a=ice-options:google-ice \r\n a=fingerprint:sha-256 1D:FF:00:C8:2F:1F:AB:46:DB:54:A2:84:E3:60:DE:68:C7:C2:C8:D3:A1:84:4A:8B:45:CB:D6:EE:6D:C6:63:E5 \r\n a=setup:actpass \r\n a=mid:audio \r\n a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level \r\n a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time \r\n a=sendrecv \r\n a=rtcp-mux \r\n a=rtpmap:111 opus/48000/2 \r\n a=fmtp:111 minptime=10 \r\n a=rtpmap:103 ISAC/16000 \r\n a=rtpmap:104 ISAC/32000 \r\n a=rtpmap:0 PCMU/8000 \r\n a=rtpmap:8 PCMA/8000 \r\n a=rtpmap:106 CN/32000 \r\n a=rtpmap:105 CN/16000 \r\n a=rtpmap:13 CN/8000 \r\n a=rtpmap:126 telephone-event/8000 \r\n a=maxptime:60 \r\n a=ssrc:04961018 cname:Wt6xrtzLs4afF64u \r\n a=ssrc:804961018 msid:Nx8K9PN3fk7DvH1VbAIN4izR1Vun54R1BsCO 0c042 047-22ae-438f-bae2-a12b4ff75262 \r\n a=ssrc:804961018 mslabel:Nx8K9PN3fk7DvH1VbAIN4izR1Vun54R1BsCO \r\n a=ssrc:804961018 label:0c042047-22ae-438f-bae2-a12b4ff75262 \r\n m=video 1 RTP/SAVPF 100 116 117 \r\n c=IN IP4 0.0.0.0 \r\n a=rtcp:1 IN IP4 0.0.0.0 \r\n a=ice-ufrag:HHmGJW8kW+vJrSls \r\n a=ice-pwd:STM3FsFrbMQbTWA5xssnpfwY \r\n a=ice-options:google-ice \r\n a=fingerprint:sha-256 1D:FF:00:C8:2F:1F:AB:46:DB:54:A2:84:E3:60:DE:68:C7:C2:C8:D3:A1:84:4A:8B:45:CB:D6:EE:6D:C6:63:E5 \r\n a=setup:actpass \r\n a=mid:video \r\n a=extmap:2 urn:ietf:params:rtp-hdrext:toffset \r\n a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time \r\n a=sendrecv \r\n a=rtcp-mux \r\n a=rtpmap:100 VP8/90000 \r\n a=rtcp-fb:100 ccm fir \r\n a=rtcp-fb:100 nack \r\n a=rtcp-fb:100 nack pli \r\n a=rtcp-fb:100 goog-remb \r\n a=rtpmap:116 red/90000 \r\n a=rtpmap:117 ulpfec/90000 \r\n a=ssrc:2344016839 cname:Wt6xrtzLs4afF64u \r\n a=ssrc:2344016839 msid:Nx8K9PN3fk7DvH1VbAIN4izR1Vun54R1BsCO 81095f30-aba4-4770-83a5-5bf8cb135828 \r\n a=ssrc:2344016839 mslabel:Nx8K9PN3fk7DvH1VbAIN4izR1Vun54R1BsCO \r\n a=ssrc:2344016839 label:81095f30-aba4-4770-83a5-5bf8cb135828 \r\n ","type":"offer"}}

Capítulo 7. Conclusiones y propuesta de trabajo futuro

Trabajar en este proyecto ha sido enriquecedor, sobretodo el aspecto práctico, como se implementa y como funciona WebRTC. Siempre le intentamos ver que aplicación podemos darle a lo que hacemos fuera de la universidad y en este caso es una tecnología de comunicación muy potente. Estoy seguro de que día tras día cada vez habrá más aplicaciones gracias a ser de código abierto por google.

Personalmente, he tenido que aprender aspectos de HTML5, de CSS y de JavaScript que eran completamente nuevos para mí, y lo cual me ha llevado mucho tiempo controlar, además de instalar el servidor y dejarlo funcionando correctamente para que sea accesible desde el exterior de la red.

Con vistas al futuro, podemos dejar volar la imaginación, esto no es solo un servidor que podamos instalar en una empresa o un proveedor de servicios y únicamente comunicar videollamadas, puede tener aplicaciones en seguridad conectando cámaras de vigilancia a un servidor web, aplicaciones con sensores haciendo uso del DataChannel, ayuda para personas mayores mediante una pantalla interactiva donde pueda hablar con su médico, uso de dispositivos interactivos como Kinect junto a un Smart TV, uso en videojuegos online como comenté al principio de la memoria…

Y con gran ambición, propondría integrar WebRTC en la propia web de la ETSIT, ya que así profesores y alumnos podrían hacer tutorías por videoconferencia en caso de no poder hacerlo de forma presencial, o los propios profesores entre ellos comentar aspectos de asignaturas si por ejemplo uno de ellos se encuentra en otro país.

Otra de las grandes ventajas de usar HTML5 es su uso en dispositivos móviles, ya que al ser compatible no hace falta tener un diseño de web para estos aparatos, lo que permite ampliar todavía más el horizonte de WebRTC.

Capítulo 8. Anexos

8.1 VP8 Encode Parameters.

Aquí traigo algunas opciones para configurar en el códec VP8 que emplea WebRTC. [16]

Tipo	Nombre	Función
Básico	--width (w) --height (h)	Definen las dimensiones
Básico	--fps	Definen el número de fps objetivo
Básico	--target-bitrate	Bitrate objetivo, definido en kbits/segundo
Calidad/Velocidad	--Best --good --rt	Elige la mejor calidad, calidad intermedia, o en tiempo real para autoajustar velocidad y calidad automáticamente.
Calidad/Velocidad	--cpu-used	Valores de 0 a 5, ajusta la velocidad que le damos al codificador.
BitRate	--end-usage	Variable Bit Rate, Constant Bit Rate y Constrained Quality
BitRate	--cq-level	Cuando se usa modo Constrained level, se define un valor de 0 a 63 con el valor del quantizer para recomponer las imagenes.
BitRate	--buf-initial-sz --buf-optimal-sz --buf-sz	Tamaños de buffer.
Reensamblado	--drop-frame	0, deshabilitado, 1-100, % del buffer para que el codificador empiece a eliminar frames con el objetivo de llegar al buffer optimo
Reensamblado	--resize-allowed --resize-down --resize-up	Habilitado o no. Down y Up representan el nivel del buffer para empezar a cambiar el tamaño interno de imagen.
Otros	--noise-sensitivity	Filtrado de ruido
Otros	--timebase	Precisión de las marcas de tiempo a la salida, por defecto 1ms.

8.2 RTCPeerConnection

Aqui traigo algunas propiedades, manejadores y métodos que ofrece RTCPeerConnection [17]

PROPIEDADES	
iceConnectionState	Información del estado del Interactive Connectivity Establishment.
iceGatheringState	Información del estado cuando se está reuniendo información ICE.
localDescription	SDP local.
peerIdentity	Contiene un nombre de dominio (idp) y un nombre que representan la identidad remota.
remoteDescription	SDP remoto.
signalingState	Información de señalización de la conexión local, describe la oferta SDP.

EVENTOS	
onaddstream	Llamado al ejecutar "addstream", un MediaStream se ha añadido a la conexión
ondatachannel	Llamado cuando un RTCDaraChannel se añade.
onicecandidate	Llamado cuando un RTCICECandidate es añadido al script.
oniceconnectionstatechange	Llamado cuando "iceConnectionState" cambia.
onidentityresult	Llamado cuando se crea una nueva identidad o durante la creación de una oferta o respuesta.
onidpassertionerror	Llamado cuando aparece un error por identidad asociada
onidpvalidationerror	Llamado cuando aparece un error por identidad asociada
onnegotiationneeded	Llamado para establecer una futura negociación por el navegador
onpeeridentity	Llamado cuando se recibe y se establece una identidad remota.
onremovestream	Llamado cuando un MediaStream es eliminado de la conexión.
onsignalingstatechange	Llamado cuando "signalingState" cambia.

Los métodos más importantes han sido utilizados en la parte del cliente y explicados en el código, por lo que no los añado.

Capítulo 9. Bibliografía

[1] Google compra GIPS. http://www.noticiasdot.com/wp2/2010/05/19/google-compra-una-compania-de-audio-y-video-online-por-535-millones/ [online]

[2] WebRTC, la jugada maestra de Google: http://alt1040.com/2011/06/webrtc-la-jugada-maestra-de-google-para-relegar-a-skype [Online]

[3] WebRTC by Muaz Khan. https://www.webrtc-experiment.com/video-conferencing/ [Online]

[4] Talky: https://talky.io/ [Online]

[5] Apache https://www.apachefriends.org/es/index.html [Online]

[6] Mohan Palat; Justin Hart; "*WebRTC for Dummies*" Sonus Special Edition. John Wiley & Sons, Inc. 2014.

[7] Alan B. Johnston; Daniel C. Burnett; "*WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*". Digital Codex LLC. Junio 2013.

[8] Node.js http://nodejs.org/ [Online]

[9] Rob Manson: "Getting Started with WebRTC". Packt Publishing. Septiembre 2013

[10] HTML5 Rocks: "Getting Started with WebRTC" http://www.html5rocks.com/en/tutorials/webrtc/basics/

[11] Código Facilito: http://codigofacilito.com/cursos/HTML5 [Online] http://codigofacilito.com/cursos/CSS [Online]

[12] MCLibre: "Lista de propiedades CSS". http://www.mclibre.org/consultar/amaya/css/css_propiedades.html [Online]

[13] Wireshark: https://www.wireshark.org/ [Online]

[14] IANA: "Real-Time Transport Protocol (RTP) Parameters". http://www.iana.org/assignments/rtp-parameters/rtp-parameters.xhtml [Online]

[15] Wikipedia.es: "Session Description Protocol": http://es.wikipedia.org/wiki/Session_Description_Protocol

[16] Webm Project: "VP8 Encode Parameter Guide" http://www.webmproject.org/docs/encoder-parameters/ [Online]

[17] Mozilla Foundation: "RTCPeer Connection" https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection [Online]

[] W3C: "WebRTC 1.0: Real-Time Communication Between Browsers" W3C Editor's Draft 01 July 2014. http://dev.w3.org/2011/webrtc/editor/archives/20140704/webrtc.html [Online]

[] New Think Tank: http://www.newthinktank.com/web-design/learn-html/ [Online]