Capítulo 1

Introducción

El proyecto que explica esta memoria se encuadra dentro del manejo, control, recogida y procesado de datos de sensores y actuadores de un drone a distancia. El drone es un vehículo aéreo no tripulado al que podemos definir dentro de la robótica aérea. En las siguientes páginas se dará unas pinceladas sobre la robótica, su historia y uso actual. También hablaremos sobre los sistemas actuales de control y manejo de drones, y para finalizar daremos una visión global sobre las tecnologías existentes dentro de las Comunicaciones en Tiempo Real (RTC, Real Time Communications).

1.1. Robótica

Robótica es la rama de la ingeniería mecánica, ingeniería eléctrica, ingeniería electrónica y ciencia de la computación que se ocupa del diseño, construcción, operación, disposición estructural, manufactura y aplicación de los robots. Estos robots están diseñados para realizar tareas o trabajos que los humanos no podemos, por lo que requieren de cierta inteligencia. Las ciencias y tecnologías de las que depende son: el álgebra, los autómatas programables, las máquinas de estados, la mecánica o la informática.

Es una rama que no solo ha conseguido mediante el diseño y evolución grandes avances no solo en tareas que realizaban con anterioridad personas, sino ademas en otras que suponen una gran dificultad para ser realizadas por estas ya sea por su complejidad, como ensamblar elementos milimétricos en placas bases, o por realizarse en entornos peligrosos. Por otro lado también se han desarrollado robots domésticos para hacer nuestro día a día mas sencillo.

Morfología

Entre los componentes *hardware* que componen un robot se encuentran las fuentes de alimentación, para dotar de autonomía a los robots; sensores y actuadores, que se asemejan a los órganos sensoriales humanos y que sirven para obtener información del entorno que les rodea como temperatura u objetos próximos e interactuar con ellos; memoria, microprocesadores y dispositivos de comunicación, que es la electrónica que se comunica con el dispositivo remoto y permite gobernar los movimientos del robot.

Por otro lado tenemos el software, que es quien da la inteligencia al robot para llevar a cabo las funciones para las que fue diseñado.

Historia

El término *robot* fue publicado por primera vez por el checo Karel Capek en *Rossum's Universal Robots* (1921), mientras que la palabra *robótica* fue usada por primera vez en la obra *¡Embustero!* (*Liar!*), de Isaac Asimov, en 1941.

Los precursores de esta disciplina son los autómatas. Son máquinas con apariencia de seres animados que realizan movimientos propios de estos. Su origen se remonta a la prehistoria, con máquinas como la estatua de Memón en Etiopía, que emitía sonidos al recibir luz. Leonardo da Vinci (1452-1519) diseñó un león mecánico y consiguió que anduviese por una habitación.

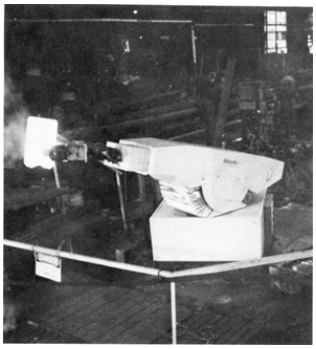
Es famoso el autómata de Henri Maillardet, creado alrededor del 1800, que realizaba varios dibujos y escribía poemas, pero siempre los mismos. Estos son solo algunos de los muchos ejemplos de autómatas en la historia.

Hasta la segunda mitad del siglo XX no aparecieron los primeros robots reales, fuera de la ciencia ficción. Los primeros modelos surgen en los años 50 y funcionaban en entornos muy controlados. Unimate (figura 1.1a) fue el primer robot comercial y se usó en la fabricación de automóviles. En los 60 fue desarrollada *la bestia* (figura 1.1b), un pequeño robot con capacidad para explorar paredes en busca de enchufes para recargarse.

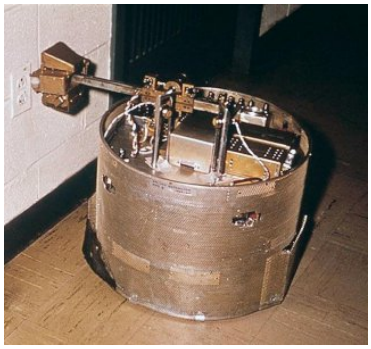
Otros ejemplos posteriores son *Shakey* y su sucesor *Flakey* (figura 1.1c), cuyo propósito era desplazarse y evitar obstáculos.

Ya en los 90 se desarrollaron las técnicas de creación de mapas y de

navegación en entornos no estructurados. *Xavier*[**?**] fue diseñado con este propósito.



(a) Brazo robótico Unimate.



(b) "La bestia".

(c) Flakey.

Figura 1.1: Algunos de los primeros robots.

A finales del siglo XX aparecen los primeros humanoides, robots con apariencia humana. Algunas funciones de los humanoides son realizar tareas de asistencia a enfermos o personas mayores, investigación para mejorar extremidades ortopédicas, entretenimiento, algunos trabajos como recepcionista o en el sector industrial, así como probar vehículos o herramientas diseñados para la forma humana. *ASIMO* es uno de estos humanoides, y es capaz de andar, correr, reconocer objetos en movimiento, gestos y posturas, y apartarse al encontrarse con gente, entre otros.

1.2. Robótica aérea

Esta rama de la robótica lleva experimentando desde hace varios años un auge espectacular. A este tipo de robots se les conoce también como Vehículo

Figura 1.2: Humanoide Asimo.

Aéreo No Tripulado ó UAV (*Unmanned Aerial Vehicle*), y de una manera mas coloquial también como *drones*. Se trata de aeronaves con capacidad de volar sin la presencia de un piloto a bordo que lo controle. Algunos de estos drones tienen la capacidad de volar de manera autónoma, aunque lo más comun es que haya un piloto u operador teleoperandolo desde tierra.

Historia

Sus orígenes se remontan a la Primera Guerra Mundial, cuando aparecieron los primeros blancos aéreos (1916). También aparecen entonces los precursores de los misiles (Aeroplano Automático de Hewitt-Sperry). Después de la guerra, estos objetivos pasan a volar por control remoto (*Fairey Queen*, 1931). El *Queen Bee*, en 1935, fue el primero de estos objetivos con capacidad de reutilización. En esta década el veterano británico de la Primera Guerra Mundial Reginald Denny trabajó en EEUU en la creación de varios de estos aeroplanos, desde el *RP-1* hasta el *RP-4*. También se investigó el uso de *drones* de ataque. Fueron usados, si bien de forma limitada, durante la Segunda Guerra Mundial (TDN-1, Project Fox...).

Los avances continúan durante la Guerra Fría. El *RP-71*, posteriormente conocido como *MQM-57 Falconer*, fue el primer BTT (*Basic Training Tar-*

gets) que evolucionó para ser utilizado con funciones de reconocimiento de terreno, y tuvo su primer vuelo en 1955. Drones de reconocimiento fueron usados también en la guerra de Vietnam, en China y Corea.

La modificación del modelo *B-17 Flying Fortress* permitió utilizar varios como drones para reunir datos radiactivos en pruebas nucleares siendo enviados cerca de la nube provocada por la explosión. Esto ocurrió en las pruebas realizadas en el Atolón Bikini.

En la década de los 70 fueron desarrollados en Israel varios modelos importantes. El *Firebee 1241* fue diseñado a partir del *Firebee* estadounidense, y se usó con funciones de reconocimiento y como señuelo.

El *Scout* fue un modelo ligero y pequeño difícil de detectar y derribar que transmitía datos procedentes de su radar, que realizaba barridos de 360º. Ya en los 80, Israel y EEUU desarrollaron el *Pioneer*. Este UAV podía volar por trayectorias preprogramadas, con piloto automático o controlado desde tierra. Era necesario monitorizar la posición del Pioneer con un enlace radio. Contaba con una autonomía de 5'5 horas. El ejército estadounidense lo utilizó en la Guerra del Golfo.

Clasificación y usos

Podemos encontrar varios tipos de drones atendiendo a su forma y componentes materiales. Los UAV de ala fija son similares a pequeños aviones y despegan y aterrizan del mismo modo. Estos son capaces de alcanzar altas velocidades. Un ejemplo es el UAV MAVinci. Otro tipo es el de fuselaje sustentador, el cual carece de alas y se sirve del propio cuerpo para producir la fuerza de sustentación que le permite volar. Los de ala rotatoria se asemejan a los helicópteros. Suelen tener cuatro o más motores (cuadricópteros, octocópteros, etc). Tienen la ventaja de poder permanecer cernidos en un punto fijo. Ejemplos de cuadricópteros son el Phantom 4 de DJI y el ArDrone 2.0 de Parrot.

En la actualidad los UAV tienen utilidad en múltiples campos. Algunos de ellos son los siguientes:

- **Militares.** Blancos móviles aéreos, reconocimiento de terreno y combate, entre otras tareas.
- **Vigilancia.** Seguridad en hogares, vigilancia de autopistas, costas, etc.

- **Inspección y reparaciones.** Fotografiar torres eléctricas, oleoductos, presas, gaseoductos, molinos eólicos, puentes, plataformas petrolíferas, etc, con el objetivo de vigilar o buscar daños que deban repararse.
- **Filmación.** Grabación de video para retransmisiones deportivas, anuncios o escenas de cine difíciles de grabar con cámaras convencionales.
- **Sondas de investigación.** Es posible enviar UAV para obtener datos a partir de sus sensores o tomar muestras de partículas, microorganismos, etc. Por ejemplo, se han realizado estudios de huracanes por medio de medidas de presión y temperatura tomadas por UAV enviados al huracán.
- **Rescates.** Es más eficiente para rescatar personas que hayan sufrido accidentes en el mar, montañas, u otras zonas de difícil acceso, o bien víctimas de desastres naturales, contar con la ayuda de UAV que faciliten la localización de supervivientes.
- **Detección de incendios.** Otra utilidad es la detección de focos de fuego, por ejemplo en incendios forestales. En general son útiles para la conservación de reservas naturales o zonas protegidas.

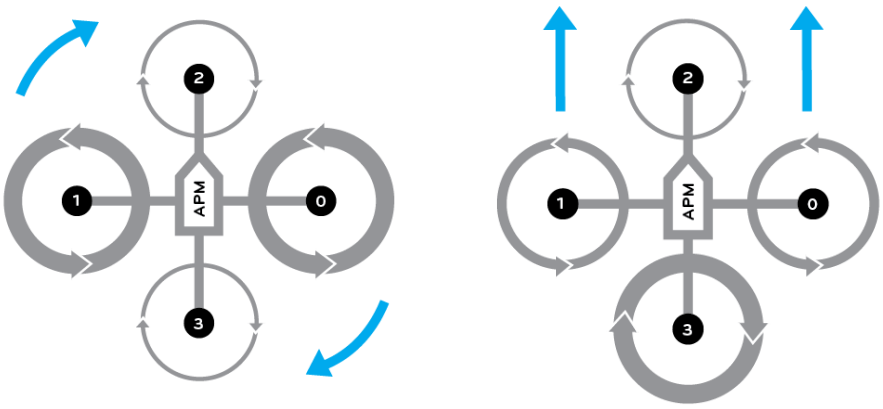
Cuadricópteros

Dentro de todos los tipos diferentes de vehículos UAV vamos a centrarnos en los cuadricópteros, y aunque no es necesario es conveniente que hablemos sobre la física que hace posible el vuelo de estos robots.

Un cuadricóptero utiliza los principios básicos de un helicóptero, utilizando cuatro rotores en vez uno. Estos rotores se acoplan a un esqueleto, el cual puede tener forma de 'x' o de '+'. Con la primera configuración tendríamos 2 motores delanteros y dos traseros (derecho e izquierdo), mientras que con la segunda configuración habría uno delantero, uno trasero y uno en cada lado.

Un problema que tienen que afrontar los helicópteros de un solo rotor, es que este produce una fuerza de torsión en el sentido de giro, por lo que es necesario otro rotor más pequeño perpendicular al principal para producir otra fuerza de sustentación que se oponga a la torsión y que el helicóptero no esté continuamente dando vueltas en torno a su eje vertical. En el caso de los cuadricópteros, al disponer de varios rotores, la solución es que el giro de las hélices de una misma extremidad sea opuesto al giro de las de la otra extremidad, de forma que las torsiones se anulen.

Dirigir y controlar el movimiento del vehículo se consigue variando la velocidad relativa de cada rotor para cambiar el empuje y el par motor de cada uno de ellos. Las hélices de los rotores, al girar, producen una fuerza de empuje hacia arriba llamada sustentación que es la que hace que se eleve el aparato. Esta fuerza es perpendicular a la velocidad del fluido relativa a la hélice y está contenida en el plano definido por la misma velocidad y la normal a la superficie de la hélice. Para que el drone despegue, la suma de fuerzas provocadas por cada rotor debe superar su peso. Una vez en el aire, si la suma de fuerzas es igual al peso, el drone permanecerá en una altitud fija o cernido (*hovering*). Para aterrizar, o desplazarse hacia abajo, es necesario hacer que la fuerza resultante sea algo menor que la del peso.



(a) Giro a la derecha. (b) Movimiento hacia delante.

Figura 1.3: Distintas configuraciones de los motores del cuadricóptero para desplazarse.

Para provocar un giro en sentido horario será preciso aumentar la potencia en los rotores con el sentido contrario, al mismo tiempo que se reduce proporcionalmente la potencia de los otros dos para que la fuerza de sustentación siga constante, ya que en caso contrario el robot se desplazaría en su eje z.

El movimiento hacia delante-atrás o hacia la derecha-izquierda se consigue disminuyendo la potencia de los rotores que estén en el lado hacia el cual se deba desplazar y aumentando los del lado contrario en igual proporción si el drone debe permanecer a una altura fija. Es decir, para movernos hacia la derecha habrá que disminuir la potencia de los rotores derechos y aumentar

la de los izquierdos, de forma que el drone se incline hacia la derecha y la fuerza de sustentación tenga una componente horizontal no nula.

1.3. Sistemas de control de drones

Los drones pertenecen a la rama de robótica aérea, pero a su vez también son vehículos aéreos no tripulados (*UAV, Unmanned Aerial Vehicle*). Es un vehículo no tripulado, pero no autónomo, por lo que necesitan ser teleoperados desde tierra. Los sistemas actuales para ello se pueden dividir en dos grupos, los controlados mediante radiofrecuencia y los que usan sistemas alternativos.

1.3.1. Radiocontrol

Es la técnica que permite el gobierno de un objeto a distancia de manera cámbrica mediante una emisora de control remoto. Por otra parte, a bordo del vehículo, en nuestro caso un drone, debe ir una receptora de radio control.

La comunicación entre receptor y transmisor se efectúa mediante radiofrecuencia, existiendo diferentes sistemas de emisión, como AM, FM o 2.4Ghz con diferentes tipo de codificación, PCM, PPM...

Estos sistemas tienen varias limitaciones. Una es el número de canales máximo del sistema, ya que se usa un canal para cada elemento de control disponible: elevación, giro, rotación... La segunda y posiblemente más crítica son las interferencias. Si se producen interferencias ya sea por ruido o por varios emisores trabajando en las cercanías se puede perder el control de la aeronave produciendo una posible colisión, destruir la misma o incluso dañar a personas.

1.3.2. Sistemas alternativos

En la actualidad a parte del radiocontrol tenemos el control a través de WiFi. Este sistema consiste en la creación de una red WiFi por parte del drone a la cual se conecta el dispositivo con el que se maneja. Este dispositivo puede ser un mando diseñado y comercializado por la propia marca o un dispositivo móvil, el cuál usa una aplicación que es la que gestiona la

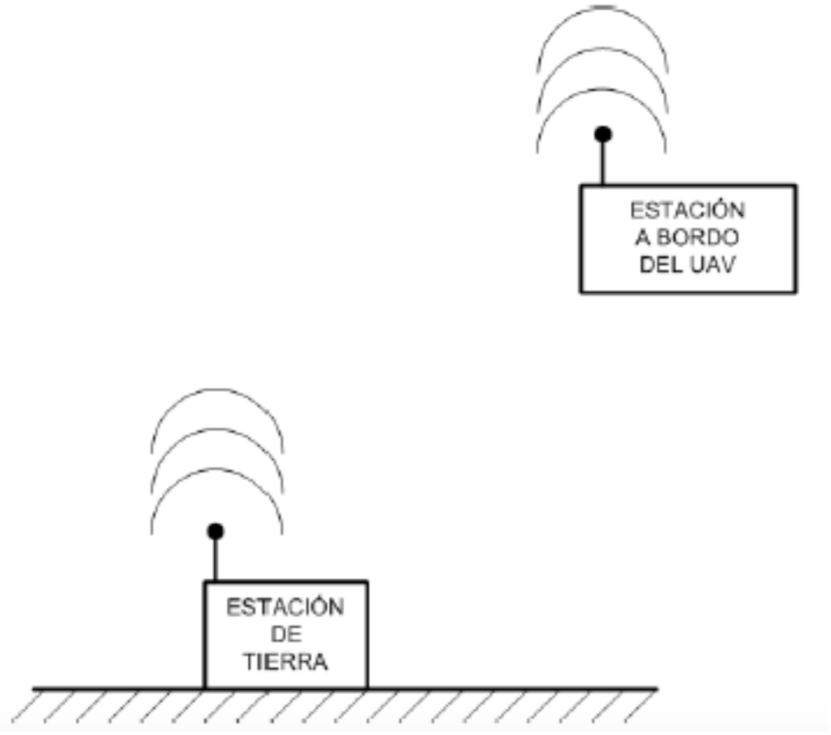


Figura 1.4: Sistema RC para estaciones UAV

conexión y transferencia de datos.

La ventaja de estos sistemas es e ahorro de batería, ya que los transmisores y antenas WiFi necesitan de menos potencia para cubrir las mismas distancias que los sistemas tradicionales de radiocontrol. Además el ancho de banda que nos proporciona es bastante elevado y podremos transferir tanto datos como las imágenes de las cámaras HD a bordo del drone.

Empresas como DJI usa sistemas mixtos que consisten en teleoperar el drone vía radiocontrol, pero la gestión de la cámara, con visualización incluida se realiza mediante una conexión WiFi como la explicada anteriormente.

La empresa francesa Parrot, la cual tiene una flota de diversos modelos de drones, utiliza un sistema de red WiFi, a la cuál conectas un dispositivo móvil ya sea Android o iOS, y mediante una aplicación desarrollada por ellos se puede teleoperar el drone, así como tener otras funcionalidades como la grabación de vídeo, captura de imágenes y gestión de parámetros de vuelo como altura o velocidad máxima. En capítulos posteriores profundizaremos

en este sistema implementado, ya que es el que usaremos para desarrollar el nuestro.

1.4. Tecnologías de Comunicación a Distancia en Tiempo Real

En la actualidad existen numerosas tecnologías de comunicación en tiempo real, pero nos centraremos en las que nos ofrecen conectividad multimedia. Primero veremos protocolos que se pueden implementar en aplicaciones de escritorio, y posteriormente veremos los protocolos web mas actuales.

1.4.1. RTP

RTP son las siglas de *Real-time Transport Protocol* o Protocolo de Transporte en Tiempo Real, el cuál es un protocolo de escritorio y de nivel de sesión utilizado para la transmisión de información en tiempo real, como por ejemplo audio, vídeo y datos. Está desarrollado por el grupo de trabajo de transporte de audio y vídeo del IETF (*Internet Engineering Task Force*). Este protocolo es la base de la industria de Voz sobre IP (*VoIP*).

Se encapsula sobre UDP y usa un puerto de usuario para cada medio que transfiere y admite direcciones de destino tanto *unicast* como *multicast*. Se encarga de enviar cualquier tipo de trama generada por cualquier algoritmo de codificación como H261, MPEG-1, MPEG-2... pero no añade ningún tipo de fiabilidad ni de calidad del servicio (*QoS*). Lo único que incorpora son marcas de tiempo para evitar el tembleque o *jitter* y la sincronización entre flujos en el destino y números de secuencia para detectar pérdidas en un flujo.

RTP trabaja junto con otros dos protocolos que lo complementan. El primero es RTCP (*Real time Control Protocol*), protocolo que proporciona información de control sobre la calidad de la transmisión. Transmite paquetes periódicos asociados a cada flujo RTP que incluye los detalles sobre los participantes, si hubiese más de uno, y las estadísticas de pérdidas que permiten el control de flujo y congestión. Según estas estadistas se puede hacer codificación adaptativa para adaptarse al medio. También trabaja sobre UDP y usa un numero de puerto superior al que usa el flujo de RTP.

El segundo es RTSP (*Real Time Streaming Protocol*), protocolo que permite realizar un control remoto de sesión de transmisión multimedia. Es un protocolo independiente del protocolo de transporte, basado en texto que permite recuperar un determinado medio de un servidor o grabar una multiconferencia.

La norma define también el protoclo SRTP (*Secure Real-time Transport Protocol*), el cuál es una extensión del perfil de RTP para conferencias de audio y vídeo que puede usarse para proporcionar confidencialidad, autenticación de mensajes y protección de reenvío para flujos de audio y vídeo.

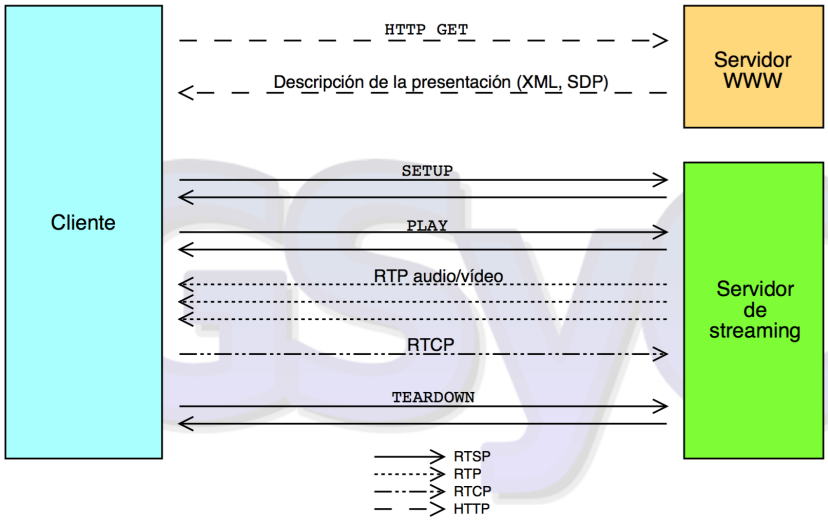


Figura 1.5: Ejemplo de conexión RTC, RTCP y RTSP

1.4.2. SIP

SIP o Protocolo de Inicio de Sesiones (*Session Initiation Protocol*) es un protocolo desarrollado por el grupo de trabajo MMUSIC del IETF con la intención de ser el estándar para la iniciación, modificación y finalización de sesiones interactivas de usuario donde intervienen elementos multimedia como vídeo, voz, mensajería instantánea...

Técnicamente no es un protocolo que transmite flujos multimedia, si no que es un protocolo de señalización cuya función es preparar el establecimiento y la terminación de sesiones multimedia entre máquinas remotas. Una de sus mas importante funciones es el intercambio de las descripciones de sesión

(*SDP*) de los usuarios. El concepto de sesión en este protocolo es muy amplio: una llamada entre dos, una videoconferencia, un juego interactivo entre varios usuarios...

Es un protocolo muy ligero. Tiene solo 6 métodos basados en texto, de manera similar a HTTP o SMTP, pero es completamente independiente del protocolo de transporte utilizado (TCP, UDP, ATM, etc).

Es el protocolo señalizador para el protocolo RTP explicado anteriormente. Entre otras, Ekiga, WengoPhone, MS Windows Messenger, Apple iChat AV ó Asterisk son algunas aplicaciones que utilizan SIP.

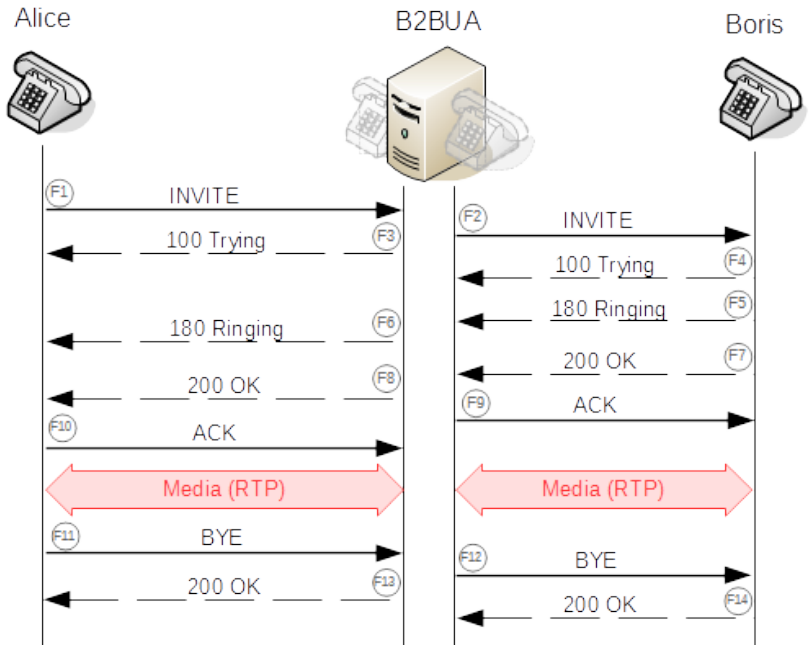


Figura 1.6: Ejemplo de conexión SIP

En la figura 1.6 podemos ver un ejemplo de una conexión multimedia con RTP que utiliza el protocolo SIP para establecer y finalizar la sesión entre los dos usuarios.

1.4.3. ORTC

Object RTC es un proyecto de código abierto que permite la comunicación en tiempo real (*RTC, Real-Time Communications*) de dispositivos móviles

con servidores u otros navegadores con el simple uso de unas API's JavaScript nativas en el navegador.

El objetivo de Object RTC es permitir crear comunicaciones en tiempo real con una alta calidad en dispositivos móviles y servidores con el simple uso de JavaScript y HTML5. Es también una obligación para ORTC ser compatible con WebRTC.

Aunque ORTC es un proyecto respaldado por empresas de la talla de Hookflash, Microsoft o Google, por el momento no es una especificación del consorcio W3C (*World Wide Web Consortium*).

ORTC comparte muchas similitudes con WebRTC, como mayor diferencia tenemos que ORTC no utiliza SDP ni el protocolo de Oferta/Respuesta, en cambio utiliza los objetos 'enviador' (*sender*), 'recibidor' (*receiver*) y 'transporte' (*transport*), los cuales tienen capacidades que describen que pueden hacer y sus parámetros que definen como están configurados. Además ORTC no está disponible nada más que en el navegador Edge de Microsoft.

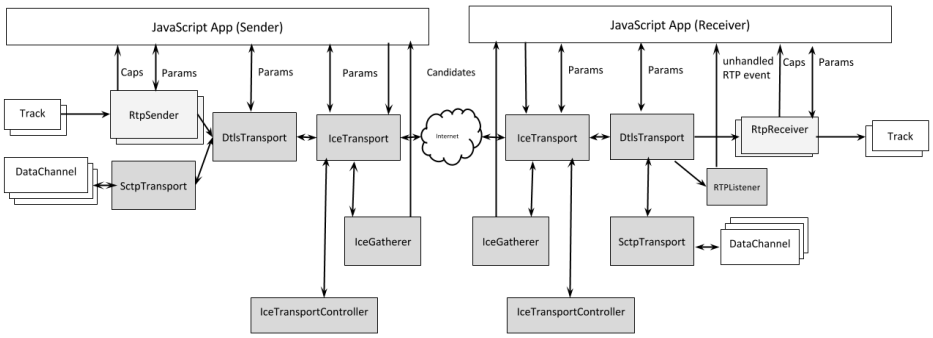


Figura 1.7: API de ORTC

1.4.4. WebRTC

En mayo de 2011 Google liberó un proyecto de código abierto basado en la comunicación entre navegadores en tiempo real. El proyecto ha sido continuado estandarizando los protocolos en el IETF y las API's de JavaScript en el W3C.

En el consorcio W3C WebRTC es aún un borrador de un proyecto en marcha el cuál esta altamente implementado en los navegadores como Mozilla

Firefox y Google Chrome. La API está basada en el trabajo previo realizado por *Web Hypertext Application Technology Working Group* (WHATWG).

Esta tecnología nos brinda la capacidad de crear numerosas aplicaciones de comunicaciones directamente en el navegador, sin necesidad de servidores internos, y está llamada a ser el futuro de las comunicaciones en tiempo real.

Figura 1.8: WebRTC

1.5. Motivación y punto de partida

En estos últimos años el desarrollo de vehículos no tripulados ha tenido un avance muy significativo, sobre todo en aplicaciones de uso civil. Las tecnologías web son otro campo que ha experimentado un avance enorme en los últimos años, permitiendo crear aplicaciones mas complejas y elaboradas.

La posibilidad de aunar estos dos campos son a para mi un gran aliciente y un gran reto a la vez. El fondo del proyecto consiste es desarrollar una aplicación web con tecnologías de última generación que permita teleoperar el drone.

Como base para el proyecto tenemos los siguientes proyectos, desarrollados también por alumnos de la URJC.

1.5.1. Surveillance 4.0 (URJC)

Surveillance 4.0 desarrollado por Daniel Castellano como su Proyecto Fin de Carrera. Esta aplicación contaba con varios sensores de distinto tipo (humedad, temperatura, gas, etc) que se conectaban inalámbricamente con un

nodo central situado en una Raspberry Pi. La conexión inalámbrica se hacía mediante transmisores Zigbee con un protocolo propio llamado WHAP. El nodo central recibía los datos de los sensores y los mostraba mediante un servidor web que corría en la misma máquina. La aplicación web se desarrolló en Python usando el entorno de desarrollo web Django. En Surveillance 4.0, los valores de los sensores se guardaban en una base de datos que la aplicación web consultaba cuando era necesario. Además, esta versión incluía un streaming de vídeo utilizando el software de código abierto M-JPEG Streamer. En la figura 1.9 se puede ver la aplicación.

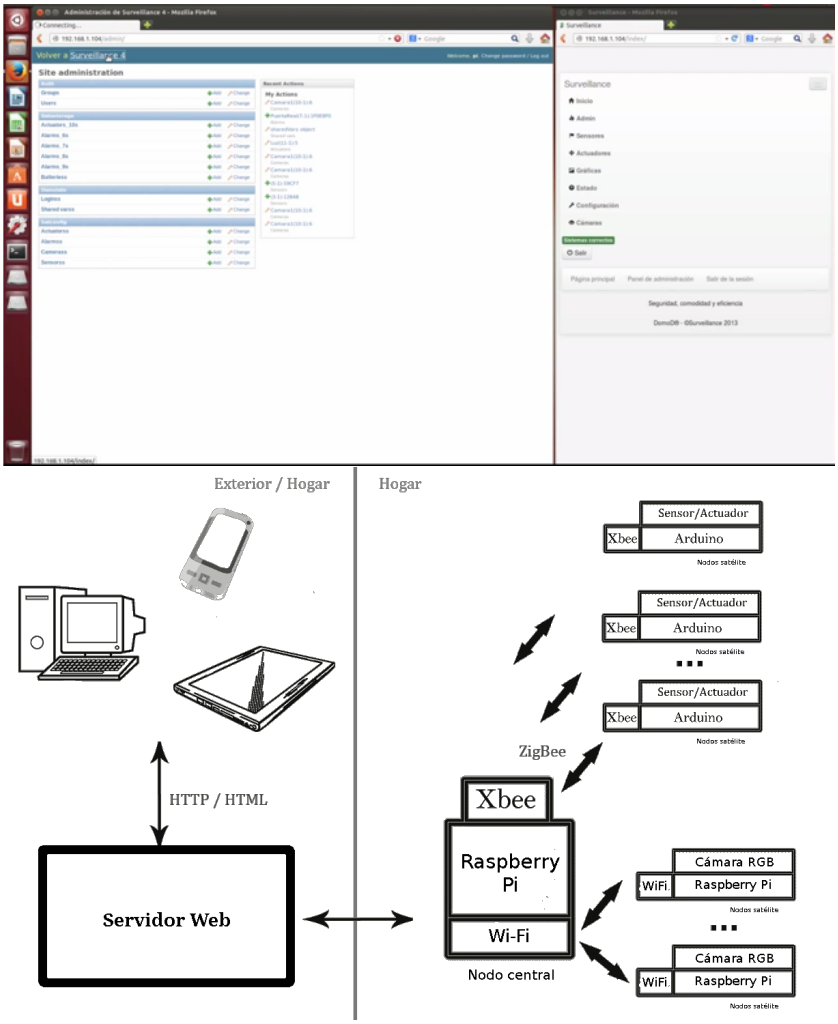


Figura 1.9: interfaz (a) y arquitectura (b).

1.5.2. Surveillance 5.1 (URJC)

Surveillance 5.1 desarrollado por Edgar Barrero como su Trabajo Fin de Grado. Esta aplicación obtenía un flujo de imágenes de una cámara web, un flujo de imágenes de profundidad de un sensor Kinect, además de datos de un sensor de humedad y de interaccionar con un actuador. La aplicación web se desarrolló en Ruby sobre Rails. En Surveillance 5.1, el servidor web se conectaba a los componente de JdeRobot mediante sus interfaces ICE. La aplicación web refrescaba estos datos mediante peticiones AJAX. En la figura 1.10 se puede ver la aplicación.

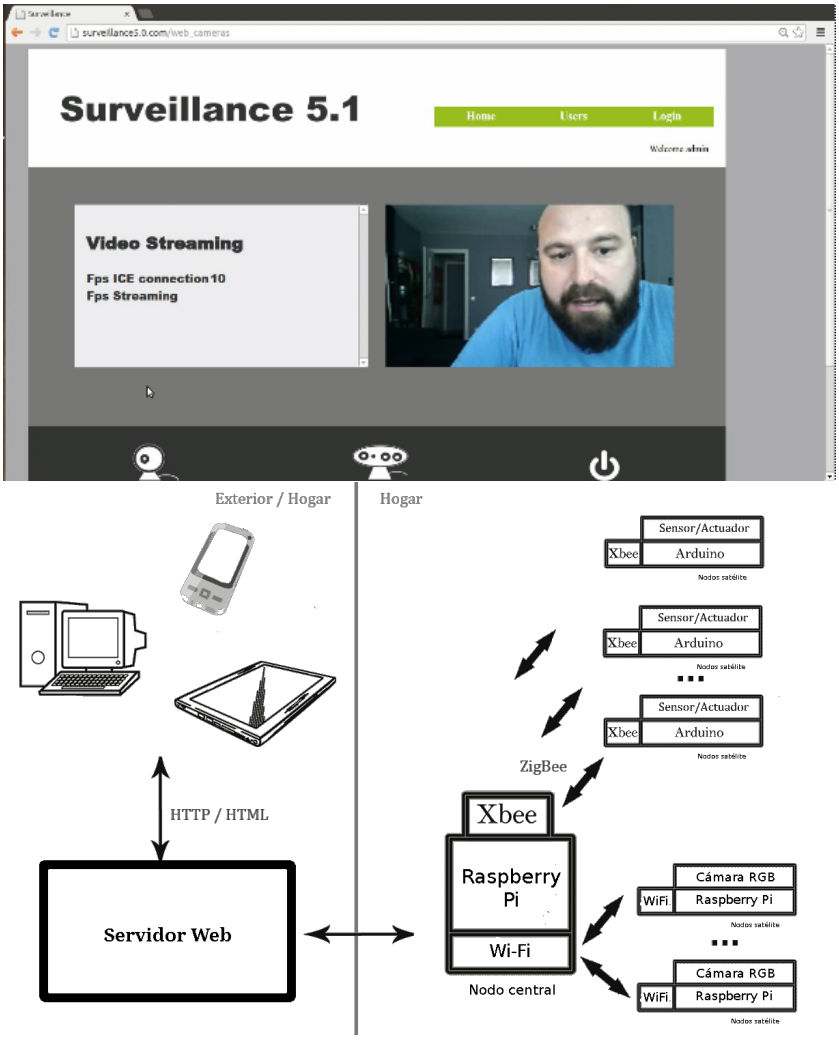


Figura 1.10: interfaz (a) y arquitectura (b).

Capítulo 2

Infraestructura Software

Una vez presentados los objetivos que tenemos marcados hay que echar una mirada a las tecnologías software y hardware que hemos utilizado como base en el proyecto. La más importante y sobre la que está centrado el proyecto es WebRTC. Además hemos emleado, como apoyo para cubrir las partes que WebRTC no llega, ICEJS junto con la plataforma robótica JdeRobot. Estas tecnologías junto con el simulador robótico Gazebo nos ha permitido probar, depurar y mejorar el código antes de realizar los experimentos sobre el drone real ArDrone, el cuál presento a continuación.

2.1. ArDrone de Parrot

El Parrot ARDrone es un drone que comercializa la marca Parrot el cuál puede ser teleoperado con una aplicación desde el dispositivos móviles. El drone crea una red WiFi, a la cuál conectas tu teléfono o tablet iOS ó Android y desde la aplicación, suministrada también por ña marca Parrot, se tiene una comunicación directa con el drone.

A parte de esto Parrot tiene una SDK para desarrolladores la cuál es libre y está muy bien documentada para que quien quiera pueda desarrolar sus aplicaciones. Esta SDK es sobre la que JdeRobot ofrece, junto con el middleware ICE (*Internet Communication Engine*), una pasarela que nos da conectividad entre el drone real y la aplicación/componente que estemos desarrollando o usando.

Para simular este drone de la manera mas realista y realizar las pruebas de manera virtual, nos vamos a apoyar en el simulador robótico Gazebo.

Figura 2.1: ArDrone de Parrot

2.2. Simulador robótico Gazebo

Gazebo es un simulador 3D de robótica desarrollado por la *Open Source Robotics Foundation (OSRF)*. Es multiplataforma y entre otras cosas nos permite diseñar y crear nuestro propio robot y escenarios realistas, con obstáculos y objetos, para probar nuestros algoritmos de una forma muy parecida a las condiciones que nos vamos a encontrar en el mundo real, pero sin poner en peligro ni nuestros robots ni a ninguna persona cercana en caso de que se comporte de una forma inesperada. Para conseguir este realismo y potencia Gazebo cuenta con un motor físico para la mecánica, iluminación, gravedad, inercia... Uno de los puntos más favorables de este simulador es que es libre y tiene una comunidad muy numerosa y participativa.

JdeRobot tiene desarrollado el componente para Gazebo que contiene el modelo, el mundo y los plugin necesarios para simular el ArDrone de Parrot, de tal manera que podremos testear el código para ir puliendo y conseguir que sea lo suficientemente estable para posteriormente poder probarlo en el drone real.

Se ha usado la versión 5.1 del simulador, la cuál es última version estable a la fecha de comienzo y 100

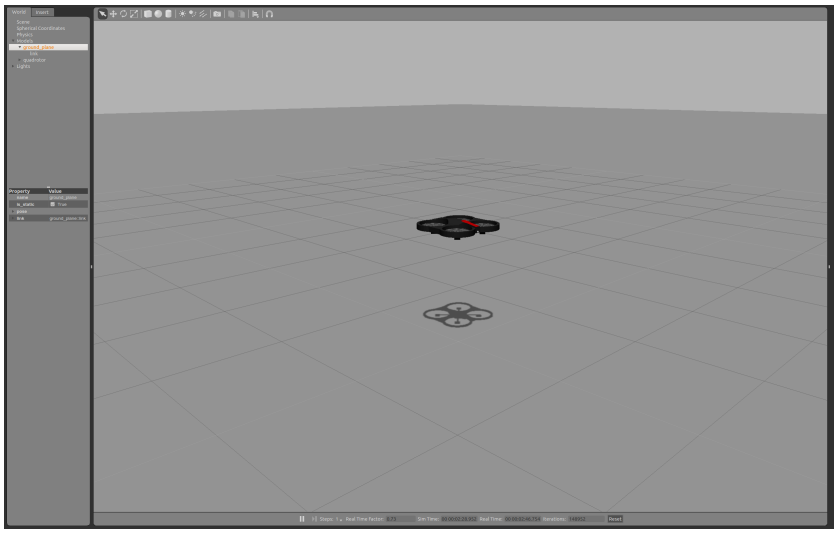


Figura 2.2: Simulador Gazebo

Este simulador junto con los plugins, mundos y mapas creados por JdeRobot es una poderosa heramienta que nos será muy útil para nuestro proyecto.

2.3. JdeRobot

JdeRobot es un proyecto desarrollado por el laboratorio de robotica de la Universidad Rey Juan Carlos de Madrid. Es un entorno para el desarrollo de aplicaciones relacionadas con la robotica, la visión artificial, automatización del hogar y escenarios con sensores y actuadores, y software inteligente. Está escrito mayormente en C++ y basado en un entorno de componentes distribuidos. Los componentes, que pueden correr de manera simultánea de manera asíncrona están conectados a través del *middleware* ICE, el cuál veremos a continuación. Los componentes pueden ser escritos en C++, Java, Python... y todos ellos interoperar a través de interfaces ICE.

JdeRobot simplifica el acceso a unidades hardware como cámaras. Obtener datos de un sensor es tan sencillo como llamar a una función local, y mandar comandos a un motor o actuar se hace también simplemente llamando a una función local. Estos sensores, actuadores o unidades hardware pueden ser simuladas o reales y estar en local o en una red de datos como

internet.

Incluye numerosas librerías y herramientas que podemos usar para desarrollar nuestra propia aplicación o para controlar nuestro propio robot. JdeRobot es software libre, bajo licencia GPL y LGPL. También utiliza software de terceros como el simulador Gazebo, ArDroneServer ROS, GTK, OpenCV...

La versión utilizada en el proyecto ha sido la 5.3.1, última versión estable en la fecha del comienzo del mismo. Dentro de las herramientas utilizadas en el proyecto hemos hecho uso básicamente de dos:

- *Plugin de Gazebo*: Desarrollado por JdeRobot para añadir el ArDrone a Gazebo con las mismas funcionalidades al drone real. Este plugin nos permite abstraernos de las conexiones a mas bajo nivel, como la lectura de sensores o en envío de comandos a los motores, simplificando el acceso a los mismo y obteniendo los datos con una simple llamada a función.
- *ArDroneServer*: Este componente es el encargado de comunicarse directamente con el ArDrone real, ejecutar las funciones y comandos de bajo nivel para obtener los datos y valores de los sensores, y ejecutar las funciones neccesarias sobre los motores y actuadores según las órdenes de movimiento dadas.

2.4. ICE

ICE o *Internet Communication Engine* es un entorno RPC orientado a objetos desarrollado por Zeroc con soporte para lenguajes como C++, C#, Java, JavaScript, y Python entre otros, y SO's como Linux, Mac OS X y Windows, que nos permite crear conexiones e interacciones de red entre máquinas con servidores corriendo en diferentes lenguajes y/o SO's. Estas conexiones pueden ser síncronas o asíncronas, usando variedad de protocolos de red como TCP, UDP ó SSL/TTS

ICEJS, o ICE for JavaScript es un *plugin* que se le puede añadir a ICE el cual añade la funcionalidad de conexiones mediante *websockets* lo que nos permite conectar navegadores a través de JavaScript a los protocolos mas comunes de ICE.

En versiones más actuales ICEJS viene incorporado en la suite principal de ICE, pero nosotros hemos usado la versión 3.5 que es la que soporta la suite de JdeRobot con la que trabajamos, por lo que debemos instalar el plugin una vez instalado ICE.

Figura 2.3: Esquema de ICE JS

2.5. WebRTC

Web Real-Time Communication (WebRTC) es un proyecto de software libre y gratuito que nos permite tener en el navegador tecnología en tiempo real ('Real-Time Communication' ó RTC), sin plugins, a través de varias APIs de JavaScript. Facilita las llamadas de voz, videollamadas, chat y compartición de archivos y datos. WebRTC es una tecnología entre pares, por lo que nos permite desarrollar estas aplicaciones para que funcionen directamente desde un navegador a otro *sin pasar por servidor intermedio.*

En todo el documento nos referimos a una llamada WebRTC entre dos navegadores, lo cual es su principal propósito, pero también está diseñado para que pueda ser integrado con otros sistemas de comunicación como voz sobre IP (VOIP), clientes SIP, e incluso sobre la red telefónica pública conmutada (PSTN).

Enviar audio y/o vídeo con calidad, e intercambiar cualquier tipo de datos requiere de muchas funcionalidades complejas en el navegador. Para no preocuparnos de estas dificultades las APIs de WebRTC proporcionan todo el conjunto completo de funciones para manejar y crear nuestras aplicaciones, como el control y administración de la conexión, codificación/decodificación del audio/vídeo, negociación entre navegadores, control de la conexión, atravesar cortafuegos y NAT.

Con unas docenas de líneas de JavaScript podemos tener una videoconferencia entre pares con intercambio de archivos o datos en tiempo real. Ese

es el potencial que WebRTC tiene. Pero aún así hay una serie de escollos como la señalización, descubrimiento de pares, negociación de la conexión o seguridad que debemos controlar para conseguir una llamada exitosa.

Que WebRTC no necesita un servidor no es del todo cierto, ya que sí que necesita de lo que llamamos Servidor de Señalización. Éste es el encargado de establecer el primer contacto entre ellos, facilitando el intercambio de paquetes de la negociación WebRTC.

WebRTC está compuesto de 3 API's:

- *getUserMedia*: adquisición del flujo local de audio y vídeo.
- *RTCPeerConnection*: comunicación de audio y vídeo.
- *RTCDataChannel*: comunicación de cualquier otro tipo de datos.

En este momento WebRTC es accesible para todos los usuarios a través de navegadores como Chrome o Firefox. Sin embargo, WebRTC está aún en construcción, tanto la forma de implementar las API's que tiene cada navegador como la propia norma, con sus protocolos de funcionamiento. Como resultado todo lo que exponga sobre estas API's se refiere a la situación actual y puede cambiar en el futuro.

A continuación vamos a desgranar y explicar cómo funciona el intercambio de paquetes de señalización así como cada una de las API's que componen WebRTC.

2.6. WebRTC: Señalización

Señalización es el proceso de intercambio de datos y metadatos necesarios para coordinar una llamada entre navegadores con WebRTC. Para realizar esta labor WebRTC necesita de la ayuda de un servidor externo ya que la norma deja el campo de la señalización a la capa de la aplicación.

Los objetivos básicos de la señalización son dos, el intercambios de los datos necesarios para establecer un flujo audiovisual y el intercambio de direcciones de red para que los pares sean alcanzables. Entre las labores para satisfacer estos objetivos se encuentran la detección de los pares, el intercambio de paquetes de control de la sesión como los candidatos *ICE (Internet*

Communication Engine) y los *SDP (Session Description Protocol)*, las prestaciones que puede darnos cada par así como cualquier otro dato o paquete necesario para realizar este 'apretón de manos' inicial.

WebRTC no especifica qué tipo de servidor hemos de usar para estas funciones. Esto es debido a que diferentes aplicaciones pueden preferir distintos servidores básicos o personalizados según sus necesidades. La única restricción es el uso de la arquitectura JSEP, la cuál especifica cómo debe ser la secuencia de señalización para tener una llamada exitosa.

El servidor debe usar la arquitectura *JSEP (JavaScript Session Establishment Protocol)*. Esta arquitectura elimina al navegador de casi todo el flujo de señalización, el cual se maneja desde JavaScript haciendo uso de dos interfaces: transfiriendo los SDP local/remoto e interactuando con la máquina de estados ICE. Esta arquitectura nos evita, entre otras cosas, que el navegador tenga que guardar estados de sesión, de tal manera que se pueden guardar en el servidor y evitar problemas si la página se recarga, por ejemplo.

JSEP no establece un modelo particular de señalización más allá de usar uno capaz de realizar el intercambio de los SDP y ICE según la norma RFC3264 de *oferta/respuesta*, [(figura 2.4)] de tal manera que ambas partes de la llamada sepan cómo actuar en cada momento. JSEP nos da los mecanismos necesarios para crear estas ofertas, así como aplicarlas a las sesión.

Figura 2.4: Oferta / Respuesta a través del servidor de señalización

El orden en que se llaman a estos mecanismos o funciones de la API es importante, por lo que la aplicación deberá saber el orden en el que tiene que llamar a cada una, convertir las ofertas en mensajes que entienda el protocolo de señalización elegido y hacer la conversión inversa con los mensajes que se reciben para obtener ofertas que entiendan as API's.

2.6.1. Estableciendo el Flujo Audiovisual: Descriptores de Sesión y Máquina de Estados

El manejo de las descripciones de sesión es simple y sencillo. Siempre que el intercambio de una oferta/respuesta es necesario, el par que establece la llamada ó *llamante (callee)* crea la oferta llamando a la función *createOffer()* de la API. Esta oferta puede ser modificada por la aplicación si así fuese necesario y se establece como configuración local en ese par con *setLocalDescription()* y se envía al par remoto a través del servidor de señalización utilizado. Al recibir esta oferta el par *llamado (called)* lo utiliza como configuración del otro par con *setRemoteDescription()* y utiliza *createAnswer()* para crear una respuesta apropiada, la cual establece como configuración local (*setLocalDescription()*) y envía la respuesta de vuelta a través del servidor de señalización. El par llamado al recibir la respuesta llama también a *setRemoteDescription()*, y de esta manera ambos lados tienen la información del descriptor de sesión propio y el del par remoto.

Para establecer un intercambio de flujo audiovisual, el *agente de usuario (user agent)* del navegador necesita parámetros específicos para indicar al par remoto qué es lo que va a transmitir, de la misma manera que necesita conocer los parámetros del flujo audiovisual que va a recibir para saber cómo decodificarlo y manejarlo. Estos datos se determinan en la descripción de sesión (SDP), los cuales se intercambian en ofertas/respuestas usando las API's JSEP como ya hemos visto anteriormente.

Si el SDP pertenece a la parte local o remota tiene su importancia. Una vez realizado el intercambio, cada parte mirará la lista de codecs soportados por él mismo y por la otra parte, y el cruce de los resultados determinará qué codecs debe usar para enviar y cuál para digerir lo recibido. Los parámetros exactos de la transmisión sólo se pueden saber una vez la oferta y la respuesta han sido intercambiados. Sin embargo, hay ocasiones en las que el llamante o el que hace la oferta puede recibir flujo audiovisual después de enviar la oferta pero antes de recibir la respuesta proveniente del otro par. Para procesar este flujo audiovisual de manera adecuada, el manejador del llamante debe conocer los detalles de la oferta antes de que la respuesta llegue.

Por lo tanto, para manejar los descriptores de sesión de manera correcta, los agentes de usuario necesitan:

- Conocer si el descriptor de sesión pertenece a la parte local o remota.
- Conocer si el descriptor de sesión es una oferta o una respuesta.

- Permitir a la oferta ser especificada independientemente de la respuesta.

Para satisfacer estas premisas JSEP aborda esto añadiendo los métodos *setLocalDescription()* y *setRemoteDescription()* y teniendo un campo en los descriptores de sesión indicando el tipo de sesión que se suministra. En la figura 2.5 podemos ver el esquema de intercambio de paquetes SDP y el ajuste de los mismos en cada par segun corresponde.

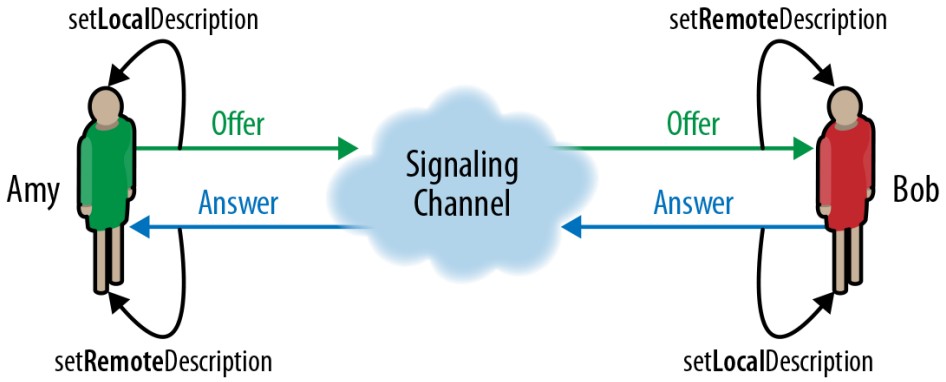


Figura 2.5: Intercambio y establecimiento de los SDP en cada par.

JSEP también permite el uso de respuestas provisionales. Estas respuestas permiten al par remoto o llamado comunicar e informar de los parámetros iniciales de la sesión al llamante, de tal manera que la sesión puede comenzar mientras se espera una respuesta final posteriormente. Este concepto es importante en el modelo oferta/respuesta, ya que al recibir una de estas respuestas el llamante puede liberar y usar más recursos como extra *candidatos ICE, candidatos TURN* o vídeo decodecs. Estas respuestas provisionales no provocan ningún tipo de des-asignación o problema, por lo que pueden ser recibidas a lo largo de la llamada para estabilizar o mejorar la misma según varíen las condiciones del ancho de banda de uno de los pares, por ejemplo.

El cometido principal del intercambio de SDP es la negociacion de video. Es un proceso a través del que cada par puede indicar al otro qué resoluciones y *frames rate* de vídeo es capaz de recibir. Esto lo hace a través del atributo *'a=imageattr'* en el SDP. Cada par puede tener límites como la capacidad de proceso que el decoder tiene, o simplemente restricciones de la aplicación.

Formato de los Descriptores de Sesión

En la especificación WebRTC, los descriptores de sesión o *session descriptions* están formados por mensajes *SDP (Session Description Protocol)*. Este formato no es el más óptimo para manipular con JavaScript, pero es el más popular y aceptado en el campo de las comunicaciones audiovisuales en tiempo real. Este formato es el que usa JSEP para formar e intercambiar los descriptores de sesión.

Para facilitar el procesado en JavaScript y una futura flexibilidad, los SDP los genera la API como un objeto o *blob*. Si en un futuro WebRTC soporta algún formato nuevo para los descriptores de sesión, estos serán fácilmente añadidos y habilitados para poder usarlos en nuestras aplicación en vez de SDP.

La forma que tiene un paquete SDP es la siguiente:

```
1  v=0
2  o=- 7729291447651054566 1 IN IP4 0.0.0.0
3  s=-
4  t=0 0
5  a=group:BUNDLE a1 d1
6  a=ice-options:trickle
7  m=audio 9 UDP/TLS/RTP/SAVPF 96 0 8 97 98
8  c=IN IP4 0.0.0.0
9  a=rtcp:9 IN IP4 0.0.0.0
10 a=mid:a1
11 a=msid:QI39StLS8W7ZbQl1sJsWUXkr3Zf12fJUvzQ1
12        QI39StLS8W7ZbQl1sJsWUXkr3Zf12fJUvzQ1a0
13 a=sendrecv
14 a=rtpmap:96 opus/48000/2
15 a=rtpmap:0 PCMU/8000
16 a=rtpmap:8 PCMA/8000
17 a=rtpmap:97 telephone-event/8000
18 a=rtpmap:98 telephone-event/48000
19 a=maxptime:120
20 a=ice-ufrag:7sFvz2gdLkEwjZEr
21 a=ice-pwd:dOTZKZNVlO9RSGsEGM63JXT2
22 a=fingerprint:sha-256 6B:8B:F0:65:5F:78:E2:51:3B:AC:6F:F3:3F
       :46:1B:35
23                      :DC:B8:5F:64:1A:24:C2:43:F0:A1:58:D0:A1
                           :2C:19:08
24 a=setup:active
25 a=rtcp-mux
26 a=rtcp-rsize
27 a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
28 a=extmap:2 urn:ietf:params:rtp-hdrext:sdes:mid
```

```
29 a=ssrc:4429951804 cname:Q/NWs1ao1HmN4Xa5
30
31 m=application 9 UDP/DTLS/SCTP webrtc-datachannel
32 c=IN IP4 0.0.0.0
33 a=mid:d1
34 a=fmtp:webrtc-datachannel max-message-size=65536
35 a=sctp-port 5000
36 a=fingerprint:sha-256 6B:8B:F0:65:5F:78:E2:51:3B:AC:6F:F3:3F
      :46:1B:35
37                        :DC:B8:5F:64:1A:24:C2:43:F0:A1:58:D0:A1
                             :2C:19:08
38 a=setup:active
```

Listing 2.1: Ejemplo paquete SDP

2.6.2. Intercambio de los Datos de Red: Interactive Connectivity Establishment (ICE)

Al igual que los pares tienen que intercambiar información sobre el media, también necesitan hacerlo sobre la información de *red* para que los pares sean visibles entre ellos y puedan alcanzarse. ICE es una técnica usada en aplicaciones de voz, vídeo, *peer-2-peer*, entre otros que nos permite solucionar problemas de alcance de red entre dos ordenadores. Estos problemas son debidos a que los ordenadores suelen estar dentro de una red privada y/o cortafuegos. Esta técnica nos permite descubrir suficiente información sobre la topología de los otros pares para encontrar una o varias rutas potenciales entre ellos.

Esta información ha de obtenerse de manera local en cada par con el *Agente ICE* asociado a cada objeto RTCPeerConnection. El Agente ICE es responsable de:

- Reunir tuplas candidatas de IP + Puerto.
- Realizar pruebas de conectividad entre los pares.
- Enviar *keepalives*.

Una vez se ha finalizado y configurado el proceso de descripción de sesión, el Agente ICE local comienza automáticamente el proceso de descubrir todos los posibles candidatos en el par local. Cada candidato posible se le llama *Candidato ICE*:

1. El Agente ICE pide al sistema operativo las direcciones IP locales.

2. Consulta a un servidor *STUN (Session Traversal Utilities for NAT)* externo la tupla de dirección IP pública y puerto del par.

3. Consulta a un servidor *TURN (Traversal Using Relays around NAT)* como último recurso.

Como podemos ver, ICE necesita de servidores externos para obtener la tupla de dirección IP y puerto públicos necesarios para el otro par si esta fuera de la misma red local. STUN es un protocolo estandarizado para descubrir direcciones IP publicas de equipos que están detrás de un NAT. TURN es un servidor para transmitir mensajes entre dos clientes. Este servidor sólo se usará si falla la conexión entre pares después de probar con las direcciones IP locales y las públicas obtenidas en el servidor STUN. No es obligatorio configurar estos servidores. Si la conexión entre los pares es en la misma red no necesitamos configurar servidores STUN/TURN ya que con las direcciones locales es suficiente.

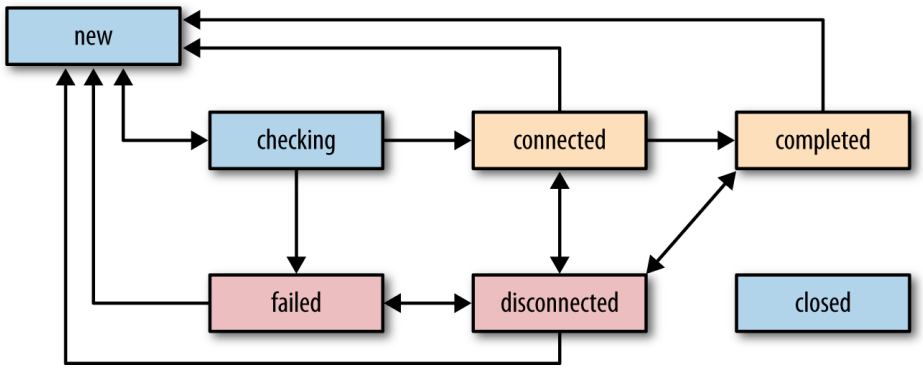


Figura 2.6: Maquina de estados y transiciones ICE.

Cuando un Candidato ICE es descubierto, se envía al par remoto y una vez allí, se añade en RTCPeerConnection la información de sesión que contiene ese paquete con *setRemoteDescription()*, de tal manera que el Agente ICE puede empezar a hacer pruebas de conectividad para ver si puede alcanzar al otro par.

Una vez los dos Agentes ICE tienen una lista completa de los Candidatos ICE de ambos pares, cada agente comprueba pareando ambas lista cuales funcionan. Para ello tienen una planificación de prioridades: primero direcciones IP locales, luego IP públicas y finalmente si ambas fallan servidor TURN. Cada comprobación es una petición/respuesta STUN que el cliente

realiza con un particular candidato enviando una petición STUN desde el candidato local al candidato remoto.

Si uno de los pares de candidatos funciona, entonces tenemos una ruta de conexión entre ambos pares. Si todos los candidatos fallan ambas conexiones RTCPeerConnection se marca como fallida o la conexión se hace a través de un servidor TURN.

Cuando una conexión se ha establecido correctamente cada Agente ICE continua haciendo peticiones STUN periódicas al otro par, lo cual sirve también como *keepalives*.

Goteo de Candidatos ICE *(ICE Candidate Trickling)*

Recopilar IP's locales es rápido, pero conseguir las IP's publicas con sus puertos a través de servidores STUN requiere de un intercambio de paquetes entre el par y el servidor STUN y por consiguiente más tiempo.

ICE Candidate Tricking es una extensión del protocolo ICE por la cuál el llamante puede incrementar el número de candidatos para el llamado después de la primera oferta. Este proceso permite al llamado comenzar a establecer las conexiones ICE, sin tener que esperar a que el llamante recopile todos los posibles candidatos. Con esta técnica conseguimos un establecimiento del flujo audiovisual más rápido.

Esta técnica es opcional aunque es la recomendada. Las aplicaciones que lo soportan pueden enviar directamente la oferta SDP inicial sin candidatos ICE inmediatamente, y enviar candidatos individuales cuando los vayan descubriendo; las aplicaciones que no lo soportan simplemente esperan la indicación de que la recopilación de candidatos está completa, crean la oferta con todos los candidatos y enviarla.

1. Intercambio ofertas SDP sin Candidatos ICE.
2. Cuando se descubre un candidato se envía directamente a través del servidor de señalización.
3. La comprobación de los Candidatos ICE se realiza en el momento de recibir uno.

Formato de un Candidato ICE

```
1 candidate:1 1 UDP 1694498815 192.0.2.33 10000 typ host
```

Listing 2.2: Ejemplo paquete SDP

Fundación (1): Identificador para cada candidato del mismo tipo, misma interfaz y servidor STUN.
ID (1): Identificador. 1 para RTP, 2 para RTCP.
Protocolo (UDP): Protocolo de transporte del candidato.
Prioridad (1694498815): Prioridad del componente dado.
Dirección IP y puerto (192.0.2.33 10000): Dirección IP y puerto del candidato.
Tipo (typ): Tipo del componente.
Dirección relacionada (host): Información opcional que contiene dirección IP y puerto privado.

2.7. WebRTC: API getUserMedia

getUserMedia es la API encargada de suministrarnos el flujo de audio y/o vídeo. Pide permiso al usuario para acceder y utilizar los dispositivos hardware como la cámara y el micrófono. Por el momento sólo está disponible para captar el hardware de audio y vídeo anteriormente mencionado, pero se pretende mejorar y ampliar la API para que en un futuro se pueda hacer *streaming* de casi cualquier fuente de datos, como un disco duro o sensores conectados al ordenador.

Para tener una rica videoconferencia no es suficiente con obtener los flujos en crudo *ó formato raw* de la cámara o el micrófono. Cada flujo debe ser procesado para aumentar la calidad, sincronizarlos y ajustar el caudal *(bitrate)* de salida según las fluctuaciones del ancho de banda y la latencia entre pares. A la hora de recibir el flujo nos encontramos en la misma situación pero a la inversa. WebRTC nos da unos motores de procesado de audio y vídeo (figura 2.7) que harán todas estas cosas por nosotros.

getUserMedia es la API que suministra las funciones y los motores necesarios para poder cumplir con las especificaciones anteriormente mencionadas, así como manipular o procesar los flujos obtenidos. El objeto *MediaStream* (figura 2.8) es la forma en la que nos suministra los flujos esta API.

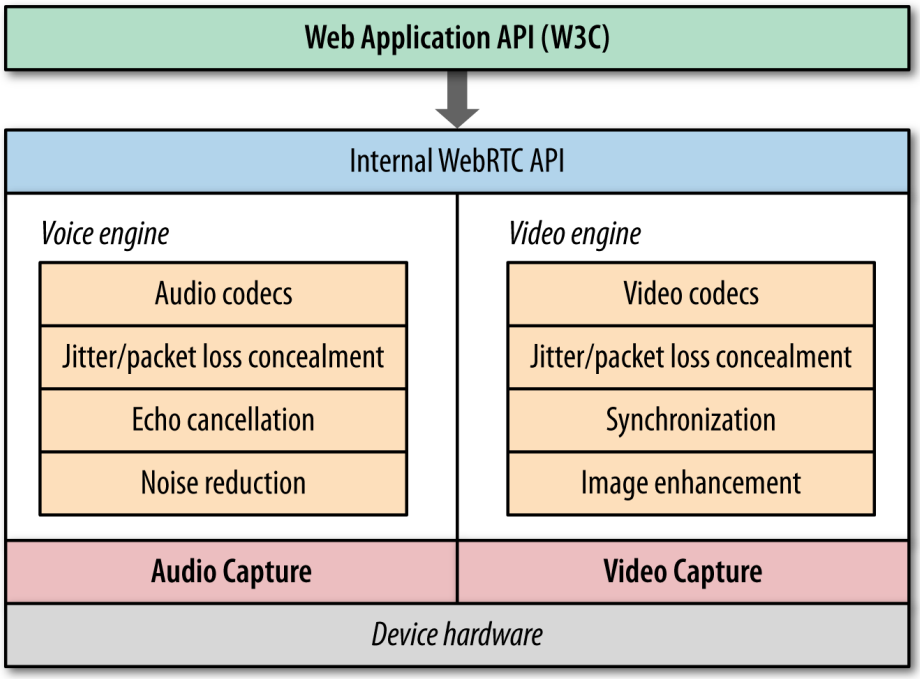


Figura 2.7: Motores de audio y vídeo de WebRTC

- El objeto MediaStream consiste en una o varias pistas o *tracks* (*MediaStreamTrack*).
- Las pistas que componen el MediaStream están sincronizadas una con la otra.
- La salida del MediaStream puede ser enviada a uno o varios destinatarios, como fuente de vídeo local, un par remoto o procesarlo con funcionalidades que nos proporciona, por ejemplo, HTML5.

Todo el procesado de audio y vídeo, como la cancelación de ruido, ecualización, mejora de la imagen y todas las demás son automáticamente manejadas por los motores de audio y vídeo.

Sin embargo, las características del flujo audivisual son restringidas por las capacidades de los dispositivos de entrada: audio mono o stereo, diferentes resoluciones de vídeo según la cámara, etc. Cuando hacemos una petición de media al navegador, *getUserMedia* nos permite indicar una lista de restricciones obligadas y opcionales.

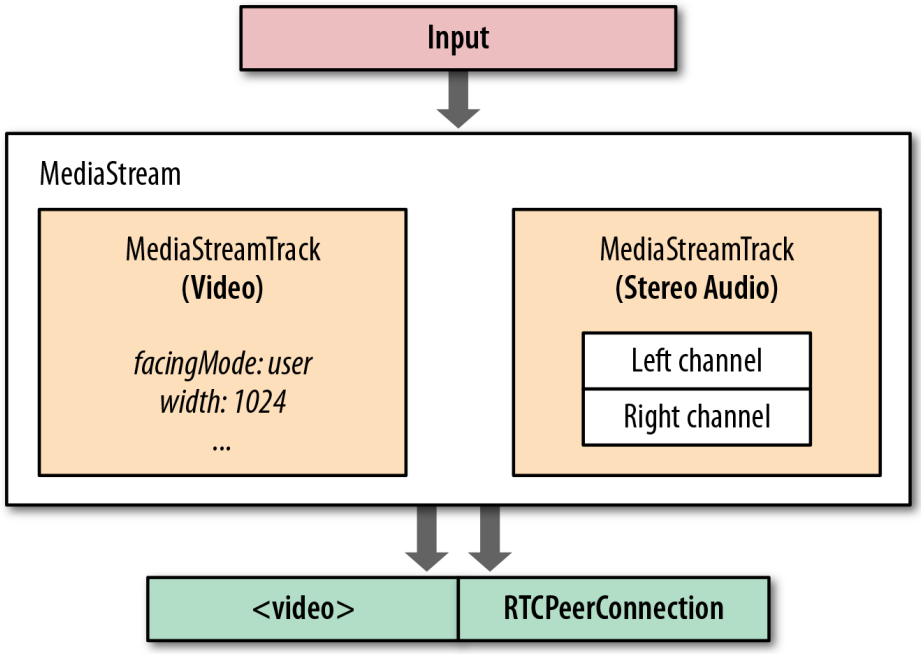


Figura 2.8: MediaStream

```
var constraints = {
    audio: false,
    video: {
        width: { min: 1024, ideal: 1280, max: 1920 },
        height: { min: 576, ideal: 720, max: 1080 },
    }
};

navigator.getUserMedia(constraints, handleUserMedia,
   handleUserMediaError);

function handleUserMedia(stream){
    var video = document.querySelector('video');
    video.src = window.URL.createObjectURL(stream);
}

function handleUserMediaError(error){
  console.log('getUserMedia error: ', error);
}
```

Listing 2.3: Llamada a función RTCPeerConnection

2.8. WebRTC: API RTCPeerConnection

Esta API es la encargada de crear la conexión *Peer-2-Peer* entre el navegador local y el remoto. Trabaja de manera diferente si es el que hace la llamada (*caller*) y el que la recibe (*called*) debido a la señalización oferta/respuesta que ya hemos visto. Para ello hace uso de sus funciones para completar el proceso de señalización descrito anteriormente.

Es también la encargada de manejar la conexión una vez establecida. Entre las funciones automáticas que realiza se encuentran:

1. Ocultamiento de paquetes perdidos.
2. Cancelación de eco.
3. Adaptación del ancho de banda.
4. Buffer dinámico en función del tembleque (*jitter*) o retardo(*delay*).
5. Control automático de ganancia.
6. Reducción o eliminación de ruido.

Es el desarrollador de la aplicación el encargado de llamar a las funciones que componen esta API en el orden, tiempo y forma correcta para cumplir con la arquitectura JSEP y conseguir un intercambio de descriptores de sesión y de Candidatos ICE exitoso.

La función admite dos argumentos opcionales:

```
1 var PC = new RTCPeerConnection(ICEconfig, pcConstraints);
```

Listing 2.4: Llamada a función RTCPeerConnection

ICEConfig es una variable la cuál contiene los datos necesarios para conectarse con el servidor STUN y TURN y poder hacer NAT Trasversal. *pcConstraints* también es una variable a la que se le pueden añadir una serie de restricciones como *RtpDataChannels*, la cuál estará obsoleta en futuras versiones y se dejará de usar. Esta variable está para futuras mejoras.

Protocolos de transporte en tiempo real

El cerebro humano es muy bueno 'rellenando huecos' pero altamente sensible a los retardos. Si perdemos unas muestras de audio o vídeo no nos

afecta demasiado en la percepción de lo que estamos recibiendo, pero en cambio añade un retardo al audio con respecto al vídeo y hará que ese material nos sea hasta molesto.

Por este motivo las aplicaciones de audio y vídeo en tiempo real están diseñadas para tolerar pérdidas intermitentes de paquetes. Los codecs pueden rellenar estos pequeños espacios que dejan los paquetes perdidos, muchas veces incluso con muy poco impacto con respecto a la imagen real. Una baja latencia y el vicacidad son mucho mas importantes que la fiabilidad (*reliability*).

Este requerimiento de vicacidad antes que fiabilidad es la primera razón por la que el protocolo UDP es elegido para el envío de datos en tiempo real. TCP es fiable y tiene entrega ordenada de paquetes. Si uno de ellos se pierde, entonces TCP almacena los paquetes siguientes y para la retransmisión hasta que el paquete perdido es reenviado y recibido. En cambio UDP no garantiza la entrega de paquetes, el orden de entrega, la ruta de los paquetes ni control de congestión de red.

WebRTC usa UDP como protocolo de transporte. Dadas las características de UDP, ¿podemos simplemente enviar cada paquete según llega y olvidarnos? no, ya que también necesitamos mecanismos para atravesar NAT's y cortafuegos, negociar los parámetros de cada flujo, encriptar los datos de usuario, congestión de red... Para abastecer estas necesidades WebRTC tiene una lista de protocolos y servicios que trabajan por encima de UDP [(figura 2.9)].

ICE, STUN y TURN son necesarios para establecer y mantener la conexión *Peer-2-Peer* sobre UDP, como ya hemos visto en el apartado de señalización.

Entrega de vídeo con SRTP

WebRTC nos permite adquirir el vídeo/audio y enviarlo para la visualización en el otro par. También nos permite elegir las características con las que queremos adquirir ese vídeo/audio, pero a partir de ahí es WebRTC y el motor de red el que se encarga del resto. Optimización en la codificación, tratar con paquetes perdidos, tembleque, etc, son algunas de las cosas con las que tiene que tratar WebRTC. Por este motivo WebRTC no garantiza la entrega en el otro par del vídeo a su máxima resolución.

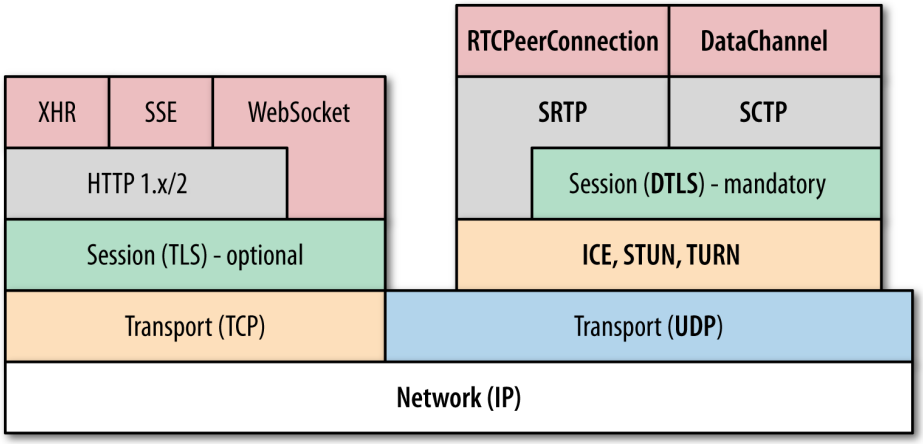


Figura 2.9: Pila de protocolos WebRTC

El motor de red tiene su propio flujo de datos sin tener en cuenta desde el comienzo la capacidad de la red ni el caudal del flujo. Primero empieza enviando el vídeo y el audio a un bitrate bajo (por debajo de 500kbps) y luego comienza a ajustar la calidad del flujo según la capacidad del ancho de banda. Según pueda variar el ancho de banda de la conexión de red así actúa el motor de red para ajustarlo.

SRTP define un formato de paquete estándar para enviar audio y vídeo a través de IP, pero por si mismo no proporciona ningún mecanismo o garantías de entrega en orden, fiabilidad en la entrega o corrección de errores. Simplemente encapsula el vídeo y el audio con metadata adicional. Entre esta metadata cada paquete SRTP tiene un numero de secuencia incremental, una marca de tiempo y un identificador SSRC, lo que permite al par que recibe el flujo detectar si los paquetes llegan desordenados, sincronizar los diferentes flujos y asociar cada paquete al flujo correspondiente.

SRTP usa un protocolo que lo complementa. Este es SRTCP, el cual es el protocolo que controla el número de paquetes y bytes perdidos, último número de secuencia recibido, *jitter* de los paquetes SRTP recibidos y otras estadísticas. Periódicamente estas estadísticas se intercambian entre los pares y la usan para ajustar la tasa de envío, calidad de codificación y otros parámetros.

Ambos protocolos corren directamente sobre UDP y trabajan conjuntamente para adaptar y optimizar la conexión.

2.9. WebRTC: API RTCDataChannel

Llegamos a la tercera API de WebRTC. RTCDataChannel nos da la posibilidad de transferir todo tipo de datos u objetos a través de la conexión entre pares establecida con RTCPeerConnection.

Esta conexión de datos es *full duplex* y nos permite el intercambio de datos, intercambio de archivos, sincronización de juegos online, etc, todo ello con un retardo mínimo. Las posibilidades son infinitas y se puede adaptar a cualquier necesidad que se tenga para nuestra aplicación.

Capacidades

RTCDataChannel soporta un juego muy flexible de tipos de datos. Soporta *strings*, binarios de JavaScript, *Blobs*, *ArrayBuffer* y *ArrayBufferView*. Según nuestras necesidades nos puede resultar más útil un tipo de datos u otro.

Como protocolo de transporte soporta TCP, UDP y SCTP. Esto nos permite configurar la conexión de datos de manera fiable (*reliable*) o no fiable (*unreliable*). La primera de ellas nos garantiza la entrega de todos los mensajes que enviemos y que los mismos lleguen en el mismo orden que los hemos enviado. Esto provoca una sobrecarga que puede provocar un funcionamiento más lento además de tener un mayola sobrecarga permitiéndonos tener una conexión más rápida. SCTP es un protocolo de transporte similar a TCP y UDP que puede funcionar directamente en la cima del protocolo IP. Sin embargo, en WebRTC, SCTP es construido sobre un túnel DTLS, el cuál corre encima de UDP.

	TCP	**UDP**	**SCTP**
Fiabilidad	Fiable	No fiable	Configurable
Entrega	Ordenada	No ordenada	Configurable
Transmisión	Orientada al byte	Orientada al mensaje	Orientada al mensaje
Control de flujo	Si	No	Si
Control de congestión	Si	No	Si

Figura 2.10: Capacidades de RTCDataChannel.

Como las API's anteriores llamamos a RTCDataChannel con una variable

con opciones de configuración:

- *Ordered*: Booleano para indicar si queremos que nos garantice la entrega ordenada de paquetes.
- *maxRetransmitTime*: Tiempo máximo para intentar retransmitir cada paquete si la entrega falla. (Fuerza el modo no fiable).
- *maxRetransmits*: Número máximo de veces que queremos que reenvíe cada paquete si la entrega falla. No puede usarse junto con *maxRetransmitTime*. (Fuerza el modo no fiable).
- *Protocol*: Permite el uso de un subprotocolo pero tiene que ser soportado por TCP/UDP.
- *Negociated*: Si se configura a verdadero, elimina la configuración automática del *datachannel* en el otro par. Se da por hecho que tienes previsto crear el canal de otra manera con el mismo ID.
- *Id*: Permite dar tu propio ID al canal.

Seguridad

La encriptación es una obligación para todos los componentes WebRTC. Tanto audio, vídeo, data e información de la aplicación debe estar encriptado cuando se transmite.. En *RTCDatachannel* todos los datos son codificados con *Datagram Transport Layer Security (DTLS)*. DTLS es un derivado de SSL por lo que los datos que intercambies irán igual de seguro que si usásemos SSL. Es obligatorio, para que un navegador pueda usar WebRTC, que tenga implementada esta tecnología.

Entrega de datos con SCTP

Como ya hemos visto en la figura 2.9, *RTCDataChannel* trabaja con un protocolo llamado *Stream Control Transmission Protocol (SCTP)*, el cual corre sobre DTLS, y este a su vez corre sobre UDP. Recalco esto ya que a diferencia del audio y el vídeo, para enviar data de la aplicación sí que necesitamos que lleguen todos los paquetes, por lo que si alguno se ha perdido hay que reenviarlo.

WebRTC requiere de 4 características que debe cumplir el protocolo:

1. El protocolo de transporte debe permitir tener varios canales independientes multiplexados.

   a) Cada canal debe permitir entrega ordenada y desordenada.

   b) Cada canal debe tener entrega fiable.

   c) Cada canal debe tener niveles de prioridad definidos por la aplicación.

2. El protocolo debe proveer 'orientación al mensaje', por lo que debe tener fragmentación y reagrupacion de los datos.

3. El protocolo debe tener mecanismos control del flujo y de la congestión.

4. El protocolo debe tener seguridad y confidencialidad en los datos que se envían.

La última característica se cumple ya que SCTP corre sobre el túnel DTLS, por lo que los datos que enviemos van encriptados y seguros hasta nuestro destinatario. Por otro lado SCTP, como vemos en la tabla 2.10 permite configurar la fiabilidad y la entrega ordenada de paquetes. SCTP también trocea los datos que queremos enviar y los encapsula en paquetes SCTP de 224 bits.

Para el control de flujo y de congestión SCTP tiene un saludo inicial similar al de TCP. Ambos usan la misma ventana inicial de congestión así como la misma lógica de crecimiento y decrecimiento para reducir la congestión una vez la comunicación está activa.