



Grado en Ingeniería en Sistemas Audiovisuales y Multimedia

Escuela Técnica Superior de Ingeniería de Telecomunicación

Curso académico 2015-2016

Trabajo Fin de Grado

Manejo de un drone con WebRTC y JdeRobot

Autor: Iván Rodríguez-Bobada Martín

Tutor: José María Cañas Plaza

Madrid 2015

*A mis padres y hermanos,
que siempre han estado a mi lado,
y siempre lo estarán.*

*Y a mis amigos, por ser
verdaderamente mis amigos.*

Gracias.

Índice general

1. Introducción	8
1.1. Robótica aérea	9
1.1.1. Clasificación y usos de los UAV en general	10
1.1.2. Cuadricópteros	11
1.1.3. Aplicaciones de los drones	12
1.1.4. Sistemas de control de drones	17
1.2. Tecnologías de comunicación en tiempo real	19
1.2.1. RTP	19
1.2.2. SIP	21
1.2.3. ORTC	21
1.2.4. WebRTC	22
1.3. Antecedentes	23
1.3.1. ArDroneServer	23
1.3.2. Surveillance 4.0	23
1.3.3. Surveillance 5.1	24
1.3.4. Teleoperadores y visores Web en JdeRobot	24
2. Objetivos	27
2.1. Objetivos	27
2.2. Metodología y plan de trabajo	28
3. Infraestructura Software	31
3.1. ArDrone de Parrot	31
3.2. Simulador robótico Gazebo	32
3.3. JdeRobot	34

<i>ÍNDICE GENERAL</i>	4
3.4. ICE	35
3.5. WebRTC	36
3.6. WebRTC: Señalización	37
3.6.1. Estableciendo el Flujo Audiovisual: Descriptores de Sesión . . .	38
3.6.2. Intercambio de los Datos de Red: Interactive Connectivity Es- tablishment (ICE)	41
3.7. WebRTC: API getUserMedia	44
3.8. WebRTC: API RTCPeerConnection	47
3.9. WebRTC: API RTCDataChannel	50
4. Desarrollo software	53
4.1. Diseño	53
4.2. Conexión local drone-navegador	56
4.2.1. ArDrone_server y WebSockets	56
4.2.2. getUserMedia	63
4.3. Conexión entre navegadores	65
4.3.1. Servidor de señalización	66
4.3.2. RTCPeerConnection: Transmisión de vídeo	67
4.3.3. RTCDataChannel	72
4.4. Interfaz web de usuario en cliente remoto	77
4.4.1. Visualización del vídeo	77
4.4.2. Joysticks y Gamepad	78
4.4.3. Relojes de navegación	83
4.4.4. Visualización espacial 3D del drone	85
5. Experimentos	89
5.1. Experimentos con simulador Gazebo	89
5.2. Vuelo del drone real	91
5.3. Vuelos con multidispositivos	95
6. Conclusiones	98
6.1. Conclusiones	98
6.2. Trabajos futuros	99

ÍNDICE GENERAL

5

Bibliografía

100

Índice de figuras

1.1. Distintas configuraciones de los motores del cuadricóptero para desplazarse.	12
1.2. Lily (a) y funcionamiento en 3 pasos (b).	13
1.3. Drone de Endesa realizando labores de revisión.	14
1.4. Uso de drones en agricultura de precisión.	15
1.5. Amazon Prime Air	15
1.6. Drone de Salvamento Marítimo	16
1.7. Uso de un drone para grabación.	17
1.8. Sistema WiFi de control del drone de Parrot.	19
1.9. Ejemplo de conexión RTC, RTCP y RTSP	20
1.10. Ejemplo de conexión SIP	22
1.11. Seguimiento horizontal usando ardrone_server.	24
1.12. interfaz (a) y arquitectura (b).	25
1.13. Cliente web UavViewer.js	26
2.1. Esquema general del desarrollo del proyecto.	29
3.1. ArDrone de Parrot	32
3.2. Simulador Gazebo	33
3.3. Estructura de ArDrone_Server	35
3.4. Esquema de ICE JS	35
3.5. Oferta / Respuesta a través del servidor de señalización	38
3.6. Intercambio y establecimiento de los SDP en cada par.	40
3.7. Maquina de estados y transiciones ICE.	43
3.8. Motores de audio y vídeo de WebRTC	45

3.9. MediaStream	46
3.10. Pila de protocolos WebRTC	49
3.11. Capacidades de RTCDataChannel.	51
4.1. Esquema general del proyecto.	53
4.2. Flujo de llamada del proyecto	55
4.3. Joysticks.	79
4.4. Relojes de navegación.	85
4.5. Mapa 3D con three.js	87
4.6. Control de batería y velocidad del cuadricoptero.	88
4.7. Interfaz web de usuario.	88
5.1. Experimento con el simulador Gazebo y un ordenador remoto.	89
5.2. Experimento con el simulador Gazebo.	90
5.3. Secuencia de movimiento del dron dentro de Gazebo.	91
5.4. Esquema del experimento con el dron real.	91
5.5. Experimento uno con dron real.	92
5.6. Secuencia de movimiento del experimento con el dron real.	92
5.7. Esquema de la configuración del experimento.	93
5.8. Configuración de los elementos a bordo del dron.	93
5.9. ArDrone no levanta el vuelo por exceso de peso.	94
5.10. Comprobación del funcionamiento de la aplicación.	94
5.11. Esquema del experimento con móvil como par remoto.	95
5.12. Experimento con móvil como par remoto.	96
5.13. Secuencia de movimiento del experimento con dispositivo móvil como par remoto.	96
5.14. Esquema del experimento con móvil como par local.	96
5.15. Experimento con móvil como par local.	97
5.16. Secuencia de movimiento del experimento con dispositivo móvil como par local.	97

Capítulo 1

Introducción

El drone es un vehículo aéreo no tripulado al que podemos ubicar dentro de la robótica aérea. El proyecto que explica esta memoria se encuadra dentro del manejo, control y recogida de datos de sensores y actuadores de un drone a distancia y su manejo en tiempo real desde un navegador web en un ordenador, tableta o teléfono.

En estos últimos años el desarrollo de vehículos aéreos no tripulados ha tenido un avance significativo, sobre todo en aplicaciones de uso civil. Uno de los campos en los que más ha despuntado es en el uso para la grabación de cualquier tipo de eventos, tanto a nivel profesional como a nivel *amateur*.

Las tecnologías web son otro campo que ha experimentado un avance enorme en los últimos años, permitiendo crear aplicaciones más complejas y elaboradas. A la vez de ser más potentes se pueden implementar directamente en un navegador, sin necesidad de ningún tipo de instalación en el ordenador del cliente.

El fondo del proyecto consiste aunar estos dos campos. Se pretende desarrollar una aplicación web con tecnologías de última generación que permita teleoperar el drone desde un ordenador o dispositivo móvil.

En las siguientes páginas se darán unas pinceladas sobre la robótica y su uso actual. También hablaremos sobre los sistemas actuales de control y manejo de drones. Para finalizar daremos una visión global sobre las tecnologías existentes dentro de las comunicaciones en tiempo real (RTC, *Real Time Communications*), que se pueden emplear para comunicar, por ejemplo, un drone con una estación en tierra.

1.1. Robótica aérea

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, entre otras: el álgebra, los autómatas programables, las máquinas de estados, la mecánica o la informática.

Es una rama que ha conseguido grandes avances no sólo en tareas que realizaban con anterioridad personas, sino además 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 más sencillo.

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

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

Dentro de la robótica, la rama de la robótica aérea lleva experimentando desde hace varios años un auge espectacular. A este tipo de robots se les conoce también como Vehículos Aéreo No Tripulados 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 común es que haya un piloto u operador teleoperándolo desde tierra.

1.1.1. Clasificación y usos de los UAV en general

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.** Fotografíar 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 vídeo 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 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.
- **Agricultura de precisión.** A través del uso de UAV conseguimos un estudio de las parcelas agrícolas más detallado y preciso, de manera que pueden aplicarse

tratamientos de manera localizada. Permite reducir costes, mejorar la rentabilidad de los cultivos y disminuye el impacto ambiental al poder aplicarse productos agroquímicos directamente y exclusivamente a las zonas afectadas.

1.1.2. Cuadricópteros

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 a este mismo problema hacer 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 dron 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 dron 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.

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.

La figura 1.4 muestra las configuraciones de las diferentes velocidades de los motores cuándo hay un giro hacia la derecha y cuándo el drone va hacia delante.

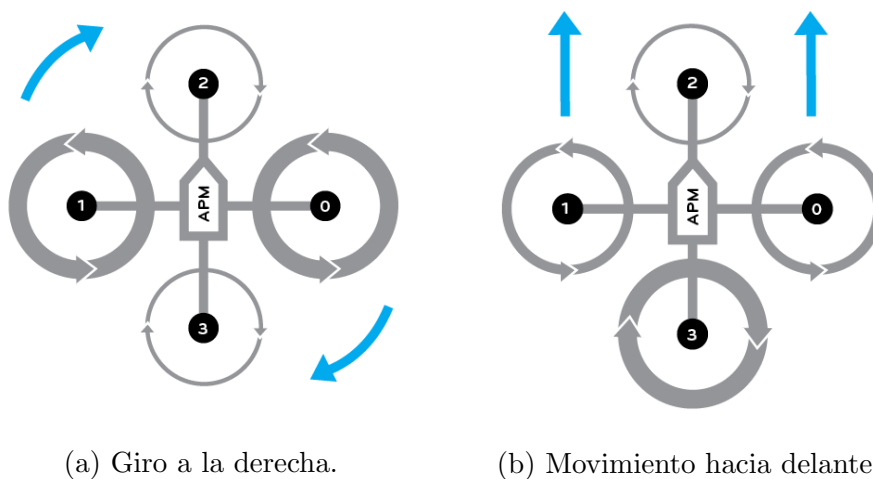


Figura 1.1: Distintas configuraciones de los motores del cuadricóptero para desplazarse.

1.1.3. Aplicaciones de los drones

Debido al auge de los cuadricópteros en los últimos tiempos se han desarrollado proyectos con ellos como elemento principal. Aquí resaltamos algunas de ellas que nos parecen interesantes y representativas.

Lily Drone

Lily¹ es un cuadricóptero de reducidas dimensiones y peso (1.3kg) el cuál va equipado con una cámara capaz de grabar vídeo hasta resolución de alta definición a 1080p y 120fps. Lo más llamativo de este drone no son sus capacidades de vuelo ya que sólo

¹<https://www.lily.camera>

nos permite vuelos de hasta 15 metros de altura y unos 40km/h de velocidad, si no que simplemente llevando un dispositivo de detección de reducidas dimensiones (28g), lo enciendes y él, sin necesidad de configuración, lo detecta y sigue todos sus movimientos mientras a la vez te está grabando. Los vuelos son de unos 20 minutos de duración, y el drone es resistente al agua, por lo que puedes usarlo prácticamente en cualquier tipo de situación. Uno de sus puntos fuertes es su funcionamiento en tres pasos, como muestra la figura 1.2 (b).



Figura 1.2: Lily (a) y funcionamiento en 3 pasos (b).

Revisión del tendido eléctrico con drones

La compañía eléctrica Endesa ha comprado una flota de catorce drones para la revisión de las líneas eléctricas en España. Los drones cuentan con cámara de alta resolución y termográfica. Con el uso de estos aparatos se agilizan las inspecciones, se mejora la calidad y ofrecen mayor seguridad al no tener que trabajar el operario sobre la red. Esto a su vez hace que no sea necesario cortar el suministro eléctrico. La figura

1.3 muestra un drone realizando servicios de revisión del tendido eléctrico.



Figura 1.3: Drone de Endesa realizando labores de revisión.

Hemav, Agricultura de precisión.

La empresa Hemav² ofrece entre otros servicios agricultura de precisión a través del vuelos con cuadricópteros. Utilizan sensores multiespectrales y/o térmicos para generar mapas aéreos mediante vuelos autónomos que su suministran información relevante para la toma de decisiones (figura 1.4 (b)).

Posteriormente esa información es analizada estadísticamente para generar patrones de estados, similitudes de terreno, detección de patologías, etc, con los que finalmente se genera un informe con las conclusiones específicas.

Entrega de paquetes con drones.

La entrega de paquetería usando vehículos aéreos no tripulados es el futuro, aunque ya muy presente en prototipos en cuanto a la entrega de paquetes a domicilio se refiere. Las funcionalidades son numerosas, desde entrega de comida o compras realizadas a domicilio, hasta entrega de medicamentos o comida a personas en apuros o situaciones peligrosas.

²<http://hemav.com>



(a) Hemav.



(b) Finca acotada para el vuelo autónomo.

Figura 1.4: Uso de drones en agricultura de precisión.

Empresas como Amazon con su proyecto *Prime Air* (figura 1.5), DHL, Google con *Project Wing* o la NASA están trabajando para desarrollar sus propios sistemas de entrega de paquetería a domicilio a través de drones.



Figura 1.5: Amazon Prime Air

Drones en salvamento marítimo.

La Agencia de Medios de Vodafone (MEC) y los expertos de Trabajos con Drone³ (TcD) han desarrollado un proyecto de drones socorristas. Estos reducen hasta en tres veces el tiempo en llegar al lugar del accidente con respecto al socorrista tradicional.

En caso de incidente, el socorrista-piloto conducirá el drone con el mando de radio-control ayudándose de la cámara que lleva a bordo, cuando llega al punto de incidencia lanzara un par de flotadores a los que se pueden agarrar hasta que llegue el equipo de salvamento.

En la figura figura 1.6 se puede ver a un piloto de drones realizando pruebas de salvamento marítimo con un drone.



Figura 1.6: Drone de Salvamento Marítimo

Drones en el cine y televisión.

Hace tiempo que los robots teledirigidos participan en los rodajes, pero sólo recientemente han pasado de tener un papel secundario a convertirse en protagonistas. En la industria cinematográfica se ha añadido el uso del drone para grabar escenas que antes eran complicadas, costosas o imposibles de grabar. Han cobrado tanta importancia que hasta han creado certámenes como el *Flying Robot International Film Festival*⁴ para

³<http://trabajoscondrone.com>

⁴<http://friff.co>

premiar su uso en la grabación. Los drones nos proporcionan nuevas perspectivas en todo tipo de secuencias.

Películas como el Lobo de Wall Street, Jurassic World o Skyfall han incorporado entre sus métodos de grabación la utilización de drones. También se utilizan en el mundo de la televisión. Entre otros el famoso programa inglés *Top Gear* o el documental de la BBC Planeta Tierra. En la figura 1.7 podemos ver un drone grabando una escena.



Figura 1.7: Uso de un drone para grabación.

1.1.4. Sistemas de control de drones

Hay varios sistemas que nos permiten teleoperar o controlar un drone, los cuáles se pueden dividir en dos grupos, los controlados mediante radiofrecuencia y los que usan sistemas alternativos.

Radiocontrol: Es la técnica que permite el gobierno de un objeto a distancia de manera inalámbrica 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.

Sistemas alternativos: A parte del radiocontrol tenemos el control a través de WiFi. Este sistema consiste en la creación de una red WiFi por parte del dron 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 conexión y transferencia de datos.

La ventaja de estos sistemas es el 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 dron.

Empresas como DJI usan sistemas mixtos que consisten en teleoperar el dron vía radiocontrol, pero la gestión y visualización de la cámara 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 dron, 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 como referencia para desarrollar el nuestro. En la figura 1.8 se muestra cómo se teleopera el cuadricóptero ArDrone 2.0 de Parrot.



Figura 1.8: Sistema WiFi de control del drone de Parrot.

1.2. Tecnologías de comunicación 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 más actuales.

1.2.1. RTP

RTP son las siglas de *Real-time Transport Protocol* o Protocolo de Transporte en Tiempo Real, el cuál es un protocolo para aplicaciones 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. 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 estadísticas se puede hacer codificación adaptativa para adaptarse al medio. También trabaja sobre UDP y usa un número 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 protocolo 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.

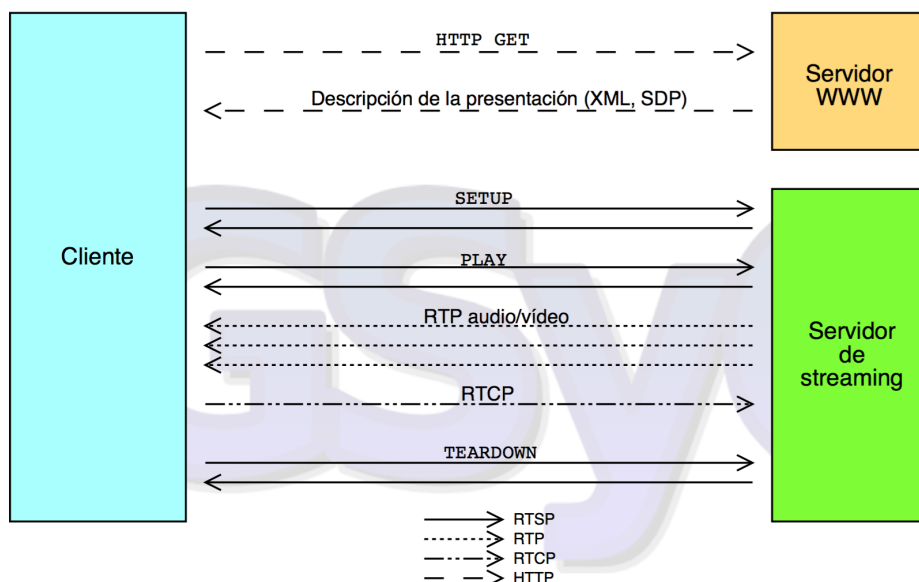


Figura 1.9: Ejemplo de conexión RTC, RTCP y RTSP

1.2.2. 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 más importantes 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).

En la figura 1.10 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.

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.

1.2.3. 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*).

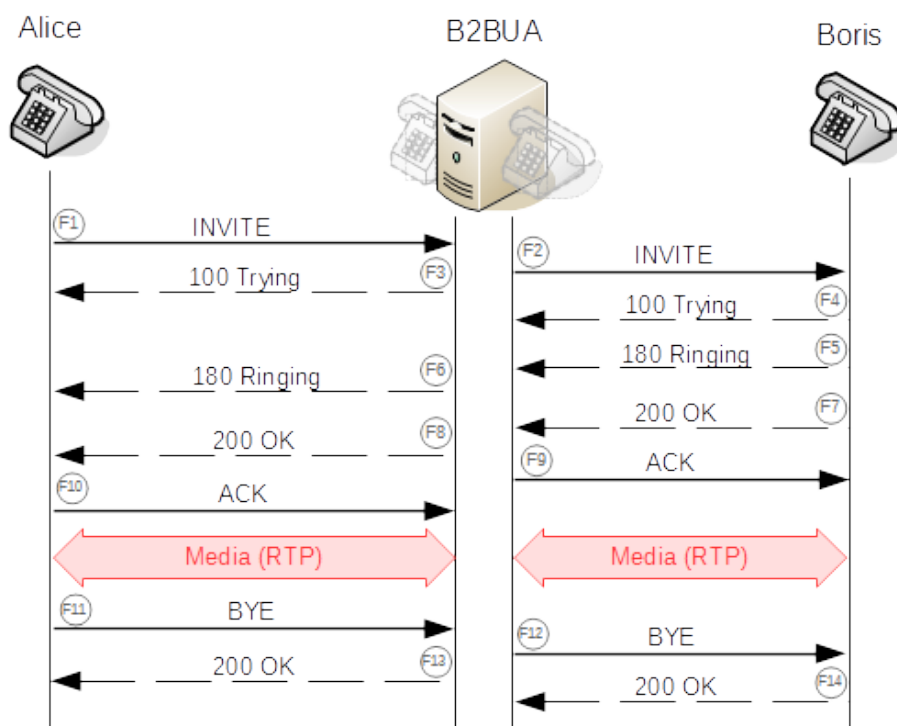


Figura 1.10: Ejemplo de conexión SIP

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.

1.2.4. 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 está 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 entre navegadores, sin necesidad de servidores internos, y está llamada a

ser el futuro de las comunicaciones en tiempo real.

1.3. Antecedentes

Como base para el proyecto tenemos los siguientes trabajos, desarrollados también por alumnos de la URJC. Todos estos proyectos tienen en común que utilizan como base las herramientas que nos ofrece JdeRobot, que es el marco *software* en el que se ubica el presente trabajo.

1.3.1. ArDroneServer

ArDroneServer⁵[5] es un *driver* para manejar drones desde la plataforma JdeRobot. Utiliza la SDK de ArDrone de Parrot, ha sido desarrollado por Alberto Martín como parte de su Grado Fin de Carrera. Esta aplicación está inspirada en los paquetes ROS *ardrone_brown* y *ardrone_autonomy*. Implementa tres interfaces que nos permiten recibir las imágenes del drone, controlar el drone y recoger información de los sensores y cambiar la configuración del drone. Junto a este servidor Alberto ha desarrollado una aplicación capaz de hacer seguimiento horizontal y vertical de objetos. En la figura 1.11 vemos a Alberto realizando un experimento, en el que el drone hace seguimiento vertical de un objeto.

1.3.2. Surveillance 4.0

Surveillance 4.0⁶[6] es un ejemplo de conexión web con sensores y actuadores, 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.

⁵<http://jderobot.org/Amartinflorido-tfg>

⁶<http://jderobot.org/D.castellanob-pfc>

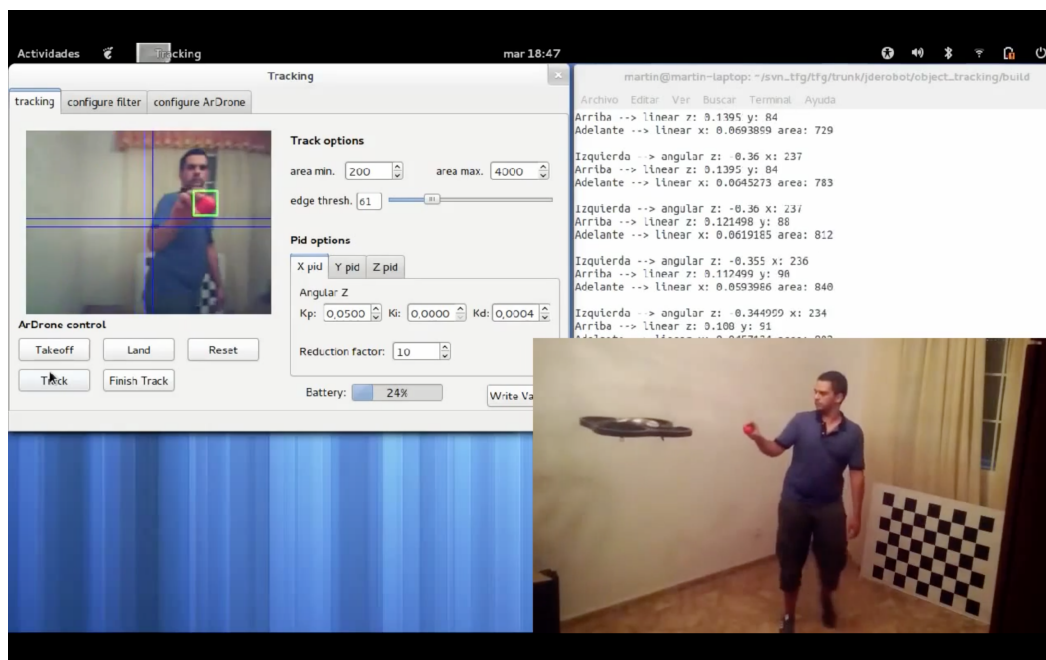


Figura 1.11: Seguimiento horizontal usando ardrone_server.

1.3.3. Surveillance 5.1

Otro ejemplo de conexión web con sensores y actuadores es Surveillance 5.1⁷[7] 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 1.12 se puede ver la aplicación.

1.3.4. Teleoperadores y visores Web en JdeRobot

Un ejemplo ya sin servidores web intermedios, directo entre sensores, robots y navegadores web es esta plataforma desarrollada por Aitor Martínez como su Trabajo Fin de Grado⁸[8]. Está compuesta por cuatro clientes Web: CameraViewJS, RGBD-ViewerJS, KobukiViewerJS y UavViewerJS, los cuales son las versiones web de las herramientas homónimas desarrolladas por JdeRobot. Estas nuevas herramientas hablan directamente con los servidores desarrollados por JdeRobot para acceder a los

⁷<http://jderobot.org/Aerobeat-colab>

⁸<http://jderobot.org/Aitormf-tfg>

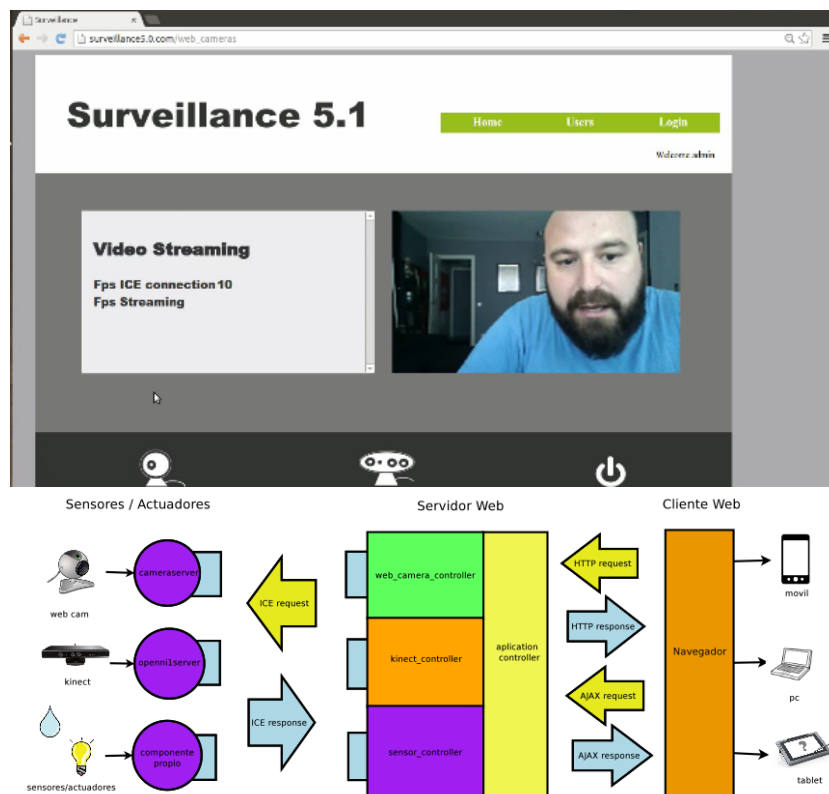


Figura 1.12: interfaz (a) y arquitectura (b).

sensores y robots, y como diferencia principal destacable no necesitan de servidores intermedios para funcionar, utilizando *websockets* para establecer las conexiones necesarias.

Las funcionalidades de estos clientes web son:

1. *CameraViewJS*: Cliente web similar a la herramienta *CameraView* para visualizar imágenes procedentes del servidor *Camerasetver*.
2. *RGBDViewerJS*: Cliente web similar a la herramienta *RGBDViewer* para visualizar datos de color y profundidad procedentes del servidor *Openni1Server*.
3. *KobukiViewerJS*: Teleoperador para manejar y ver los datos de los sensores de los robots *Kobuki* y *Pioneer* del laboratorio de robótica de la URJC. Versión web de *KobukiViewer*
4. *UavViewerJS*: Cliente web similar a la herramienta *UavViewer* para teleoperar drones tanto reales como simulados y ver los datos de sus sensores.

En el proyecto que aquí se presenta se desarrolla una aplicación web capaz de teleoperar un cuadricóptero usando los servidores desarrollados por JdeRobot para

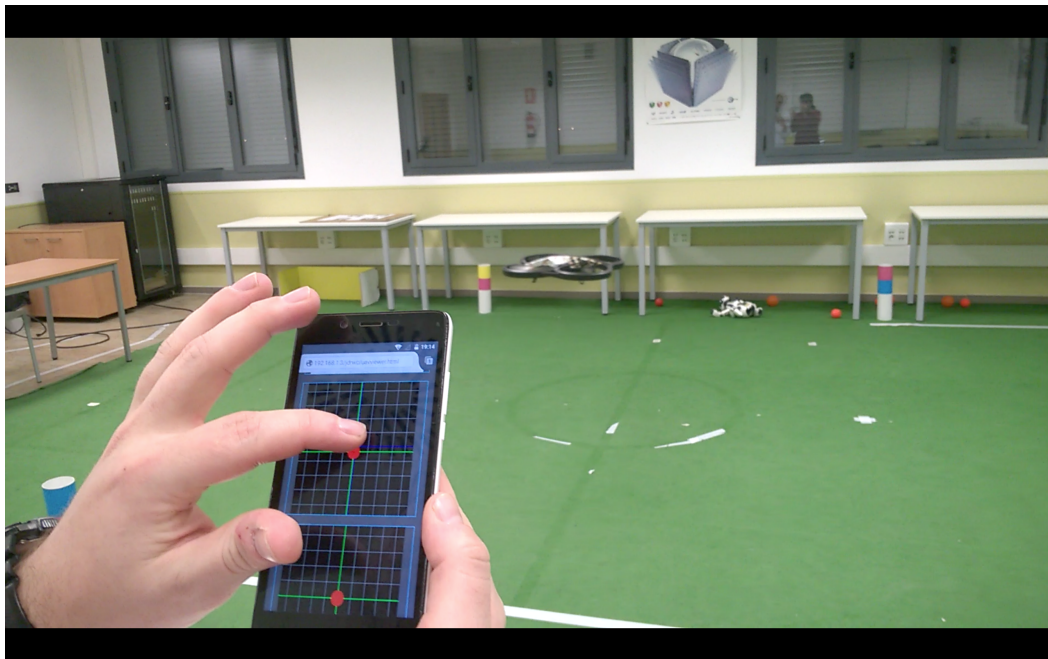


Figura 1.13: Cliente web UavViewer.js

conectarnos al drone, sus sensores y actuadores, y añadiendo la tecnología web de última generación *WebRTC* para establecer la conexión entre el ordenador que se comunicará con el drone y el ordenador remoto. Desde el navegador se podrán ver los valores de los sensores y la cámara a bordo del drone, además como ya he mencionado, de teleoperarlo. Estas conexiones se realizarán sin el uso de servidor intermedio, usando tecnologías en tiempo real que se conectarán mediante *WebSockets* de *JavaScript*.

Una vez presentada la introducción y el contexto del proyecto en este capítulo, continuamos exponiendo los objetivos que nos hemos marcado. En el capítulo tres se habla sobre las infraestructuras software en las que nos hemos ayudado para el desarrollo. Posteriormente en el capítulo cuatro veremos cómo hemos conseguido cumplir los objetivos marcados, exponiendo en el capítulo cinco las pruebas y experimentos realizados. En el último capítulo presentaremos las conclusiones del proyecto.

Capítulo 2

Objetivos

Una vez situado el Trabajo Fin de Grado en su contexto vamos a presentar los objetivos concretos que nos marcamos para la realización del mismo.

2.1. Objetivos

Como objetivo global nos proponemos teleoperar un drone desde dispositivos móviles como teléfonos o tabletas utilizando tecnologías web de última generación. Este objetivo se ha desglosado en tres subobjetivos concretos que explicamos a continuación.

Como escenario nos encontramos con un ordenador que irá a bordo del drone, y un navegador en una máquina remota. Durante toda la memoria nos referiremos como ordenador local, par local o navegador local al dispositivo que usaremos para conectarnos al cuadricóptero y el cuál deberá ir a bordo del drone, y ordenador remoto, par remoto o navegador remoto al ordenador desde el cuál teleoperaremos el vehículo.

- ***Conexión local:*** Como primer subobjetivo tenemos que desarrollar una conexión directa del navegador local con el *hardware* del drone. Esta conexión tiene que ser bidireccional, ya que tenemos que acceder a los sensores y actuadores del drone, así como a su cámara, pero también es necesario mandarle órdenes de movimiento.

Como requisito fundamental para esta conexión con el servidor de los actuadores y sensores a bordo es que tiene que ser en tiempo real y que sea lo suficientemente fluida y ligera como para que no introduzca ningún tipo de retardo.

- **Conexión multimedia entre navegadores:** El segundo punto con el que nos enfrentamos es utilizar una tecnología web moderna y actual con la que poder teleoperar el dron. Esta tecnología tiene que ser lo suficientemente versátil como para poder implementarla en nuestro proyecto. Por otro lado, al igual que la conexión local, tiene que ser bidireccional, poder transportar audio, vídeo y datos genéricos e introducir el mínimo retraso en la comunicación posible para tener una experiencia positiva y controlada del vuelo del dron.

Esta conexión deberá realizarse entre el ordenador local y un segundo ordenador, que será desde el que el usuario podrá teleoperar el vehículo.

- **Interfaz de usuario amigable:** Uno de los subobjetivos es desarrollar una interfaz web de usuario clara, que nos muestre de una manera lo más realista, clara y concisa posible los datos recogidos de los sensores del dron y la cámara para permitirnos conocer el estado de vuelo del dron en cada momento: altitud, inclinación, velocidad...

Esta interfaz deberá permitirnos el manejo del cuadricóptero de la forma mas realista y similar posible a los sistemas de control de drones que hemos hablado en la sección 1.1.4. Para que la experiencia de vuelo sea satisfactoria la interfaz tiene que tener unos controles de movimiento que sean sencillos, simples e intuitivos.

2.2. Metodología y plan de trabajo

La realización de un proyecto requiere una metodología que establezca las pautas a seguir y la planificación de las tareas que se deben llevar a cabo para cumplir los objetivos. Hemos escogido el modelo de *desarrollo en espiral*, ya que es un modelo ampliamente usado en la ingeniería de *software*. Este modelo define una serie de ciclos que se repiten en un bucle hasta el final del proyecto, dividiéndolo en varias subtarefas más sencillas y estableciendo puntos de control al final de cada iteración en los que se evalúa el trabajo realizado y se enfocan las nuevas tareas para continuar.

Esta metodología recibe su nombre por la forma de espiral que tiene su representación gráfica o diagrama de flujo, que podemos ver en la figura 2.1. En cada iteración se llevan a cabo las siguientes actividades:

- **Determinar los objetivos**, dividir en subobjetivos y fijar requisitos.

- **Analizar los riesgos** y factores que impidan o dificulten el trabajo y las consecuencias negativas que este pueda ocasionar.
- **Desarrollar** las tareas para lograr los objetivos según los requisitos especificados.
- **Planificar** las próximas fases tras evaluar el transcurso del proyecto.



Figura 2.1: Esquema general del desarrollo del proyecto.

Durante el ciclo de vida del proyecto se han hecho reuniones periódicas con el tutor. En ellas se evaluaban los avances logrados y se marcaba la hoja de ruta a tomar para los siguientes días de desarrollo. Si los puntos marcados en sesiones anteriores no se habían finalizado se ampliaba el plazo o se intentaba buscar otra manera de avance.

Para facilitar el seguimiento del proyecto se ha utilizado de un mediawiki¹ de Jde-Robot en el que se iba actualizando cada avance que se lograba, con explicaciones y vídeos e imágenes. Para el código fuente se ha empleado un repositorio en la plataforma web de control de versiones GitHub².

El plan de trabajo para todo el proyecto se puede dividir en las siguientes etapas:

¹<http://jderobot.org/Irodmar-tfg>

²<https://github.com/RoboticsURJC-students/2015-tfg-irodmar>

- **Familiarización con JdeRobot:** Primer contacto con esta plataforma y sus herramientas para conocer su funcionamiento.
- **Aprendizaje de tecnologías web necesarias:** Conocer las tecnologías web que van a ser necesarias para el desarrollo del proyecto. Entre ellas se encuentra WebRTC, HTML5, CSS3, WebGL, ThreeJS, o jQuery. Primer contacto también con el *middleware* ICE.
- **Desarrollo de la conexión local:** Creación de toda la infraestructura necesaria para la interconexión entre el navegador local y el drone.
- **Desarrollo conexión entre navegadores:** Desarrollo de la conexión remota que interconectará los dos pares.
- **Desarrollo interfaz web de usuario:** Desarrollo de la interfaz amigable para teleoperar el drone.
- **Experimentos:** Primero se realizan pruebas con el simulador, y cuando el código este suficientemente maduro se prueba con un drone real.

Capítulo 3

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 empleado, como apoyo para cubrir las partes a las que WebRTC no llega, ICEJS junto con la plataforma robótica JdeRobot. Estas tecnologías junto con el simulador robótico Gazebo nos han permitido probar, depurar y mejorar el código antes de realizar los experimentos sobre el drone real ArDrone, el cuál presento a continuación.

3.1. ArDrone de Parrot

El Parrot¹ ARDrone es un drone que comercializa la marca Parrot el cuál puede ser teleoperado con una aplicación desde 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 la 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 desarrollar 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 este proyecto hemos usado la versión ArDrone 2.0, el cuál cuenta como principales características de una cámara HD a 720p y 30fps y una altitud máxima de vuelo

¹<http://www.parrot.com/es/productos/ardrone-2/>



Figura 3.1: ArDrone de Parrot

de 100 metros de altura. Tiene una autonomía de vuelo de aproximadamente 11-12 minutos, y su carga máxima de pago ronda los 400 gramos de peso. En su interior tiene un microprocesador AR M Cortex A8 de 32 bits a 1GHz con DSP de vídeo a 800MHz el cuál corre Linux en su interior. En este proyecto usaremos el ArDrone 2.0 para conectar un navegador web externo con el microprocesador que corre a bordo del drone y teleoperarlo a distancia.

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

3.2. Simulador robótico Gazebo

Gazebo² 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

²<http://gazebosim.org>

nuestros algoritmos de una forma parecida a las condiciones que 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 plugin para Gazebo que contiene el modelo, el mundo y los interfaces necesarios para simular el ArDrone de Parrot, de tal manera que podremos probar el código para ir puliendo y conseguir que sea lo suficientemente estable antes de 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 del TFG y 100% compatible con el plugin de JdeRobot mencionado, el cuál se explica a continuación en mayor profundidad.

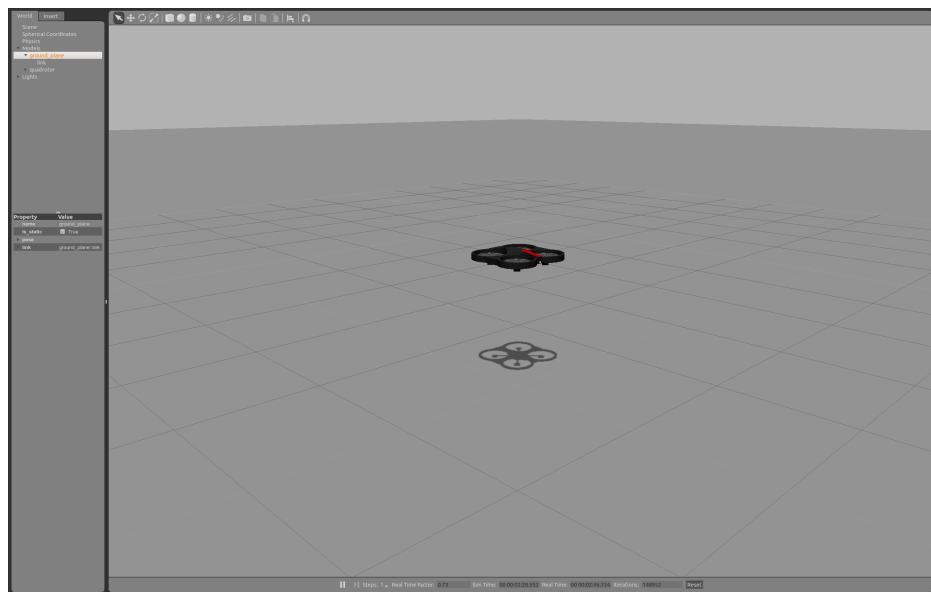


Figura 3.2: Simulador Gazebo

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

3.3. JdeRobot

JdeRobot³[3] es un proyecto desarrollado por el laboratorio de robotica de la Universidad Rey Juan Carlos. 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 interoperan 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 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, 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 para añadir el ArDrone a Gazebo con las mismas funcionalidades que el drone real. Este plugin nos permite abstraernos de las conexiones a más bajo nivel, como la lectura de sensores o en envío de comandos a los motores, simplificando el acceso a los mismos 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 necesarias sobre los motores y actuadores según las órdenes de movimiento dadas. De cara al usuario ofrece seis interfaces ICE, a las que puedes conectar tu aplicación. En la figura 3.3 puedes ver la estructura general del componente.

³<http://jderobot.org>

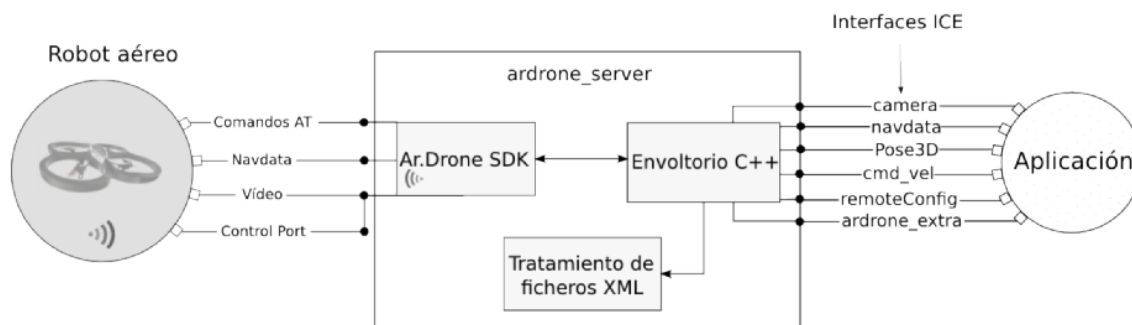


Figura 3.3: Estructura de ArDrone_Server

3.4. ICE

ICE⁴ o *Internet Communication Engine* es un entorno RPC orientado a objetos desarrollado por la empresa 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 más 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 3.4: Esquema de ICE JS

⁴<https://zeroc.com>

3.5. WebRTC

Web Real-Time Communication⁵[9] (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 permite desarrollar estas aplicaciones para que funcionen directamente desde un navegador a otro *sin pasar por servidor intermedio* para el trasiego de datos.

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 NATs.

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 los pares, facilitando el intercambio de paquetes de la negociación WebRTC.

WebRTC está compuesto de 3 API's:

- *getUserMedia*: para la adquisición del flujo local de audio y vídeo.
- *RTCPeerConnection*: para la comunicación de audio y vídeo.

⁵<https://webrtc.org>

- *RTCDataChannel*: para la 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.

En las siguientes secciones vamos a desglosar y explicar con detalle cómo funciona el intercambio de paquetes de señalización así como cada una de las API's que componen WebRTC.

3.6. 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 intercambio 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 3.5)] 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 3.5: 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 las API's.

3.6.1. Estableciendo el Flujo Audiovisual: Descriptores de Sesión

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.

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. 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.

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 descriptors 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 descriptors de sesión indicando el tipo de sesión que se suministra. En la figura 3.6 podemos ver el esquema de intercambio de paquetes SDP y el ajuste de los mismos en cada par según corresponde.

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

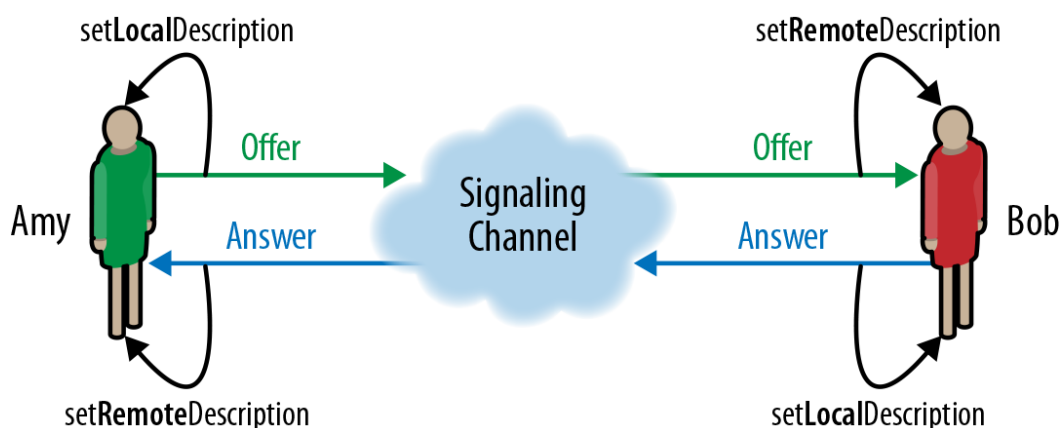


Figura 3.6: Intercambio y establecimiento de los SDP en cada par.

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.

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 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 procesamiento 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:d0TZKZNV109RSGsEGM63JXT2
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=imageattr:97 send [x=[128:16:1280],y=[72:9:720]] recv [x
    =[128:16:1280],y=[72:9:720]]
30 a=ssrc:4429951804 cname:Q/NWs1ao1HmN4Xa5
31
32 m=application 9 UDP/DTLS/SCTP webrtc-datachannel
33 c=IN IP4 0.0.0.0
34 a=mid:d1
35 a=fmtp:webrtc-datachannel max-message-size=65536
36 a=sctp-port 5000
37 a=fingerprint:sha-256 6B:8B:F0:65:5F:78:E2:51:3B:AC:6F:F3:3F:46:1B:35
38                        :DC:B8:5F:64:1A:24:C2:43:F0:A1:58:D0:A1:2C:19:08
39 a=setup:active

```

Listado 3.1: Ejemplo paquete SDP

El cometido principal del intercambio de SDP es la negociación de vídeo. 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.

3.6.2. 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, preferiblemente directas.

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. A 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 para hacer de intermediario trasegador de los datos multimedia.

Como podemos ver, ICE necesita de servidores externos para obtener la tupla de dirección IP y puerto públicos necesarios para el otro par si está fuera de la misma red local. STUN es un protocolo estandarizado para descubrir direcciones IP públicas 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 directa 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.

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.

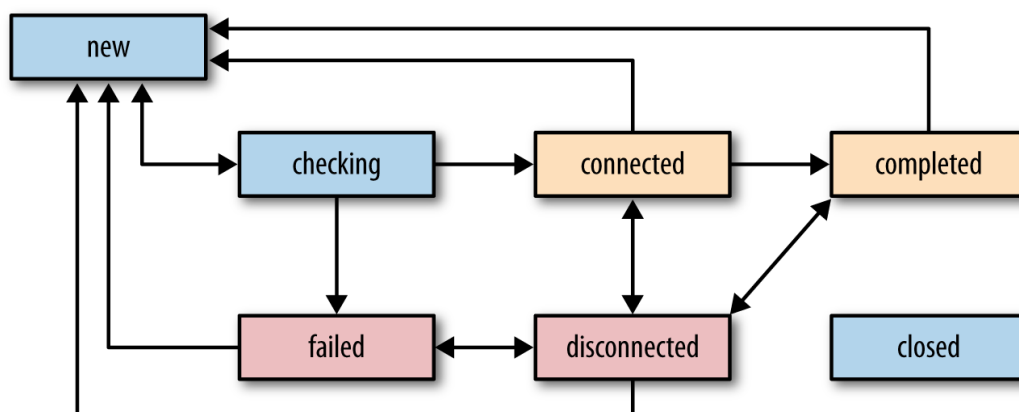


Figura 3.7: Máquina de estados y transiciones ICE.

Una vez los dos Agentes ICE tienen una lista completa de los Candidatos ICE de ambos pares, cada agente comprueba emparejando ambas listas cuáles 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 continúa 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 públicas 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 Trickling 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
```

Listado 3.2: 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.

3.7. WebRTC: API getUserMedia

getUserMedia es la API encargada de suministrar 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 procesamiento de audio y vídeo (figura 3.8) que harán todas estas cosas por nosotros.

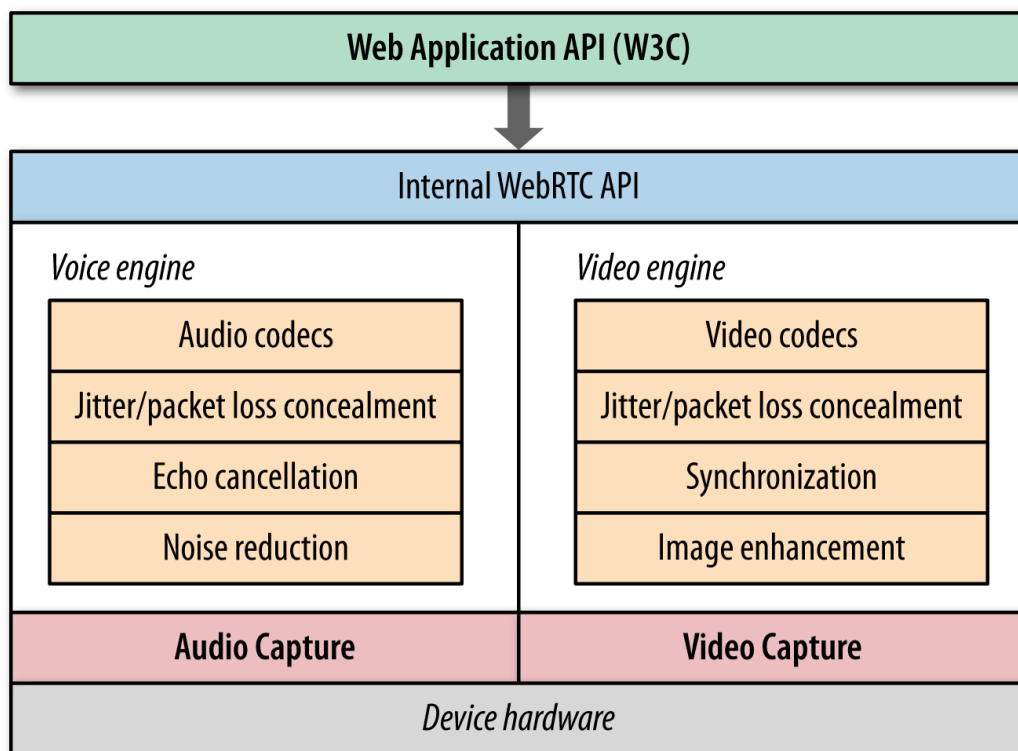


Figura 3.8: Motores de audio y vídeo de WebRTC

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 3.9) es la forma en la que nos suministra los flujos esta API.

- 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.

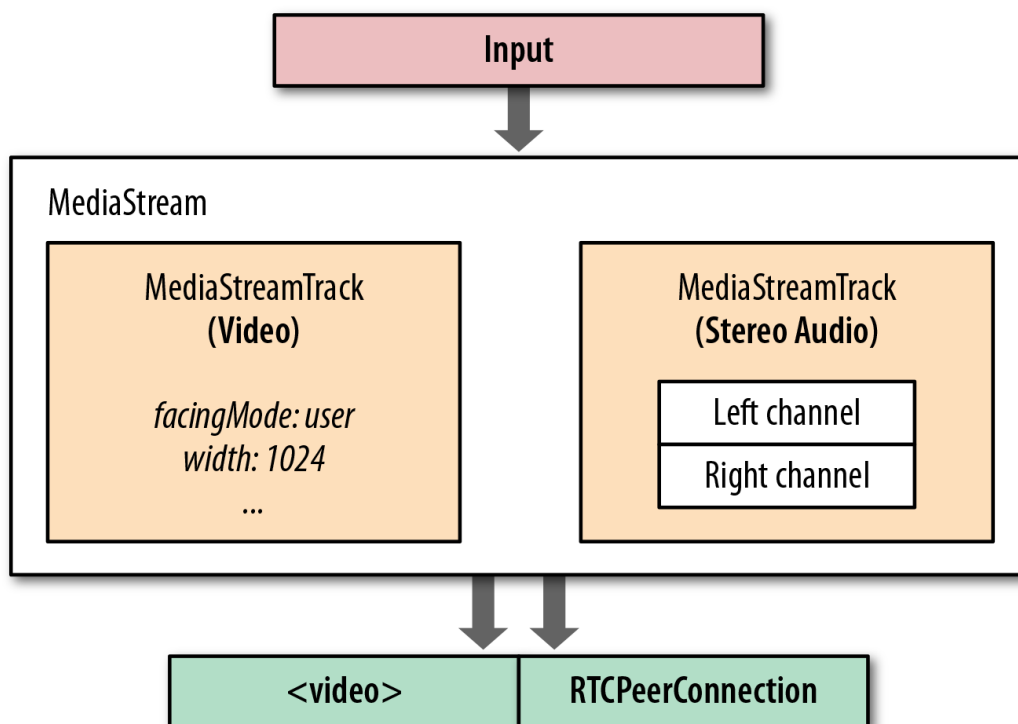


Figura 3.9: MediaStream

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 audiovisual 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.

```

1  var constraints = {
2      audio: false,
3      video: {
4          width: { min: 1024, ideal: 1280, max: 1920 },
5          height: { min: 576, ideal: 720, max: 1080 },
6      }
7  };
8
9  navigator.getUserMedia(constraints, handleUserMedia,
10     handleUserMediaError);
11
12 function handleUserMedia(stream){
13     var video = document.querySelector('video');
  
```

```
13     video.src = window.URL.createObjectURL(stream);
14 }
15
16 function handleUserMediaError(error){
17     console.log('getUserMedia error: ', error);
18 }
```

Listado 3.3: Llamada a función `RTCPeerConnection`

3.8. 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);
```

Listado 3.4: 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 Traversal. *pcConstraints* también es una variable a la que se le pueden añadir una serie de restricciones como *RtpData-Channels*, 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 vicacidad son mucho mas importantes que la fiabilidad (*reliability*).

Este requerimiento de vivacidad 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 3.10)].

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

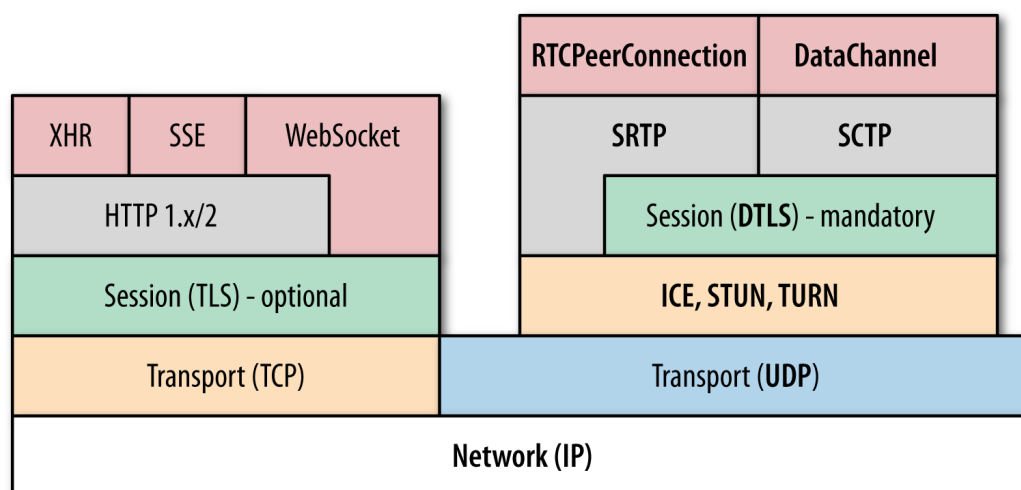


Figura 3.10: Pila de protocolos WebRTC

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.

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 (*Secure Real-time Transport Protocol*) es el protocolo que nos proporciona multiplexación de los diferentes flujos y nos proporciona control del flujo y de congestión. 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 número 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 reci-

bido, *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.

3.9. 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 enviamos 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 una mayor sobrecarga. La segunda reduce la sobrecarga dando lugar a tener una conexión más fluida y 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.

Como las API's anteriores llamamos a RTCDataChannel con una variable para las opciones de configuración:

- *Ordered*: Booleano para indicar si queremos que nos garantice la entrega ordenada de paquetes.

	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 3.11: Capacidades de RTCDataChannel.

- *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.
- *Negotiated*: 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 deben estar encriptados cuando se transmiten. 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 3.10, *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 reagrupación 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 enviamos van encriptados y seguros hasta nuestro destinatario. Por otro lado SCTP, como vemos en la tabla 3.11 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.

Capítulo 4

Desarrollo software

Una vez nos hemos situado en el contexto en el que se ubica este proyecto, y hemos expuesto los objetivos a cumplir y las herramientas necesarias, nos adentramos a explicar en este capítulo las soluciones desarrolladas para llegar a buen puerto.

4.1. Diseño

El escenario que se presenta consta de cuatro elementos principales: el drone, el ordenador a bordo, el ordenador remoto y el usuario humano final. El diseño consta de dos conexiones de tal manera que permitan al usuario final poder teleoperar el cuadricóptero: la primera conexión es entre un navegador web en el ordenador local, el cuál irá a bordo del drone, y el propio drone; la segunda conexión es entre este navegador local y un navegador remoto, dónde se creará una interfaz web amigable y de fácil uso para el control a distancia del drone por parte del usuario final.

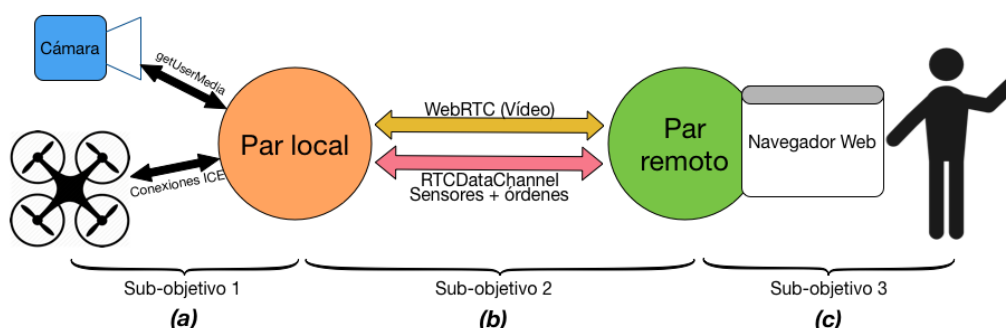


Figura 4.1: Esquema general del proyecto.

En la figura 4.1 se representa el esquema general del proyecto con los cuatro actores y las sub-divisiones realizadas. A continuación se expone una vista general a la solución desarrollada para cada uno de esos sub-problemas:

- (a) **Conexión local:** consta de dos conexiones: se ha utilizado la API *getUserMedia* de WebRTC para acceder a las imágenes de una cámara que también irá a bordo del drone; por otro lado se ha establecido la conexión entre el drone y el navegador utilizando el plugin ICEJS, el cuál nos permite conectarnos con interfaces ICE a través de *websockets* en el navegador.
- (b) **Conexión entre navegadores:** Una conexión punto a punto entre navegadores con la API *RTCPeerConnection* para transferir las imágenes de la cámara, y la API *RTCDataChannel* para transferir los datos de los sensores y las órdenes dadas.
- (c) **Cliente web:** trabaja como transductor para representar los datos recibidos de los sensores del drone y la cámara por un lado, y por el otro de recoger y enviar las órdenes de vuelo dadas por el usuario final.

De manera general en cuanto a la estructura global el par local deberá establecer la conexión con el drone, acceder a sus sensores y actuadores y a la cámara. Deberá también establecer una conexión con el ordenador remoto y enviarle todos estos datos obtenidos del drone. Del par remoto recibirá las órdenes y comandos de movimiento, que deberá enviárselos al drone.

El par remoto deberá establecer la conexión con el par local. De éste recibirá todos los datos del drone y deberá representarlos de una manera que el usuario final pueda conocer la situación de vuelo del drone en cada momento. Deberá tener una interfaz amigable que le permita recoger las órdenes de movimiento dadas por el usuario, y enviárselas al par local.

Como primer problema se presentó decidir cuál de los dos ordenadores que necesitamos para la conexión WebRTC sería el que realizaría la llamada y en qué momento del flujo. Éste no es un problema trivial, ya que la selección de uno u otro haría que el desarrollo de la aplicación fuese completamente distinto.

Se optó por que el par que lleva la batuta de la conexión fuese el ordenador local, ya que éste a su vez también es el encargado de establecer la conexión con el drone. De esta manera tenemos un par que es el que actuará de maestro, estableciendo ambas conexiones en los momentos oportunos.

Como ya se comentó en la sección 3.6 el momento en el que se envía y se recibe cada paquete de información es crítico en este sistema de señalización de oferta/respuesta, por lo que el flujo de la comunicación se diseñó y se ha desarrollado como se muestra en la figura 4.2.

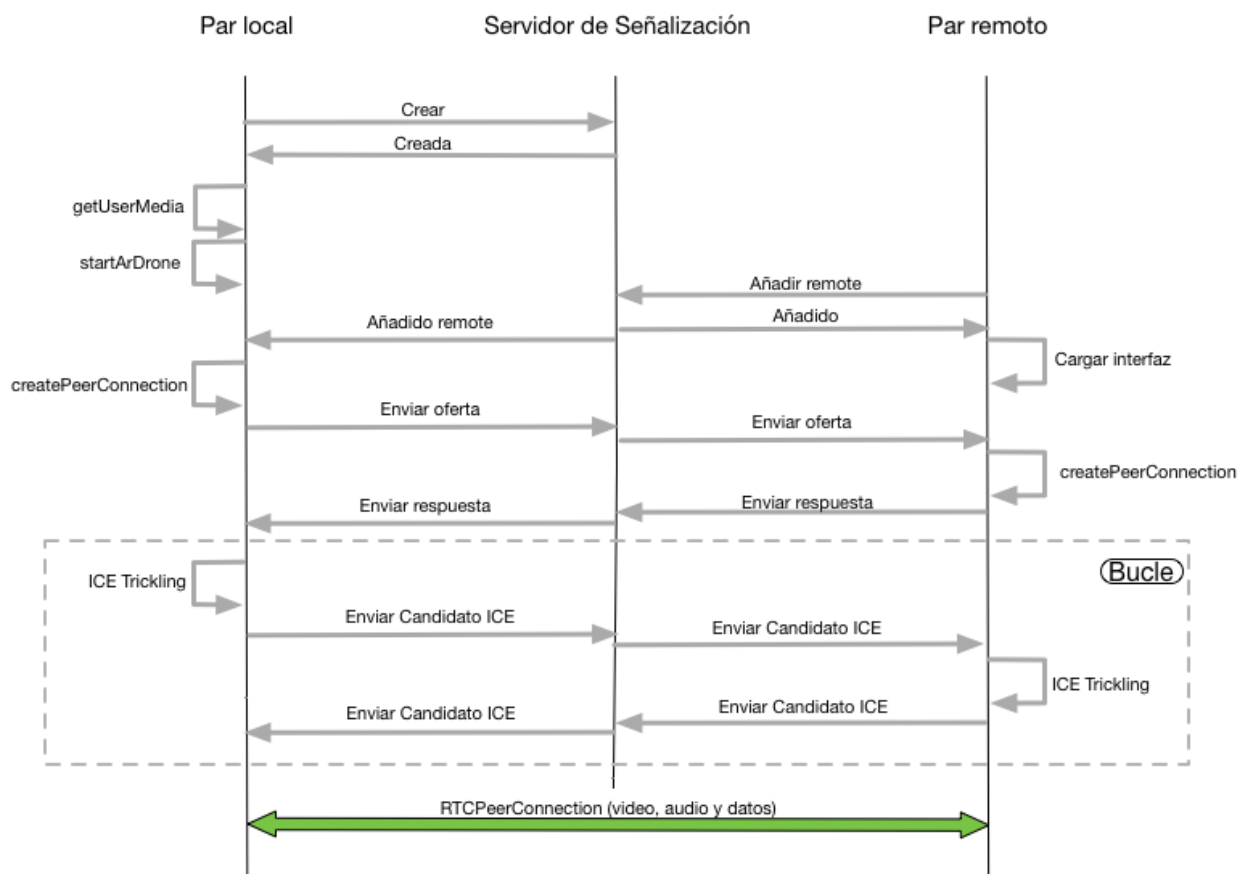


Figura 4.2: Flujo de llamada del proyecto

Como se puede observar el par local es el que lleva la iniciativa de la conexión. En primer término inicia la comunicación con el servidor de señalización. Una vez que éste le contesta afirmativamente a su mensaje de creación de la comunicación inicia un proceso en el que se conecta al cuadricóptero y accede a la cámara. Una vez realizados estos dos procesos espera a que un par remoto quiera unirse a la conexión.

Cuando el par remoto accede al servidor de señalización, éste le envía un mensaje al par local indicando que un par remoto se ha conectado. En este momento el par local inicia la creación de la conexión *RTCPeerConexction* de WebRTC. Primero se produce el intercambio de paquetes SDP a través del servidor de señalización, y posteriormente el de ICE. Una vez finalizados ambos ya tenemos establecida la conexión

WebRTC entre ambos pares.

4.2. Conexión local drone-navegador

Esta parte la hemos solucionado dividiendo el problema en dos sub-problemas. Por un lado la conexión con el servidor encargado de conectarse al cuadricóptero, *ardrone_server*. A través de esta conexión seremos capaces de conectarnos a los elementos, sensores y actuadores del drone. Por otro obtener un flujo de vídeo para poder visualizarlo en la parte remota. Este problema lo hemos solucionado incorporando una cámara a bordo del drone que irá conectada al par local y a la cual accederemos desde el navegador con la API *getUserMedia* que nos brinda WebRTC.

Se explica a continuación la conexión con el servidor y posteriormente el acceso a la cámara con *getUserMedia*.

4.2.1. ArDrone_server y WebSockets

Para establecer la conexión con el drone se ha utilizado el componente **ardrone_server** de JdeRobot. Este componente son dos piezas. Un plugin en Gazebo para el drone simulado, y el servidor **ardrone_server** para el drone real. Estas dos piezas cuentan con las mismas interfaces de conexión ICE por lo que con un único desarrollo es posible utilizar indistintamente tanto la versión simulada con Gazebo como el drone real.

Esta conexión se ha dividido en dos partes, servidor y cliente. En la parte de servidor se ha tratado su configuración y la activación de *WebSockets* como protocolo sobre el que van mensajes ICE, y la parte del cliente se ha desarrollado desde cero el código *JavaScript* necesario para establecer y controlar la conexión.

Servidor

Los componentes JdeRobot utilizan, como ya hemos visto, interfaces ICE para el intercambio de información del que se ocupan. Los navegadores no tienen capacidad de usar el *middleware* ICE directamente. Para poder conectarnos con estas interfaces ICE hemos tenido que instalar el plugin ICE for JavaScript, o ICEJS. Este plugin nos habilita la opción de conectarnos a estas interfaces directamente desde el navegador

usando *websockets*.

Aunque estaba disponible la versión 3.6 de ICE¹ la cuál trae en la *suite* instalado de serie ICE_JS, hemos usado la versión 3.5, ya que es la versión compatible con la versión actual de JdeRobot². En esta versión ICE_JS es un plugin el cuál hay que instalarlo descargando el código fuente desde la página de Zeroc³ y compilarlo.

Una vez instalado hay que activar este plugin en el servidor ICE. Para ello hay que añadir la siguiente línea en el archivo de configuración del servidor ICE, en nuestro caso *ardrone_server*:

```
1 # ICE-JS
2 Ice.Plugin.IceWS=IceWS:createIceWS
```

Listado 4.1: Activación del plugin ICEJS

Posteriormentes, en el mismo archivo hay que indicarle las direcciones IP y los puertos de cada interfaz de conexión. Cada conexión se corresponderá con un *Web-Socket*, y la nomenclatura es la siguiente:

```
1 # ICE-JS
2 :ws -h ip -p puerto
```

Listado 4.2: Formato *endpoints* de los *WebSocket* de ICEJS

El servidor tiene hasta un máximo de seis interfaces diferentes a las que te puedes conectar. Cada una de estas interfaces nos ofrece un servicio diferente. Para nuestro proyecto hemos usado cuatro de esas interfaces:

- **Pose3D:** Con esta interfaz accedemos a los datos de orientación y posición del cuadricóptero (x, y, z, h y *quaternion*).
- **Navdata:** Esta interfaz nos proporciona los datos de navegación procedentes de los sensores, como la velocidad de las componentes (x, y, z) del drone, altitud, la velocidad del viento, el nivel de batería, etc.
- **CMDVel:** Esta interfaz es la encargada de recibir las órdenes de movimiento.
- **BaseExtra:** Esta interfaz nos da funcionalidades extra como las órdenes de aterrizaje o de despegue del cuadricóptero.

¹<https://doc.zeroc.com/display/Ice36/Home>

²<http://jderobot.org/Manual-5>

³<https://zeroc.com>

Así pues, esta es la forma final que tiene nuestro archivo de configuración:

```

1 # Ice-JS
2 Ice.Plugin.IceWS=IceWS:createIceWS
3
4 # Variables de control para ver traceroutes de las conexiones ICE.
5 #Ice.Trace.Network = 3
6 #Ice.Trace.Protocol=1
7
8
9 ArDrone.Pose3D.Endpoints=default -h 0.0.0.0 -p 9998:ws -h 0.0.0.0 -p
   19000
10 ArDrone.Pose3D.Name=ardrone_pose3d
11
12 ArDrone.RemoteConfig.Endpoints=default -h 0.0.0.0 -p 9997
13 ArDrone.RemoteConfig.Name=ardrone_remoteConfig
14
15 ArDrone.Navdata.Endpoints=default -h 0.0.0.0 -p 9996:ws -h 0.0.0.0 -p
   15000
16 ArDrone.Navdata.Name=ardrone_navdata
17
18 ArDrone.CMDVel.Endpoints=default -h 0.0.0.0 -p 9995:ws -h 0.0.0.0 -p
   11000
19 ArDrone.CMDVel.Name=ardrone_cmdvel
20
21 ArDrone.Extra.Endpoints=default -h 0.0.0.0 -p 9994:ws -h 0.0.0.0 -p
   17000
22 ArDrone.Extra.Name=ardrone_extra
23
24 ArDrone.NavdataGPS.Endpoints=default -h 0.0.0.0 -p 9993
25 ArDrone.NavdataGPS.Name=ardrone_navdatagps

```

Listado 4.3: Archivo de configuración de ArDrone_Server.

A parte de los *endpoints* es también importante configurar correctamente el nombre de cada uno de las interfaces, ya que será necesario para una correcta conexión desde el navegador. La dirección ip 0.0.0.0 indica que la dirección de escucha del servidor es la ip local del equipo. Se puede también observar que hemos creado únicamente *WebSockets* para las cuatro interfaces anteriormente descritos.

Cliente

Una vez configurado el servidor hemos creado el método **ArDrone.js** en el navegador local, el cuál es el encargado de conectarse y manejar la conexión con el servidor **ardrone_server**.

Para establecer una comunicación ICE lo primero que se ha creado han sido las variables ICE necesarias. Estas han sido creadas de la siguiente manera:

```
1 // Variable ICE para la conexion
2 var id = new Ice.InitializationData();
3 id.properties = Ice.createProperties();
4 id.properties.setProperty("Ice.Trace.Network", "3"); // Propiedad
   para tracear la conexion
5 id.properties.setProperty("Ice.Trace.Protocol", "1"); // Propiedad
   para tracear la conexion
6 var communicator = Ice.initialize(id);
```

Listado 4.4: Variables para la conexión ICE.

Por si fallase la comunicación ICE se le ha añadido a la variable *id* una propiedad con la que podemos seguir la ruta de la comunicación y detectar en qué punto se produce el fallo.

Posteriormente es necesario crear una variable que actuará como *proxy*. Es necesario crear una variable por cada interfaz a la que necesitemos conectarnos. Esta es la nomenclatura que sigue:

```
1 var proxy = communicator.stringToProxy("nombre_interfaz:ws -h " + ip
   + " -p " + "puerto");
```

Listado 4.5: Nomenclatura de variable que actuará como *proxy*

Nótese en la nomenclatura que donde pone *nombre_interfaz* se corresponde con el nombre que le hemos asignado al *endpoint* en el archivo de configuración del servidor. Asimismo el puerto también se corresponde con el configurado.

La comunicación con las interfaces se realiza mediante el objeto promesa o *promise* de JavaScript. Este objeto se usa para las comunicaciones asíncronas y se caracteriza por tener tres estados: pendiente, cumplida o rechazada. Cuando una promesa ha sido llamada puede presentar el estado cumplida o rechazada, lo que nos permite llamar al argumento correspondiente y poder actuar en consonancia. De esta manera podemos hacer que métodos asíncronos actúen como métodos síncronos.

El núcleo de la conexión con el servidor *ardrone_server* queda como sigue:

```

1  // base extra connection
2  var baseextra = communicator.stringToProxy("ardrone_extra:ws -h " +
    ip + " -p " + baseextraPort);
3  jderobot.ArDroneExtraPrx.checkedCast(baseextra).then(
4      function(ar){
5          extraProxy = ar;
6          console.log("extraProxy connected: " + ar);
7      },
8      function(ex, ar){
9          console.log("extraProxy NOT connected: " + ex);
10     }
11 );
12
13 // NavData
14 var basenavdata = communicator.stringToProxy("ardrone_navdata:ws -h "
    + ip + " -p " + navdataProxyPort);
15 jderobot.NavdataPrx.checkedCast(basenavdata).then(
16     function(ar){
17         console.log("navdataProxy connected: " + ar);
18         navdataProxy = ar;
19         navdataProxy.getNavdata().then(
20             function(navdata){
21                 if (navdata.vehicle == ARDRONE_SIMULATED) {
22                     virtualDrone = true;
23                     console.log("virtualDrone = true");
24                 } else {
25                     virtualDrone = false;
26                     console.log("virtualDrone = false");
27                 }
28             },
29             function (ex, ar){
30                 console.log("Fail getNavdata() function: " + ex);
31             }
32         );
33     },
34     function (ex, ar){
35         console.log("navdataProxy NOT connected: " + ex);
36     }
37 );
38
39 // CMDVelPrx
40 var basecmdVel = communicator.stringToProxy("ardrone_cmdvel:ws -h " +
    ip + " -p " + cmdVelProxyPort);
41 jderobot.CMDVelPrx.checkedCast(basecmdVel).then(
42     function(ar){
43         console.log("cmdVelProxy connected: " + ar);
44         cmdVelProxy = ar;
45     },
46     function(ex, ar){

```

```

47     console.log("cmdVelProxy NOT connected: " + ex);
48 }
49 );
50
51 // Pose3D
52 var basepose3D = communicator.stringToProxy("ardrone_pose3d:ws -h " +
53     ip + " -p " + pose3DProxyPort);
54 jderobot.Pose3DPrx.checkedCast(basepose3D).then(
55     function(ar){
56         console.log("pose3DProxy connected: " + ar);
57         pose3DProxy = ar;
58         window.po = pose3DProxy;
59         resolve("Stuff worked!");
60         pose3DProxy.getPose3DData().then(
61             function (ar){
62                 console.log("getPoseDData().");
63                 pose = ar;
64             },
65             function(ex, ar){
66                 console.log("Fail call getPoseDData().");
67             });
68     },
69     function(ex, ar){
70         console.log("pose3DProxy NOT connected: " + ex)
71     });

```

Listado 4.6: Núcleo ArDrone

Tenemos cuatro promesas con sus respectivos argumentos de éxito o fallo, una para cada interfaz ICE. Las variables que nos devuelven los argumentos de éxito se guardan en las variables correspondientes de cada interfaz (`extraProxy`, `navdataProxy`, `cmdVelProxy`, `pose3DProxy`). Estas variables son importantes ya que son las que utilizaremos en las funciones manejadoras que se explican a continuación.

En este punto ya estamos conectados con las interfaces del servidor, y por consiguiente, con el dron. Para poder teleoperar el dron hay que crear unas funciones que serán los manejadores. Por un lado hemos creado las funciones en JavaScript de aterrizaje y de despegue. Estas funciones utilizan la interfaz *ardrone_extra*:

```

1 // extraProxy functions
2 function takeoff() {
3     extraProxy.takeoff().then(
4         function(ar){
5             console.log("Take Off.");
6         },
7         function(ex, ar){

```

```

8         console.log("Take Off failed.")
9     }
10    );
11 }
12
13 function land() {
14     extraProxy.land().then(
15         function(ar){
16             console.log("Landing.");
17         },
18         function(ex, ar){
19             console.log("Landing failed: " + ex)
20         }
21     );
22 }

```

Listado 4.7: Funciones aterrizaje y despegue.

Ejecutar estas órdenes en el drone se realiza en JavaScript también con el objeto promesa, con sus respectivos argumentos de éxito o error en la ejecución de la misma.

Las interfaces ICE *Navdata* y *Pose3D* son las encargadas de suministrar todos los datos de navegación de los sensores. Las funciones JavaScript con las que accedemos a estas interfaces y actualizamos todos estos datos son las siguientes:

```

1 function updateNavData() {
2     navdataProxy.getNavdata().then(
3         function(ar){
4             navdata = ar;
5             //console.log("updateNavData()");
6         },
7         function (ex, ar){
8             console.log("Fail getNavdata() function." + ex)
9         }
10    );
11 }
12
13 function updatePose(){
14     pose3DProxy.getPose3DData().then(
15         function (ar){
16             pose=ar;
17             //console.log("getPose3DData. ")
18         },
19         function(ex, ar){
20             console.log("Fail call getPoseDData(): " + ar);
21         });
22 }

```

Listado 4.8: Variables actualización datos de los sensores.

Llamando a estas funciones periódicamente tenemos actualizados en todo momento los datos de navegación procedentes de los sensores: brújula, posición, velocidad, altitud...

Para poder teleoperar el drone se ha implementado una función que es la encargada de enviarle las órdenes de movimiento. Estas órdenes nos llegan desde el par remoto a través de la conexión entre navegadores que explicaremos con mas detenimiento más adelante.

```
1 function sendVelocities () {  
2     cmdVelProxy.setCMDVelData(cmd).then(  
3         function(ar){  
4             //console.log("sendVelocities.");  
5         },  
6         function(ex, ar){  
7             console.log("sendVelocities failed.")  
8         }  
9     );  
10 }
```

Listado 4.9: Función manejadora de las órdenes.

Esta función admite una variable como argumento, (*cmd*) la cuál contiene los parámetros que recibimos del drone. Esta variable es una variable CMDVel de JdeRobot y tiene esta estructura:

```
1 var cmd = new jderobot.CMDVelData;  
2 cmd.linearX=0.0;  
3 cmd.linearY=0.0;  
4 cmd.linearZ=0.0;  
5 cmd.angularZ=0.0;  
6 cmd.angularX=0.0;  
7 cmd.angularY=0.0;
```

Listado 4.10: Variable CMD

Contiene seis parámetros que son las velocidades lineales X, Y, Z y las angulares en los mismos ejes. Estas velocidades pueden variar desde 0.0 hasta 1.0 en intervalos de 0.1.

4.2.2. getUserMedia

La interfaz que no hemos utilizado de las que nos ofrece el servidor `ardrone_server` es la interfaz `cameraserver`. Esta interfaz se encarga de recoger las imágenes de la cámara.

ra integrada en el dron. La razón por la que no usar la interfaz del dron que accede a la cámara es que queríamos utilizar la tecnología que nos brinda WebRTC con drones, y esto incluye el acceso a una cámara con la API *getUserMedia*. Esta cámara se conecta a nuestro ordenador local a través de una conexión USB. Cámara y MiniPC van a bordo del dron.

Como WebRTC aún no es una norma, para acceder a la cámara desde cualquier navegador debemos crear una variable que sea compatible con todos los que tengan implementado las APIs de WebRTC. Para ello hay que añadirle los prefijos correspondientes de cada navegador:

```
1 navigator.getUserMedia = navigator.getUserMedia || navigator.  
  webkitGetUserMedia || navigator.mozGetUserMedia;
```

Listado 4.11: Variable de *getUserMedia*.

Acceder a la cámara con *getUserMedia* se hace a través de una función que admite tres argumentos. El primer argumento que admite la función es una variable en la que le indicamos las restricciones que queremos: audio, vídeo, solo uno de ellos, resoluciones... Las configuraciones para nuestro proyecto son las siguientes:

```
1 var constraints = {  
2   audio: false,  
3   video: {  
4     width: { ideal: 1280, max: 1920 },  
5     height: { ideal: 720, max: 1080 },  
6   }  
7 };
```

Listado 4.12: Restricciones de *getUserMedia*

Sólo accedemos al vídeo de la cámara, ya que el audio no lo necesitamos para el proyecto. Por otro lado indicamos una resolución ideal, de 1280x720 píxeles. Si la cámara que le conectamos al ordenador tiene más capacidad restringimos su resolución a HD (1920x1080 píxeles).

Los siguientes dos argumentos son dos llamadas de vuelta o *callbacks*. El primero es el callback de éxito, y el segundo el de error. Según sea de exitoso el acceso a la cámara se llamará a una función u otra. Si nos devuelve éxito se llama a una función con la que guardaremos el flujo y lo visualizaremos en un elemento vídeo de HTML5. Esta visualización que se produce en el par local no es necesaria pero se ha incluido para tenerlo como referencia a la hora de realizar los experimentos. Si devuelve error

mostramos en el terminal del navegador un mensaje del error ocurrido.

```
1  // Manejador de éxito
2  function handleUserMedia(stream){
3      localStream = stream;
4
5      if (window.URL){
6          localVideo.src = window.URL.createObjectURL(stream);
7      } else{
8          localVideo.src = stream;
9      }
10     //console.log('Adding local stream.');
```

```
11     // Envio un mensaje al servidor como ack de éxito al llamar
12     //getUserMedia()
13 }
14
15 // manejador de error
16 function handleUserMediaError(error){
17     console.log('getUserMedia error: ', error);
18 }
19
20 //Funcion getUserMedia
navigator.getUserMedia(constraints, handleUserMedia,
    handleUserMediaError);
```

Listado 4.13: getUserMedia.

El flujo que nos proporciona *getUserMedia* y el cuál guardamos en la variable *localStream* (línea 3 del listado 4.13) será el que enviemos al par remoto con la API *RTCPeerConnection*. Este punto lo veremos en la sección 4.3.2.

4.3. Conexión entre navegadores

Interconectar el ordenador remoto al drone a través del ordenador local es el segundo de los objetivos marcados. Para ello vamos a utilizar WebRTC por su comunicación entre pares sin necesidad de utilizar servidores intermedios.

Este objetivo se ha dividido en tres sub-objetivos como siguen:

- *Visualización en remoto de cámara a bordo*: La conexión tiene que ser capaz de transportar el flujo audiovisual de la cámara del drone desde el navegador local hasta el navegador remoto.

- *Sensores de navegación.* Los datos obtenidos de los sensores de navegación del cuadricóptero como la brújula, el GPS, altímetro... deberán ser enviados al ordenador remoto donde se visualizarán.
- *Órdenes.* Desde el ordenador remoto deberán enviarse hacia el ordenador local las órdenes dadas para el aterrizaje, despegue, y comandos de movimiento.

Para desarrollar la conexión multimedia entre los navegadores con la tecnología WebRTC lo primero que se ha desarrollado ha sido el servidor de señalización.

4.3.1. Servidor de señalización

Las necesidades a cubrir en cuanto al servidor de señalización es que sea capaz de intercambiar los datos de red necesarios (Candidatos ICE) y de paquetes SDP. El intercambio debe hacerse con el protocolo de oferta/respuesta según lo establecido en la arquitectura JSEP explicada con anterioridad.

Se ha optado por desarrollar el servidor escrito sobre *Node.js*⁴. Las razones por las que hemos elegido este servidor es que está escrito en JavaScript, por lo que nos resulta muy útil al utilizarse el mismo lenguaje de programación que vamos a utilizar para el resto del proyecto. Por otro lado es un servidor muy liviano.

Para cumplir con la arquitectura JSEP vamos a utilizar la librería *Socket.io*⁵, la cuál nos facilita el desarrollo de aplicaciones con necesidades de conexión entre equipos a través de *Websockets*.

Como se puede ver en la figura 4.2, el servidor recibe cuatro tipos de paquetes diferentes. Cuando se conecta el par local, cuando se conecta el par remoto, intercambio de candidatos ICE e intercambio de SDP. Sabiendo que tanto los candidatos ICE como los paquetes SDP son objetos, decidimos crear el servidor aceptando tres tipos diferentes de paquetes: el inicial del par local, el inicial del par remoto, y mensajes genéricos que contendrían los objetos anteriormente mencionados. Así pues, esta es la forma del código que gestiona en el servidor el intercambio de paquetes para la señalización.

⁴<https://nodejs.org/>

⁵<http://socket.io>


```

1 io.sockets.on('connection', function (socket){
2
3   // Manejador de mensajes genericos 'message' (intercambios SDP y
4     Candidatos ICE)
5   socket.on('message', function (message) {
6     //log('Server --> got message: ', message);
7     // Si el que envia es Droner hay que mandarlo al remote
8     if (socket.id == dronerID) {
9       io.sockets.socket(newPeer).emit('message', message);
10    } else if (socket.id== newPeer) {
11      io.sockets.socket(dronerID).emit('message', message);
12    }
13  });
14
15  // manejador de mensjes 'create' enviados por Droner
16  socket.on('create', function () {
17    //log('Server --> Droner has sido conectado');
18    socket.join();
19    dronerID = socket.id;
20    socket.emit('created');
21  });
22
23
24  // Manejador de mensajes 'join remote' enviados por remote
25  socket.on('join remote', function () {
26    //log("Server --> Un 'remote' se ha unido.");
27
28    io.sockets.in().emit('join remote');
29    socket.join();
30    newPeer = socket.id;
31    socket.emit('joined');
32  });
33 });

```

Listado 4.14: Núcleo servidor de señalización

Una vez tenemos el servidor operativo vamos a explicar cómo se ha creado la conexión entre pares y cómo se ha usado esta conexión para transmitir el vídeo de la cámara y los datos de los sensores del dron hasta el par remoto.

4.3.2. RTCPeerConnection: Transmisión de vídeo

La conexión entre pares se crea en el momento en el que el par local recibe del servidor de señalización un mensaje indicando que el par remoto se ha conectado y está preparado para establecer la conexión. En ese momento se ejecuta una función

que configura los parámetros necesarios y crea la comunicación.

Al igual que en *getUserMedia* debemos configurar las variables necesarias para que sean compatibles con todos los navegadores. Para establecer la conexión necesitamos tres variables distintas:

```

1 RTCPeerConnection = window.RTCPeerConnection || window.
  mozRTCPeerConnection ||
2     window.webkitRTCPeerConnection || window.
  msRTCPeerConnection;
3 RTCSessionDescription = window.RTCSessionDescription || window.
  mozRTCSessionDescription ||
4     window.webkitRTCSessionDescription || window.
  msRTCSessionDescription;
5 RTCIceCandidate = window.RTCIceCandidate || window.mozRTCIceCandidate
  ||
6     window.webkitRTCIceCandidate || window.
  msRTCIceCandidate;
```

Listado 4.15: Variables WebRTC

- *RTCPeerConnection*: Esta variable es la encargada de crear y mantener la conexión entre pares. Esta variable necesitará de las otras dos para crear la conexión.
- *RTCSessionDescription*: Crea los SDP locales y se encarga de gestionar los SDP recibidos.
- *RTCIceCandidate*: Encargada de descubrir los Candidatos ICE y de gestionar los recibidos del otro par.

También debemos indicarle los servidores STUN y TURN a los que conectarse para averiguar los pares de IP y puerto para los Candidatos ICE:

```

1 var ICE_config = {
2   'iceServers': [
3     {
4       'urls': 'stun:stun.l.google.com:19302'
5     },
6     {
7       'urls': 'stun:23.21.150.121'
8     },
9     {
10      'urls': 'turn:192.158.29.39:3478?transport=udp',
11      'credential': 'JZE0Et2V3Qb0y27GRntt2u2PAYA=',
12      'username': '28224511:1379330808'
```

```

13     },
14     {
15         'urls': 'turn:192.158.29.39:3478?transport=tcp',
16         'credential': 'JZE0Et2V3Qb0y27GRntt2u2PAYA=',
17         'username': '28224511:1379330808'
18     }
19 ]
20 };

```

Listado 4.16: Servidores STUN y TURN

Una vez tenemos las variables configuradas, la forma en la que creamos la conexión WebRTC es la siguiente:

```

1
2 var PeerConnection = new RTCPeerConnection(ICE_config, pc_constraints
   );

```

Listado 4.17: RTCPeerConnection.

Esta función admite dos argumentos. El primero es la configuración ICE que acabamos de ver y el segundo es una variable con las restricciones de todas las funcionalidades y configuraciones que tiene *RTCPeerConnection*. En nuestro caso esa variable está vacía ya que la configuración por defecto es más que adecuada para nuestros intereses.

Para el intercambio de paquetes en la señalización es necesario crear los manejadores que *RTCPeerConnection* necesita para los SDP y para los Candidatos ICE. El par local tiene sus manejadores ya que es el encargado de llevar la batuta de la conexión, y el par remoto tiene otros manejadores diferentes.

Los SDP se manejan en el par local con el método *createOffer* que al ser llamado activa el proceso de crear la oferta SDP. Cada vez que crea uno nuevo salta un evento en la función que se le ha indicado, llamada *gotLocalDescription*. Esta función se encarga de ajustar el SDP local al SDP que se acaba de crear y de mandárselo al otro par a través del servidor de señalización. Si al crear un SDP se produce un error se llama a la función de error, la cual se encargará de notificarlo en la consola del navegador.

```

1 // Funcion de exito
2 function gotLocalDescription(sessionDescription){
3     PeerConnection.setLocalDescription(sessionDescription,
4         successLocalSDP, errorLocalSDP);
5     sendMessage(sessionDescription);
6 }

```

```

6 |
7 | // Funcion de error
8 | function onSignalingError(error){
9 |     console.log('Fallo al crear el SDP: ' + error);
10 | }
11 |
12 | // Metodo manejador de las SDP
13 | PeerConnection.createOffer(gotLocalDescription, onSignalingError);

```

Listado 4.18: Manejador de los SDP.

Esta oferta es recibida en el par remoto a través del servidor de señalización y se guarda como la oferta del otro par, ya que la necesitaremos para conocer las características del flujo audiovisual que nos llegará:

```

1 | PeerConnection.setRemoteDescription(new RTCPSessionDescription(
    message));

```

Listado 4.19: Estableciendo SDP del par remoto.

Al establecer el SDP remoto ocurre un evento del método *createAnswer* y crea una respuesta a la oferta recibida.

```

1 | PeerConnection.createAnswer(gotLocalDescription, onSignalingError);

```

Listado 4.20: Manejador de respuestas SDP

Las funciones de éxito y error *gotLocalDescription* y *onSignalingError* son las mismas que en el otro par, puedes encontrarlas en el listado 4.18.

Para manejar los Candidatos ICE utilizamos el método manejador *onIceCandidate*, el cuál llama a la función indicada, *handleIceCandidate()* en nuestro caso, en el momento en que se encuentra un Candidato ICE. Esta función se encarga de enviar el candidato a través del servidor de señalización. Ambos pares usan este mismo evento con la misma configuración en la detección y envío de candidatos ICE.

```

1 | function handleIceCandidate(event){
2 |     //console.log('handleIceCandidate event: ', event);
3 |     if (event.candidate) {
4 |         sendMessage({
5 |             type: 'candidate',
6 |             label: event.candidate.sdpMLineIndex,
7 |             id: event.candidate.sdpMid,
8 |             candidate: event.candidate.candidate});
9 |     } else {

```

```

10     console.log('End of candidates.');
```

```

11 }
12 // console.log('Local ICE candidate: \n' + event.candidate.
    candidate);
13
14 }
15
16 // Metodo manejador de candidatos ICE
17 PeerConnection.onicecandidate = handleIceCandidate; // Manejador ICE
    local (manda ICE local a remoto)
```

Listado 4.21: Manejador de los Candidatos ICE locales.

Los candidatos son recibidos en ambos pares a través del servidor de señalización. Se crea el candidato y se le añade como candidato remoto con el método *addIceCandidate*.

```

1 var candidate = new RTCIceCandidate({sdpMLineIndex:message.label,
    candidate:message.candidate});
2 PeerConnection.addIceCandidate(candidate);
```

Listado 4.22: Manejador de los Candidatos ICE remotos.

Uno de los puntos mas importantes para el proyecto que se tiene que encargar *RTC-PeerConnection* es transferir el flujo visual desde el par local al par remoto. WebRTC nos lo permite de una forma muy sencilla. En el par local, con un método al que le añadimos como argumento el flujo que hemos obtenido al llamar a la API *getUserMedia*:

```

1 PeerConnection.addStream(localStream);
```

Listado 4.23: Manejador del flujo audiovisual en el par local.

Y en el par remoto con un método manejador llamado *onaddstream*, que en el momento en el que salta el evento llama a la función *handleRemoteStreamAdded*, y ésta se encarga de configurar el flujo en una etiqueta vídeo de HTML5.

```

1 // Funcion manejadora
2 function handleRemoteStreamAdded(event) {
3
4     var remoteVideo = document.getElementById("droneVideo"); //
5
6     if (window.URL){
7         remoteVideo.src = window.URL.createObjectURL(event.stream);
8     } else{
9         remoteVideo.src = event.stream;
10    }
```

```
11     //console.log('Remote stream attached!!');
12     remoteStream = event.stream;
13 }
14
15 // Metodo manejador
16 PeerConnection.onaddstream = handleRemoteStreamAdded;
```

Listado 4.24: Manejador del flujo audiovisual en el par remoto.

Una vez creada y establecida la conexión *RTCPeerConnection* la usaremos para crear la conexión de datos *RTCDataChannel*, para transportar toda la información necesaria en ambas direcciones.

4.3.3. RTCDataChannel

RTCDataChannel nos permite transferir cualquier tipo de dato u objeto entre los pares a través de la comunicación creada por *RTCPeerConnection*. Hemos usado esta API para transferir tanto los datos obtenidos de los sensores del dron (del navegador local al navegador remoto) como las órdenes de movimiento dadas (del navegador remoto al navegador local).

Para crear esta conexión basta con llamar al método *createDataChannel()* de *RTCPeerConnection* en el par local ya que es quien arranca la conexión. Este método admite dos argumentos, el primero es el nombre que le damos a esta conexión y el segundo es un objeto en el que se especifica la configuración de las propiedades disponibles. La única propiedad que se ha configurado distinta de las que vienen por defecto es la entrega ordenada de paquetes, estableciéndolo en falso, ya que de esta manera introduce menos retraso a la comunicación, eliminando cuellos de botella ya que el medio en el que se transfieren los paquetes no es ruidoso ni tiene condiciones extremas para que se produzcan demasiadas pérdidas de paquetes.

```
1 // Creacion de la comunicacion dataChannel
2 dataChannel = PeerConnection.createDataChannel("droneDataChannel", {
3     ordered: false});
4
5 // Metodo de error para la creacion del dataChannel
6 dataChannel.onerror = function (error) {
7     console.log("Data Channel Error:", error);
8 }
```

Listado 4.25: Establecimiento conexión RTCDataChannel en el par local.

Si en el momento de crear la conexión ocurre un error salta un evento en el método *onerror* que nos comunica a través de la terminal el error concreto que se ha producido.

Para el control de la comunicación *RTCDataChannel* utiliza tres métodos para gestionar la conexión. Por un lado los métodos de inicio y final de la conexión creada, y el método que maneja los mensajes recibidos.

```
1 function handleReceiveChannelStateChange() {
2   var readyState = dataChannel.readyState;
3   if (readyState == closed) {
4     isChannelRunning = false;
5     clearInterval(intervalo); //Paramos el intervalo de actualizacion
6                               con el drone si el canal se cierra
7   }
8 }
9 dataChannel.onopen = handleReceiveChannelStateChange;
10 dataChannel.onclose = handleReceiveChannelStateChange;
11 dataChannel.onmessage = handleMessage;
12 };
```

Listado 4.26: Manejadores de *RTCdataChannel*.

La función *handleMessage* es la encargada de manejar los mensajes recibidos, pero hablaremos en profundidad de ella más adelante.

En el par remoto para establecer la conexión *RTCDataChannel* hemos configurado el método *ondatachannel*. Este método se encarga de unirse a la comunicación creada por el par local con las mismas propiedades y configuraciones que se le han indicado. Cuando se activa este método llama a la función *gotReceiveChannel* que es la encargada de manejar la conexión. Esta función se encarga de configurar la conexión en la variable local, y de establecer en los métodos de apertura y cierre de la comunicación las funciones manejadoras. Las función manejadora *handleSendChannelStateChange* es la misma que la mostrada en el listado 4.26.

```
1 function gotReceiveChannel(event) {
2   dataChannel = event.channel; // Establece el datachannel
3   dataChannel.onopen = handleSendChannelStateChange;
4   dataChannel.onclose = handleSendChannelStateChange;
5   dataChannel.onmessage = handleMessage;
6 }
7
8 // Evento manejador al recibir apertura comunicacion datachannel del
   par local.
```

```

9 | PeerConnection.ondatachannel = gotReceiveChannel;
10 | };

```

Listado 4.27: Establecimiento de RTCDataChannel en el par remoto.

Una vez establecida la comunicación de datos *RTCDataChannel* podemos enviar los datos a través del método *dataChannel.send()* y recibirlos con el método manejador *onmessage*. A continuación vamos a ver cómo hemos utilizado estos métodos para enviar y recibir los sensores del drone y las órdenes de movimiento.

Sensores de navegación

Los datos de los sensores los recogemos en el par local. Llamamos a una función del método *ArDrone*, explicado en la sección 4.2.1, que a su vez llama a las funciones que se conectan directamente con las interfaces ICE de *ardrone_server* para recoger los datos. Esta llamada la hacemos de manera periódica con una función *setinterval()* de JavaScript.

```

1 | intervalo = setInterval(arDrone.updateAndSend, 15);

```

Listado 4.28: Intervalo actualización y envío de datos de los sensores.

Después de actualizar los datos, esta función a su vez llama a la función *sendNavigationData(pose, navdata)* que es la encargada de enviar estos datos a través de la conexión *RTCDataChannel*. Esta función admite dos argumentos, el primero es el objeto *pose*, que contiene los datos recogidos de la interfaz *Pose3D*, y el segundo el objeto *Navdata*, con los datos de navegación recogidos de la interfaz *Navdata*.

```

1 | function sendNavigationData(pose, navdata) {
2 |     var s = {pose:pose, navdata:navdata};
3 |     dataChannel.send(JSON.stringify(s));
4 |     //console.log("Send navigationData.");
5 | }
6 | };

```

Listado 4.29: Envío de los datos de los sensores en el par local.

Para enviar los datos al navegador remoto primero creamos un objeto JavaScript, en el que introducimos los objetos *Pose* y *Navdata*. Este objeto lo transformamos a un objeto JSON⁶ para enviarlo por el canal de datos *RTCDataChannel*.

⁶<http://www.json.org>

Los datos llegan al par remoto en forma de evento a la función *handleMessage()*.

```
1 function handleMessage(event) {  
2     var data = JSON.parse(event.data);  
3     pose = data.pose;  
4     navdata = data.navdata;  
5 }
```

Listado 4.30: Manejo de los datos de los sensores en el par remoto.

Esta función se encarga de reconvertir el objeto JSON recibido a objeto JavaScript, y posteriormente guardamos los objetos *pose* y *navdatas* para su posterior representación en pantalla. Estos datos los visualizaremos en unos relojes de navegación, los cuáles estudiaremos con más detalle en la sección 4.4.3.

Órdenes de movimiento

El par remoto es el encargado de transferir las órdenes dadas por el usuario al par local, y éste a su vez al drone. El usuario tiene distintas maneras de introducir o hacer saber al equipo las órdenes que quiere dar. Una de ellas es a través de unos *joysticks* multidispositivo situados en las esquinas inferiores de la pantalla, y la otra mediante un mando conectado por USB al ordenador.

En la sección 4.4.2 se analizarán en profundidad ambos, mientras que en esta sección nos centraremos en el intercambio de los datos.

La forma en la que hemos diseñado la interfaz hace que tengamos dos joysticks, uno de ellos controla la dirección y velocidad del drone, y el otro controla la altitud y el giro del drone. A parte de esto tenemos otro tipo de órdenes, las llamadas especiales, que son la orden de despegue y la orden de aterrizaje. Por este motivo se han creado tres tipos de funciones de envío:

```
1 function enviarOrden(d){  
2     //console.log(d);  
3     var data = {orden:d};  
4     dataChannel.send(JSON.stringify(data));  
5 }  
6  
7 function sendCMDVel(x, y){  
8     var data = {x:x,y:y};  
9     //console.log(data);  
10    dataChannel.send(JSON.stringify(data));
```

```

11 }
12
13 function sendAltYaw(alt, yaw){
14     //console.log(d);
15     var data = {alt:alt, yaw:yaw};
16     dataChannel.send(JSON.stringify(data));
17 }

```

Listado 4.31: Funciones de envío de órdenes en el par remoto.

La función *enviarOrden()* es llamada al pulsar un botón de despegue o aterrizaje que implementaremos en la interfaz de usuario. En esta función creamos un objeto JavaScript con la orden, le reconvertimos a un objeto JSON y se envía a través del canal RTCDataChannel con *dataChannel.send(JSON.stringify(data))*;

sendCMDVel() es llamada cuándo el usuario interactúa con el *joystick* situado en la parte inferior izquierda de la pantalla. Esta función admite como argumento el valor de las componentes espaciales *x* e *y*. Creamos un objeto JavaScript y lo enviamos de la misma forma que el resto de datos que enviamos a través del canal RTCDataChannel.

Con el segundo *joystick* situado en la esquina inferior derecha controlamos la altitud y el giro del dron. La función *sendAltYaw()* también admite dos argumentos, la altura (o componente espacial *z*) y el giro del dron. Enviamos de la misma manera, creando una variable JavaScript con ellas, reconvertimos a objeto JSON y enviamos *dataChannel.send()*.

Todos estos datos los recibimos en el par local con la función *handleMessage()*. En esta función reconvertimos el objeto JSON a objeto JavaScript y llamamos a la función manejadora *handleReceiveData()* a la que le enviamos como argumento el objeto JavaScript.

```

1  // Manejador de datos recibidos
2  function handleReceiveData(data) {
3      if ("orden" in data) {
4          if (data.orden == "takeoff") {
5              arDrone.takeoff();
6          } else if (data.orden == "land") {
7              arDrone.land();
8          } else {
9              console.log("Orden no valida. ");
10         }
11     } else if ("x" in data) {
12         arDrone.setXYValues(data.x, data.y);
13     } else if ("alt" in data) {

```

```
14     arDrone.setAltYaw(data.alt, data.yaw);
15 } else {
16     console.log("Dato invalido. ");
17 }
18 }
19
20 // Recibimos los paquetes del datachannel y los enviamos al manejador
    de datos
21 function handleMessage(event) {
22     //console.log('Received message: ' + event.data);
23     var data = JSON.parse(event.data);
24     handleReceiveData(data);
25 }
```

Listado 4.32: Manejo de las órdenes recibidas en el par local.

La función manejadora se encarga de discernir qué tipo de dato ha llegado y llamar a la función correspondiente del módulo ArDrone.

En este punto tenemos toda la comunicación de datos entre pares terminada, por lo que procedemos a explicar el desarrollo de la interfaz web de usuario.

4.4. Interfaz web de usuario en cliente remoto

El diseño de la interfaz se ha diseñado con tecnologías web de ultima generación que nos permiten teleoperar el drone de una forma fácil, sencilla e intuitiva.

Esta interfaz está compuesta principalmente por cuatro ingredientes: la visualización del vídeo recibido del navegador remoto a pantalla completa, unos *joysticks* virtuales para el manejo del drone complementados con un mando físico, unos relojes de navegación que nos representarán los datos recibidos de los sensores del drone y una visualización espacial en 3D del drone.

4.4.1. Visualización del vídeo

Para teleoperar el drone de una manera efectiva necesitamos poder visualizar el vídeo que recibimos de la cámara de la mejor manera posible. Hemos optado por visualizarlo a pantalla completa en el navegador, por debajo del resto de elementos que tendremos en la pantalla. Este tipo de representación es especialmente útil cuándo se

ejecuta en un teléfono móvil o en una tableta.

Para situar el vídeo a pantalla completa primero necesitamos definir en la hoja de estilos o *CSS* el cuerpo de la página con margen cero, y posteriormente situar el vídeo en el centro de la pantalla y con tamaño 100 %. Para el tiempo que transcurre entre que se inicia el período de señalización hasta que recibimos el vídeo se establece una imagen de fondo

```
1 body{
2     background: url(background.jpg) no-repeat;
3     background-size: 100% 100%;
4 }
5
6 video#droneVideo {
7     position: fixed;
8     top: 50%;
9     left: 50%;
10    min-width: 100%;
11    min-height: 100%;
12    width: auto;
13    height: auto;
14    z-index: -100;
15    -webkit-transform: translateX(-50%) translateY(-50%);
16    transform: translateX(-50%) translateY(-50%);
17    background: url(background.jpg) no-repeat;
18    background-size: cover;
19 }
```

Listado 4.33: Vídeo a pantalla completa.

En la figura 4.7 se puede observar el vídeo procedente de la cámara mostrándose de fondo por debajo del resto de elementos de la interfaz.

4.4.2. Joysticks y Gamepad

Para que el usuario controle el dron hemos optado por desarrollar dos opciones. La primera opción es a través de unos *joysticks* situados en la pantalla, los cuáles son multidispositivo, ya que se pueden usar en un ordenador con el puntero del ratón y en dispositivos táctiles con los dedos. Por otro lado, también puede ser controlado con un mando USB conectado en el ordenador.

Joysticks virtuales

Los joystick han sido desarrollados como un canvas de HTML5 en el que dibujamos un círculo y un aro exterior.

```
1 function dibujarAro(x, y, r) {  
2     ctx.beginPath();  
3     ctx.arc(x, y, r, 0, 2 * Math.PI, true);  
4     ctx.lineWidth = 7;  
5     ctx.strokeStyle = "rgb(87,125,25)";  
6     ctx.stroke();  
7 }  
8  
9 function dibujarCirculo(x, y, r) {  
10     ctx.beginPath();  
11     ctx.fillStyle = "rgb(75, 144, 176)";  
12     ctx.arc(x, y, r, 0, 2 * Math.PI, true);  
13     ctx.fill();  
14 }
```

Listado 4.34: Aro y círculo del joystick.

El aro exterior es un elemento fijo, es decir no se mueve y lo utilizamos como "tope" para el círculo interior. El círculo es móvil según los movimientos del usuario. En la figura 4.3 podemos ver la apariencia final de los dos *joysticks*.



Figura 4.3: Joysticks.

Para dotar a éste círculo de movimiento hemos utilizado los eventos de JavaScript *mousedown*, *mousemove*, *mouseup* y *mouseout*. El primer evento se ejecuta cuando el usuario hace *click* con el ratón. Dentro de este evento nos encargamos de detectar si ese *click* ha sido en el círculo que hemos dibujado. El segundo evento ocurre cuando, sin soltar el ratón, el usuario lo mueve. Si en el primer evento hemos detectado que ha *clickado* sobre el círculo, se re-dibuja por la zona donde está moviendo el ratón. Con el evento *mouseup*, al deshacer el click, re-dibujamos el círculo a su posición inicial. El evento *mouseout* detecta si el ratón se ha salido del aro y también devuelve el *joystick*

a su posición inicial.

```

1  canvas.addEventListener('mousedown', function(evt) {
2      var mousePos = oMousePos(canvas, evt);
3
4      if (ctx.isPointInPath(mousePos.x, mousePos.y)) {
5          arrastrar = true;
6      }
7  }, false);
8
9  // mousemove
10 canvas.addEventListener('mousemove', function(evt) {
11     var m = oMousePos(canvas, evt);
12     //ctx.beginPath();
13     //ctx.arc(X, Y, maxR, 0, 2 * Math.PI);
14     if (arrastrar) {
15         delta.x = m.x - p.X;
16         delta.y = m.y - p.Y;
17         var deltaR = Math.sqrt(delta.x * delta.x + delta.y *
18             delta.y);
19         var elR = Math.min(deltaR, maxR);
20         //console.log("DeltaR: " + deltaR + " elR: " + elR);
21         var angulo = Math.atan2(delta.y, delta.x);
22         //console.log(angulo); //
23
24         x = X + elR * Math.cos(angulo);
25         y = Y + elR * Math.sin(angulo);
26
27         ctx.clearRect(0, 0, cw, ch); // Clear and redraw the
28             joystick
29         dibujarAro(X, Y, RHoop);
30         dibujarCirculo(x, y, RCircle);
31     }
32 }, false);
33
34 // mouseup
35 canvas.addEventListener('mouseup', function() {
36
37     sendAltYaw(0, 0);
38
39     arrastrar = false;
40     ctx.clearRect(0, 0, cw, ch);
41     dibujarAro(X, Y, RHoop);
42     dibujarCirculo(X, Y, RCircle);
43 }, false);
44
45 // mouseout
46 canvas.addEventListener('mouseout', function() {
47     sendAltYaw(0, 0);

```

```

46
47     arrastrar = false;
48     ctx.clearRect(0, 0, cw, ch);
49     dibujarAro(X, Y, RHoop);
50     dibujarCirculo(X, Y, RCircle);
51 }, false);

```

Listado 4.35: Movimiento y control del joystick.

Dentro del evento *mousemove* también recogemos la posición en la que se encuentra el *joystick* y llamamos a la función correspondiente que se ocupa de enviar los datos a través del canal *RTCDataChannel*. En el *joystick* izquierdo se llama a la función *sendCMDVel()* y en el de la derecha a *sendAltYaw()*.

Para dotar a los joysticks de funcionalidades táctiles hemos desarrollado las mismas funciones que las anteriores con los eventos táctiles equivalentes: *touchstart*, *touchmove*, *touchend*, *touchup*.

Estos joysticks se han situado en las esquinas inferiores de la pantalla como se puede observar en la figura 4.7 para facilitar el manejo con dispositivos táctiles.

Gamepad

También se ha desarrollado la posibilidad de teleoperar el dron mediante un mando USB conectado al ordenador. La API *Gamepad* de JavaScript no está estabilizada, hay navegadores que no la soportan y no todos los mandos funcionan de la misma manera. El mando con el que se ha desarrollado es el *Xbox* de la empresa *Microsoft*, no se han realizado pruebas con mandos de otros fabricantes por lo que no se puede asegurar su correcto funcionamiento.

Para la detección del mando al ser conectado al puerto USB hemos usado los eventos que nos da la API *Gamepad*. Al conectar el mando se ejecuta el evento *gamepadconnected* que llama a la función manejadora *connecthandler*.

```

1 function connecthandler() {
2     var gp = navigator.getGamepads()[0];
3     console.log("Gamepad connected at index %d: %s. %d buttons, %d
4         axes.", gp.index, gp.id, gp.buttons.length, gp.axes.length);
5     updateGPInterval = setInterval(updateGamePad, 100);
6     $(".joystick").fadeOut(4000);
7 }

```

```

8 window.addEventListener("gamepadconnected", connecthandler);
9 window.addEventListener("gamepaddisconnected", disconnecthandler);

```

Listado 4.36: Detección del mando.

La función *connecthandler()* guarda en una variable el mando conectado y crea un intervalo que llama periódicamente a la función que detectará los movimientos de los botones. Por otro lado se encarga de ocultar los *joysticks* virtuales, los cuales volveremos a mostrar si se desconecta el mando.

Dentro de la función *updateGamePad()* detectamos el movimiento de los dos *joysticks* del mando, y la pulsación de los botones. Hemos configurado la pulsación de una tecla para el despegue y otra para el aterrizaje, y con los *joysticks* controlamos el vuelo de igual manera que con los *joysticks* virtuales.

```

1 function updateGamePad() {
2     var gp = navigator.getGamepads()[0];
3     if (!gp && isChrome) {
4         disconnecthandler();
5         chromeInterval = setInterval(scangamepad, 1000);
6     } else {
7         if (gp.buttons[0].pressed) {
8             enviarOrden("takeoff");
9         } else if (gp.buttons[1].pressed) {
10            enviarOrden("land");
11        }
12
13        if (!isChrome) {
14            var Y = applyDeadzone(gp.axes[0], 0.12);
15            var X = applyDeadzone(gp.axes[1], 0.12);
16            var Yaw = applyDeadzone(gp.axes[3], 0.12);
17            var Alt = applyDeadzone(gp.axes[4], 0.12);
18        } else {
19            var Y = applyDeadzone(gp.axes[0], 0.12);
20            var X = applyDeadzone(gp.axes[1], 0.12);
21            var Yaw = applyDeadzone(gp.axes[2], 0.12);
22            var Alt = applyDeadzone(gp.axes[3], 0.12);
23        }
24        sendCMDVel(-X*velocidad,Y*velocidad);// Change variables and
25        sendAltYaw(-Alt*velocidad, Yaw*velocidad);
26        send the command to the drone
27    }
}

```

Listado 4.37: Manajador de mando.

En esta función también recogemos los valores y llamamos a las funciones encargadas de enviar los datos a través del canal *RTCDataChannel*.

4.4.3. Relojes de navegación

Para representar los datos recibidos de los sensores del drone como la altitud, velocidad, brújula, etc nos hemos ayudado de un plugin⁷[22] jQuery desarrollado por Matton Sébastien⁸.

Hemos usado cuatro relojes distintos:

- Uno que nos indique la altitud del drone.
- El segundo nos representa el horizonte.
- El tercero es una brújula.
- El cuarto indica la inclinación del drone.

Para crear estos relojes, creamos una etiqueta `<span>` de HTML5 para cada reloj, y posteriormente con jQuery llamamos a las siguientes funciones:

```

1 var attitude = $.flightIndicator('#attitude', 'attitude', {roll:50,
  pitch:-20, size:s, showBox : false}); // Horizon
2 var heading = $.flightIndicator('#heading', 'heading', {heading:150,
  showBox:false, size:s}); // Compass
3 var altimeter = $.flightIndicator('#altimeter', 'altimeter', {
  showBox:false, size:s});
4 var turn_coordinator = $.flightIndicator('#turn_coordinator', '
  turn_coordinator', {turn:0, showBox:false, size:s});

```

Listado 4.38: Creación de los relojes.

Para actualizar los datos de estos relojes con los que llegan al par remoto directamente desde el drone se ha desarrollado una función a la que llamamos periódicamente con las variables *pose* y *navdata* como argumentos:

```

1 this.updatePanelControl = function(navdata, pose){
2   // calculate yaw, pitch, and roll
3   var yaw = getYaw(pose.q0, pose.q1, pose.q2, pose.q3);

```

⁷<http://sebmattton.github.io/flightindicators/>

⁸<https://github.com/sebmattton>

```

4   var pitch = getPitch(pose.q0, pose.q1, pose.q2, pose.q3);
5   var roll = getRoll(pose.q0, pose.q1, pose.q2, pose.q3);
6
7   attitude.setRoll(roll);
8   attitude.setPitch(-pitch);
9
10  // Altimeter update
11  altimeter.setAltitude(pose.z*100);
12  //altimeter.setPressure(1000+3*Math.sin(increment/50));
13
14  // TC update
15  turn_coordinator.setTurn(roll);
16
17  // Heading update
18  heading.setHeading(yaw);
19
20  // Cambiamos el ancho en el estilo del relleno de la bateria
    segun el nivel de bateria que nos manda el drone
21  battery.style.width = String(navdata.batteryPercent) + "%";
22  window.navdata = navdata;
23  console.log(navdata.state);
24 }

```

Listado 4.39: Actualización de los relojes.

La situación en la pantalla de estos relojes es de gran importancia. Para no ocultar la visión de la cámara y tenerlos visibles se ha decidido colocarlos por pares en las esquinas superiores de la pantalla. Además, se les ha dotado de la propiedad *draggable* de jQuery la cuál nos permite arrastrarlos por la pantalla y situarlos dónde más cómodo le sea al usuario.

```

1  $(function() {
2      $( "#attitude" ).draggable();
3      $( "#altimeter" ).draggable();
4      $( "#turn_coordinator" ).draggable();
5      $( "#heading" ).draggable();
6  });

```

Listado 4.40: Elementos arrastrables.

En la figura 4.4 podemos observar los cuatro relojes incorporados a la interfaz web de usuario. En la figura 4.7 se pueden también observar integrados con el resto de elementos que componen la interfaz.



Figura 4.4: Relojes de navegación.

4.4.4. Visualización espacial 3D del drone

Para días de niebla o condiciones extremas es imprescindible tener localizada la posición del drone para poder traerlo de vuelta. Utilizando los datos del GPS del drone hemos creado un marco 3D con la situación del drone en tiempo real. Para desarrollar este visualizador 3D nos hemos ayudado de la librería *Three.js*⁹ la cuál nos facilita el acceso y uso de WebGL. El mapa 3D es dibujado sobre una etiqueta `<canvas>`, y está compuesto principalmente por una escena, una cámara de observación en esa escena (desde ella se ve la escena), el suelo y el drone.

Lo primero que hemos hecho ha sido crear la escena y situar la cámara en la misma.

```

1 // Escena
2 var scene = new THREE.Scene();
3
4 //Cámara
5 var camera = new THREE.PerspectiveCamera( 45, mapa.width/mapa.height,
6     0.1, 1000 );
7 camera.up.set( 0, 0, 1 );
8 camera.position.set( 18,20, 18 );
9 camera.lookAt( new THREE.Vector3( 0, 0, 0 ) );

```

Listado 4.41: Escena y cámara en el mapa 3D.

La cámara se ha decidido que esté fija apuntando a la posición inicial del drone, ya que de esta manera es más sencillo conocer la posición del drone y devolverlo al origen si fuese necesario.

Posteriormente añadimos la superficie que representa el suelo con una rejilla de celdas de 1m^2 .

⁹<http://threejs.org>

```
1 // Suelo.
2 var planeGeometry = new THREE.PlaneGeometry( 40, 40);
3 var planeMaterial = new THREE.MeshLambertMaterial( {color: 0xcccccc}
4 );
5 var plane = new THREE.Mesh( planeGeometry, planeMaterial );
6 plane.position.x = 0;
7 plane.position.y = 0;
8 plane.position.z = 0;
9 plane.castShadow = false;
10 plane.receiveShadow = true;
11 scene.add( plane );
```

Listado 4.42: Superficie que representa el suelo del mapa.

Three.js admite archivos en formato COLLADA (*COLLABorative Design Activity*). Hemos reutilizado los archivos Collada que utiliza el simulador Gazebo para representar de igual manera el drone. Cargamos el objeto collada del cuadricóptero, establecemos su posición inicial en el mapa y su tamaño de la siguiente manera:

```
1 var loader = new THREE.ColladaLoader();
2
3 loader.load('./collada/quadrotor/quadrotor_4.dae', function ( collada
4 ) {
5     dae = collada.scene;
6
7     dae.position.x = 0;
8     dae.position.y = 0;
9     dae.position.z = 1;
10    dae.scale.x = dae.scale.y = dae.scale.z = 5;
11    dae.updateMatrix();
12
13    daemesh = dae.children[0].children[0];
14    daemesh.castShadow = true;
15    daemesh.receiveShadow = true;
16
17    scene.add( dae );
18    render();
19 });
```

Listado 4.43: Carga del objeto collada.

Desde aquí se hace también la primera llamada a la función *render()* que se encargara de actualizar la posición del drone en el mapa. Este re-pintado se produce con la función *requestAnimationFrame()*, la cuál admite como argumento la función a actualizar.

```
1 var render = function () {  
2   if (pose == undefined) {  
3     dae.position.x = 0;  
4     dae.position.y = 0;  
5     dae.position.z = 0;  
6   } else {  
7     dae.quaternion.set(pose.q1, pose.q2, pose.q3, pose.q0);  
8     dae.position.set(pose.x, pose.y, pose.z);  
9   }  
10   panelControl.updatePanelControl(navdata, pose); // update sensors  
11   watchers  
12 }  
13 renderer.render(scene, camera);  
14 requestAnimationFrame( render );  
15 };
```

Listado 4.44: Actualización del lienzo.

Para liberar al navegador de un *setInterval()*, se aprovecha el 'intervalo' que crea *requestAnimationFrame()* lo utilizándolo para llamar a la función *updatePanelControl()* que actualiza los relojes de navegación. En la figura 4.5 se puede ver el resultado final del mapa.

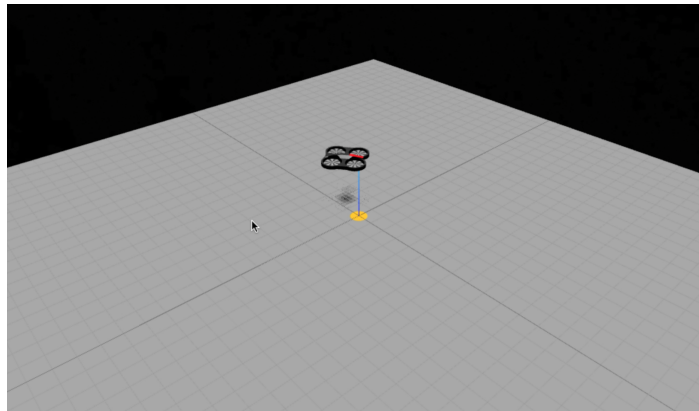
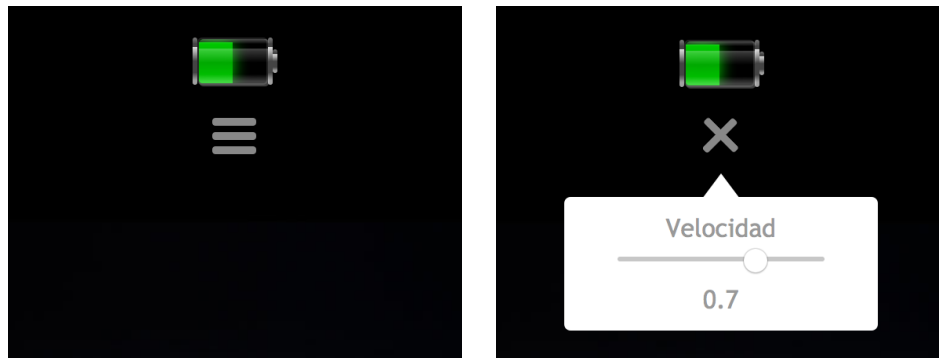


Figura 4.5: Mapa 3D con three.js

Además se le han añadido dos funcionalidades más a la interfaz web. Por un lado se ha añadido un indicador de nivel de batería, creado en HTML5 y CSS3. Esta batería está desarrollada por Jon Kantner¹⁰[23] y la hemos adaptado para nuestro proyecto. Por otro lado, se ha añadido un control de velocidad del drone, para facilitar su manejo si las condiciones son extremas o si se está realizando un vuelo en interiores. Este control está también desarrollado con HTML5 y CSS3 y fijan la velocidad máxima

¹⁰<http://codepen.io/jkantner/>

ordenable con los *joysticks* visuales y el mando. Estos dos elementos están situados en la parte central superior de la pantalla como se puede ver en la figura 4.6.



(a) Control de velocidad sin desplegar.

(b) Control de velocidad desplegado.

Figura 4.6: Control de batería y velocidad del cuadricoptero.

El resultado final de la interfaz web de usuario con todos los elementos integrados se puede ver en la figura 4.7.

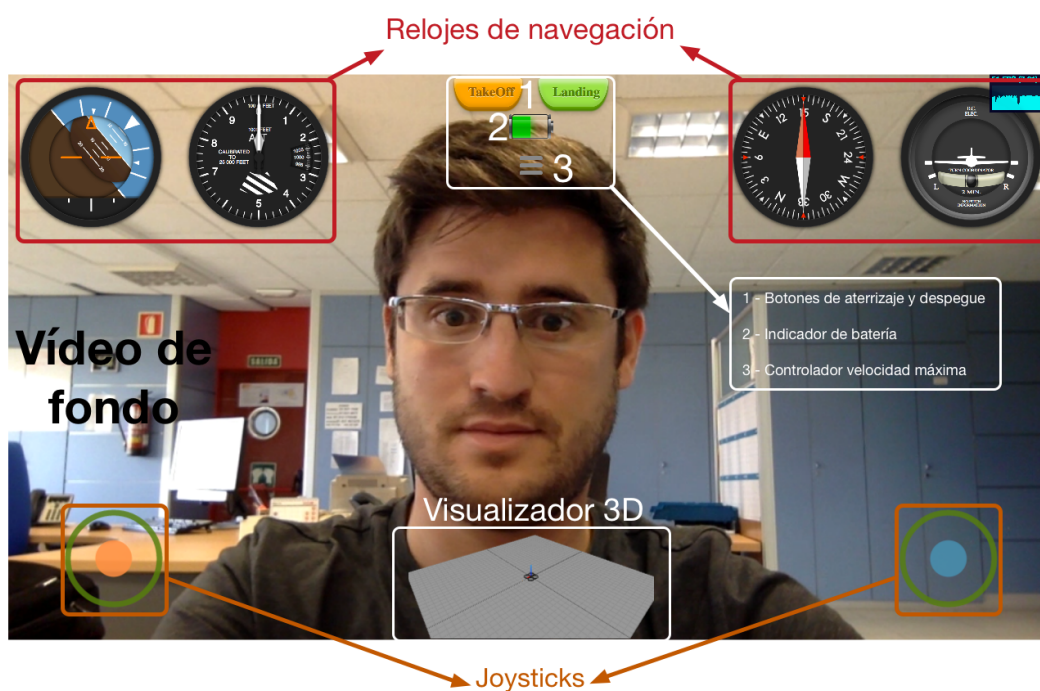


Figura 4.7: Interfaz web de usuario.

Capítulo 5

Experimentos

En capítulos anteriores se han expuesto los objetivos marcados para la realización del proyecto así como la solución desarrollada para cada uno de estos problemas y las herramientas utilizadas para los mismos. En este capítulo se muestran los experimentos que se han ido realizando según íbamos cumpliendo objetivos y que sirven como validación experimental de la solución programada.

5.1. Experimentos con simulador Gazebo

Una vez desarrollado el proyecto y satisfaciendo todos los objetivos marcados se han realizado pruebas sobre el drone simulado en Gazebo para comprobar sin ningún tipo de riesgos el funcionamiento del mismo.

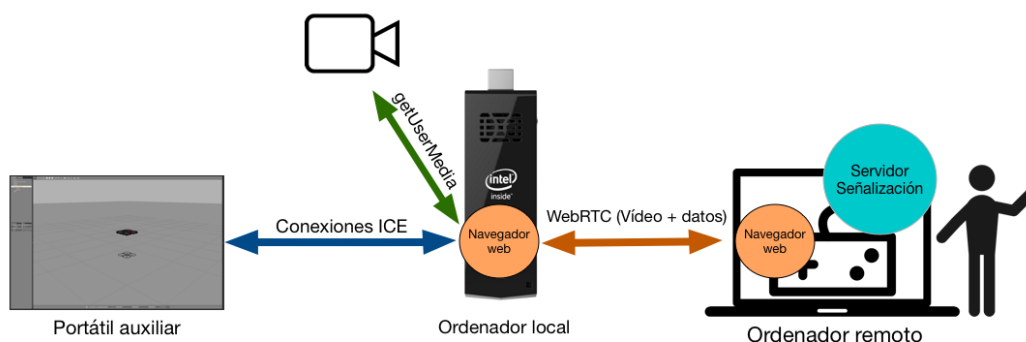


Figura 5.1: Experimento con el simulador Gazebo y un ordenador remoto.

Para realizar este experimento de la manera más realista al experimento final con

el drone real se ha utilizado un *Intel Compute Stick*¹ como par local. El navegador en ese *computer stick* es el encargado de conectarse a las interfaces ICE que nos proporciona el plugin ArDrone desarrollado por JdeRobot para el simulador Gazebo. Este ordenador está compuesto por un procesador Intel Atom que corre Ubuntu 14.04 LTS como sistema operativo. El simulador Gazebo se ha ejecutado en un portátil auxiliar y utilizaremos otro equipo portátil como par remoto desde el que teleoperaremos el drone además de ejecutar el servidor de señalización. La figura 5.1 muestra el esquema del montaje de este experimento.

Los resultados de este experimento han sido muy positivos, el control y manejo del drone han sido de manera fluida y sin realizar ningún movimiento extraño producido por algún tipo de mal funcionamiento del código. El manejo se ha realizado tanto con los *joysticks* virtuales como con el mando. Se puede ver un fotograma del vídeo² del experimento en la figura 5.2.

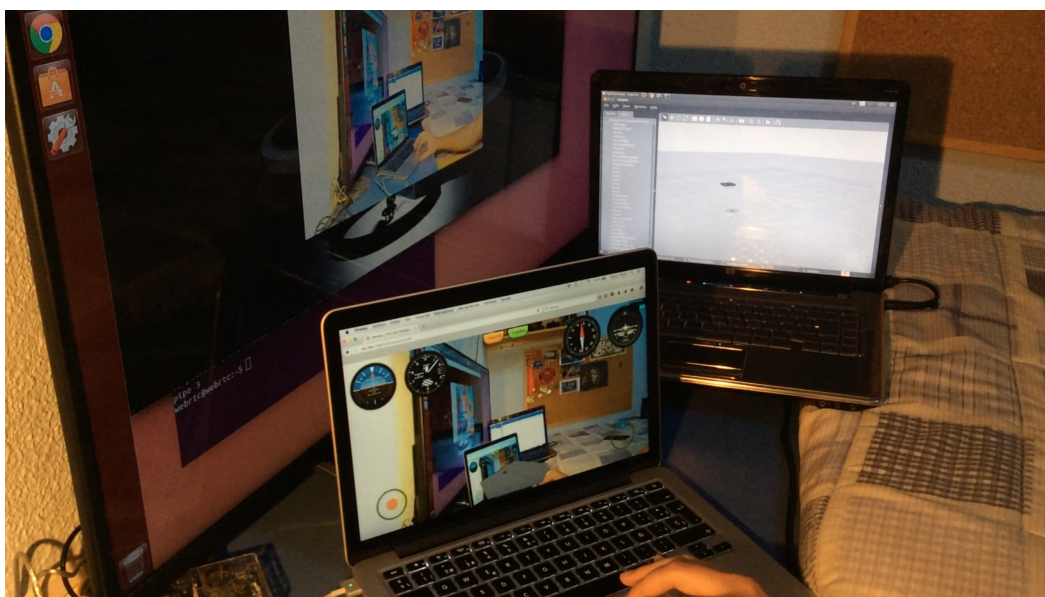


Figura 5.2: Experimento con el simulador Gazebo.

En la siguiente figura (5.3) se puede observar la secuencia de movimiento del drone durante el experimento.

¹<http://www.intel.com/content/www/us/en/compute-stick/compute-stick-product-brief.html>

²http://jderobot.org/Irodmar-tfg#WebRTC_on_a_drone_working_with_Gazebo5

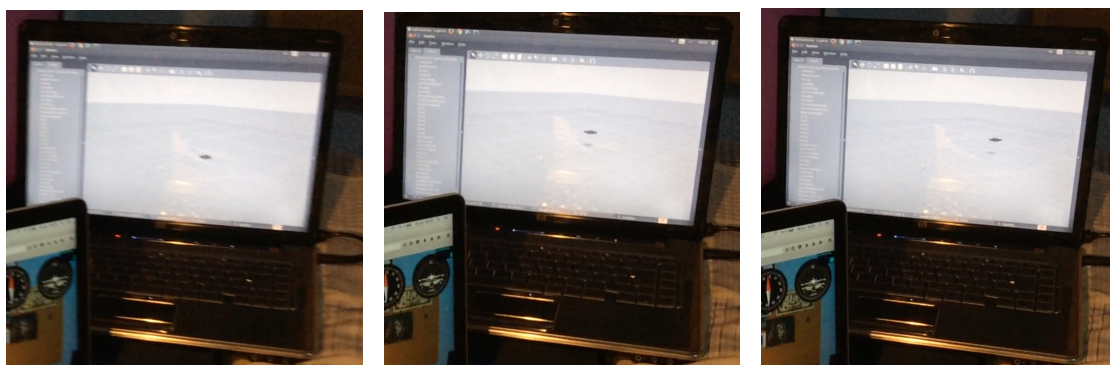


Figura 5.3: Secuencia de movimiento del drone dentro de Gazebo.

5.2. Vuelo del drone real

Las pruebas con el drone real se han dividido a su vez en dos. La primera consiste en realizar las pruebas sin colocar a bordo del drone ni el *computer stick* ni la cámara. Ambos se han usado para las pruebas pero para una primera aproximación se han realizado sin estar a bordo.

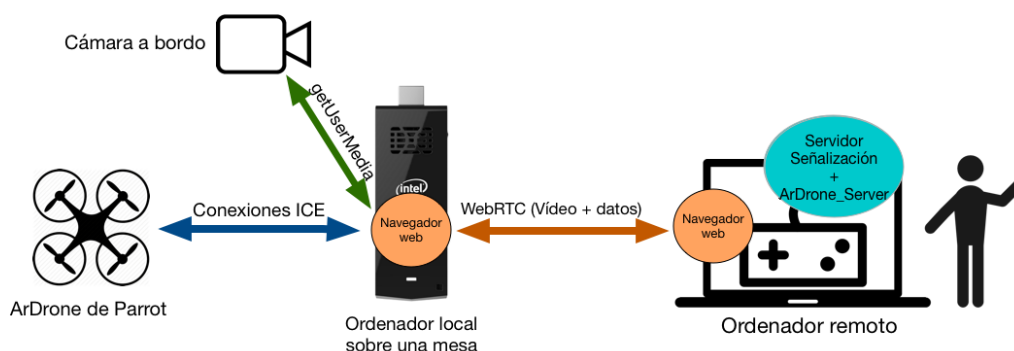


Figura 5.4: Esquema del experimento con el drone real.

Así pues la distribución de los equipos es como se muestra en la figura 5.4: computer stick como par local, que se conecta al drone real y accede a la cámara USB conectada al mismo, y ordenador portátil como par remoto desde el que se teleoperará el drone y a su vez está ejecutando el servidor de señalización y el `ardrone_server`.

La figura 5.5 muestra la realización de este experimento. En la mediawiki³[1] hay un vídeo completo del mismo. En la figura 5.6 se ha representado con tres imágenes

³http://jderobot.org/Irodmar-tfg/#First_Flying_with_Real_Drone

la secuencia de vuelo del drone durante el experimento.



Figura 5.5: Experimento uno con drone real.

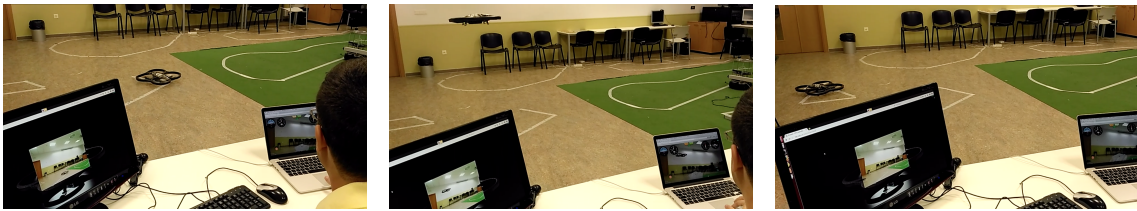


Figura 5.6: Secuencia de movimiento del experimento con el drone real.

Esta prueba también ha sido un éxito, por lo que nos marcamos el siguiente experimento con el *computer stick* y la cámara a bordo del drone. Este experimento la configuración es la misma que en el anterior, pero el manejo del drone con la aplicación será más realista ya que tenemos la cámara en primera persona.

La segunda prueba realizada ha consistido en colocar a bordo del drone la cámara, el *computer stick*, el cuál actuará como par local, estableciendo la conexión con el drone y accediendo a la cámara. El drone tiene una conexión USB de salida pero la potencia no es suficiente para hacer funcionar el *computer stick*. Por este motivo se ha tenido que colocar a bordo una pila adicional la cual hará las veces de fuente de alimentación. Por otro lado tenemos un ordenador el cuál actúa como par remoto desde el que teleoperaremos el drone y además se ha utilizado para ejecutar tanto el servidor de señalización como el servidor *ardrone_server*. La figura 5.7 muestra el esquema de la configuración del experimento.

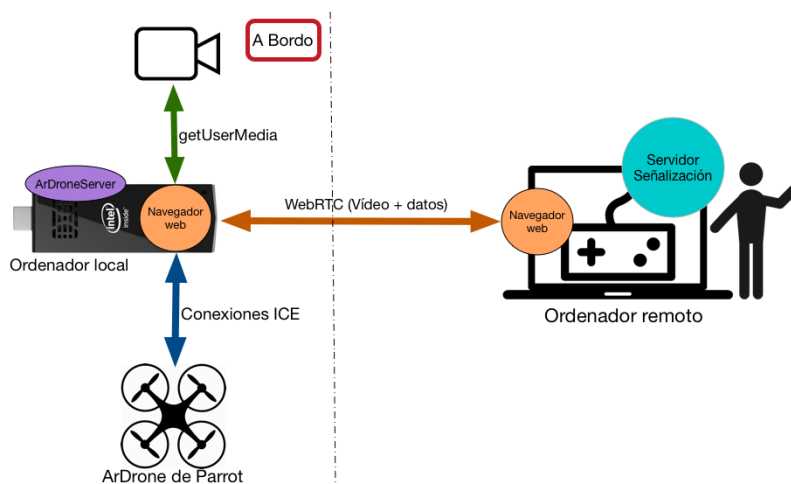


Figura 5.7: Esquema de la configuración del experimento.

En la figura 5.8 se muestra la configuración de todos los elementos que hemos colocado a bordo del drone. Como se puede observar también hemos incluido un hub USB ya que es necesario al disponer el *computer stick* de un único puerto USB y necesitar al menos dos, uno para la cámara y otro para el ratón en el momento que configuramos el navegador. Puede ver el montaje final en este vídeo⁴ de la mediawiki.

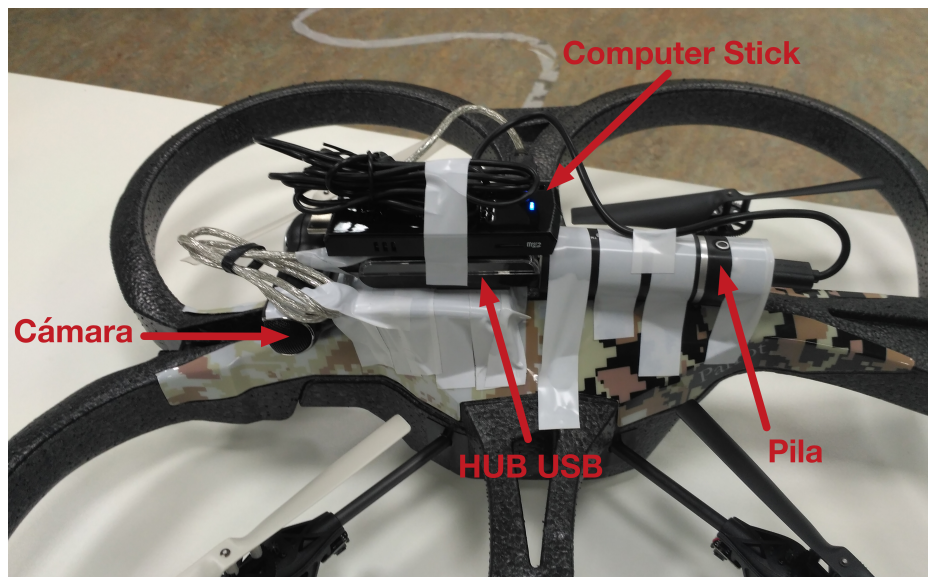


Figura 5.8: Configuración de los elementos a bordo del drone.

Este experimento ha sido fallido, únicamente por las capacidades de vuelo que nos ofrece el drone. Entre los dispositivos que hemos colocado a bordo superamos la carga máxima de peso que permite este cuadricóptero. En lo que al funcionamiento de la

⁴http://jderobot.org/Irodmar-tfg#Experiment_Setup

aplicación se refiere ha funcionado perfectamente ya que en el par remoto obteníamos las imágenes y datos de vuelo procedentes del drone, y al ejecutar la orden de despegue el cuadricóptero ha intentado levantar el vuelo sin conseguirlo.



Figura 5.9: ArDrone no levanta el vuelo por exceso de peso.

La figura 5.9 es un fotograma del vídeo⁵ que se grabó durante el experimento. En ella se ve al drone intentando levantar el vuelo. Para comprobar que el desarrollo funciona se optó por realizar otra prueba de vuelo en la que se cogía con las manos el drone y se le movía para observar que tanto la cámara a bordo cómo los relojes de vuelo variaban en el navegador remoto. La figura 5.10 muestra un fotograma del vídeo⁶ en el que se realizan estas comprobaciones.



Figura 5.10: Comprobación del funcionamiento de la aplicación.

⁵http://jderobot.org/Irodmar-tfg#Attemp_of_flying

⁶http://jderobot.org/Irodmar-tfg#Testing_the_development

5.3. Vuelos con multidispositivos

Como tercer y último experimento hemos probado a volar el drone utilizando dispositivos móviles. WebRTC tiene soporte para dispositivos móviles y los elementos de control los hemos desarrollado también para dispositivos táctiles, podemos utilizarlos como par remoto, par local o ambos. Al igual que el experimento anterior se ha realizado en dos pasos. Primero sin colocar el dispositivo a bordo del drone para confirmar su correcto funcionamiento y posteriormente repitiéndolo con el dispositivo a bordo. Realizar experimentos con dispositivos móviles implica la necesidad de la utilización de un ordenador de apoyo que será el que ejecute tanto el servidor de señalización para WebRTC como `ardrone_server`.

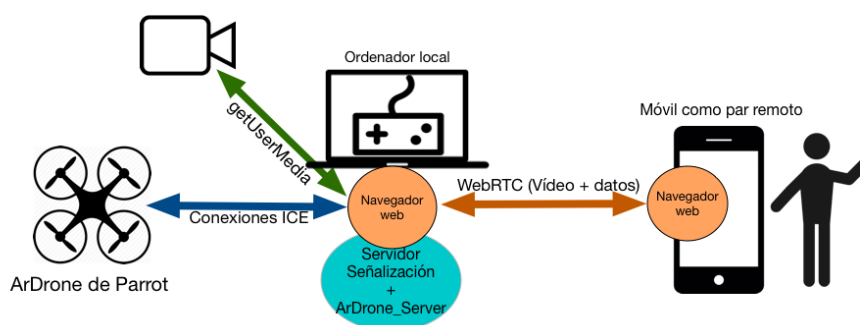


Figura 5.11: Esquema del experimento con móvil como par remoto.

Como primer experimento se ha utilizado un ordenador portátil como par local, encargado por un lado de estableciendo la conexión con el dron y de acceder a la cámara, en este caso la incorporada en el portátil, y por otro lado de correr los servidores de señalización y `ardrone_server`. Como par remoto para teleoperar el dron se ha utilizado un teléfono móvil. En la figura 5.11 se detalla el esquema de este experimento.

En la figura 5.12 se puede apreciar un instante del experimento cuyo vídeo se encuentra en la mediawiki⁷. En la figura 5.13 podemos ver la secuencia de movimiento del dron teleoperado con un dispositivo móvil.

⁷http://jderobot.org/Irodmar-tfg/#Flying_with_a_mobile_like_Remote_PC



Figura 5.12: Experimento con móvil como par remoto.

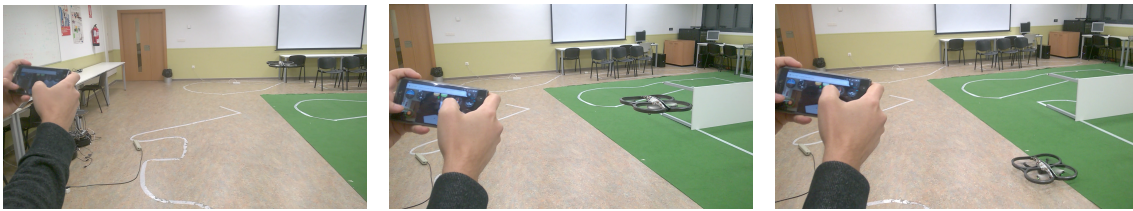


Figura 5.13: Secuencia de movimiento del experimento con dispositivo móvil como par remoto.

El segundo experimento (figura 5.14) es a la inversa, se utiliza un ordenador portátil como par remoto y un teléfono móvil como par local.

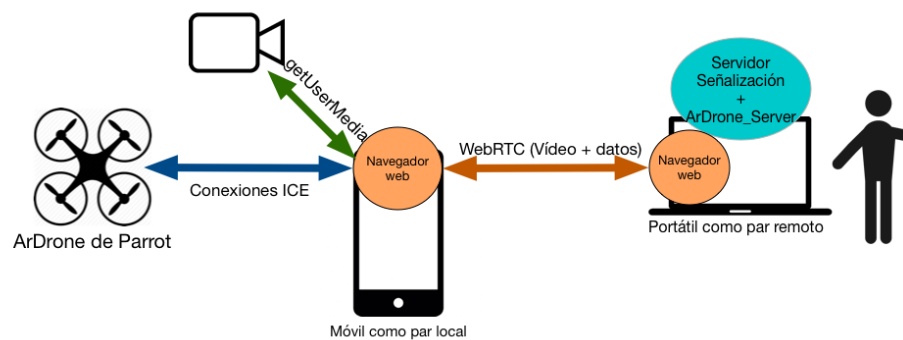


Figura 5.14: Esquema del experimento con móvil como par local.

En esta configuración el ordenador portátil se utiliza también para ejecutar los servidores de señalización y `ardrone_server`. El móvil se usa como par local, en-

cargándose de establecer la conexión con el drone. En este caso las imágenes que se envían son de una de las dos cámaras del móvil, la delantera o la trasera, pudiendo elegir cualquiera de las dos. En este experimento el móvil lo colocamos sobre la mesa mirando hacia la posición en la que se encuentra el drone, por lo que en la pantalla del ordenador remoto le veremos moviéndose. La figura 5.15 muestra un momento del experimento que está recogido en un vídeo en la mediawiki⁸. En la figura 5.16 se muestra una secuencia del vuelo realizado.

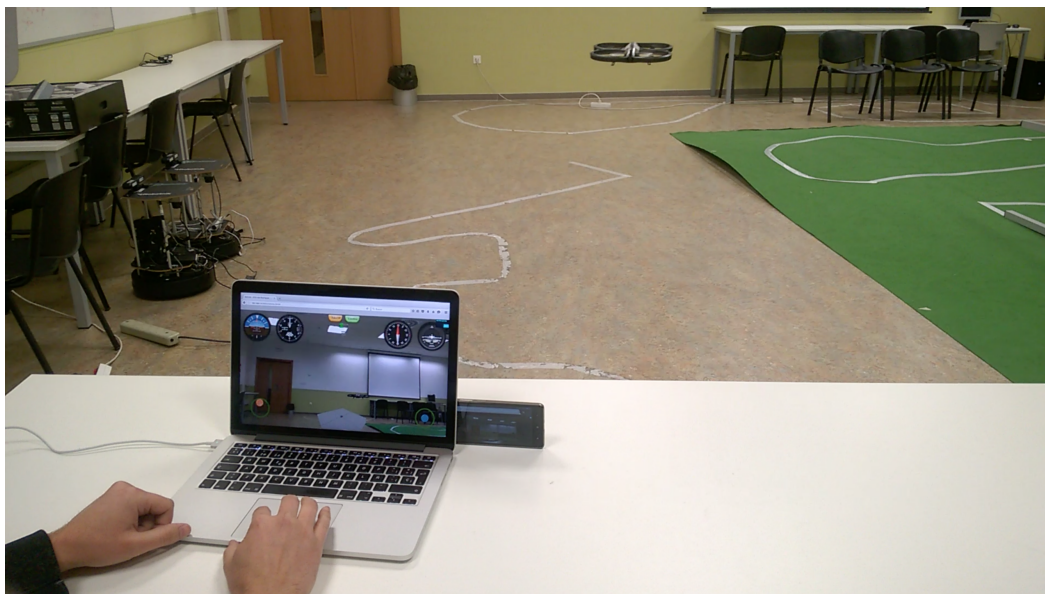


Figura 5.15: Experimento con móvil como par local.



Figura 5.16: Secuencia de movimiento del experimento con dispositivo móvil como par local.

Se ha desestimado la realización del experimento en el que situamos el móvil a bordo del drone debido a la dificultad de colocarlo en posición vertical para que la cámara apunte desde la parte delantera del drone. La única posición dónde podría colocarse es en un extremo del cuadricóptero lo que causaría una descompensación de cargas dando lugar a que el drone no pudiese estabilizar el vuelo.

⁸http://jderobot.org/Irodmar-tfg#Flying_with_a_mobile_like_Droner_PC

Capítulo 6

Conclusiones

En capítulos anteriores hemos descrito el contexto y el problema que abordamos en este proyecto, la solución propuesta junto con una serie de pruebas y experimentos que validan y software desarrollado. En este capítulo se exponen las conclusiones obtenidas y las posibles líneas por las que se puede continuar el trabajo.

6.1. Conclusiones

Bajo una mirada retrospectiva se puede observar que se han cumplido satisfactoriamente los objetivos generales que nos habíamos marcado. Hemos creado una aplicación web entre navegadores sin servidores intermedios que nos permite controlar, manejar y monitorizar un cuadricóptero desde un navegador, por ejemplo un teléfono móvil. Dentro de este objetivo nos marcamos tres subjetivos, los cuáles también hemos cumplido:

- Hemos creado una conexión local directamente con los sensores y actuadores del drone mediante un navegador web sin necesidad de servidores intermedios utilizando *getUserMedia* de WebRTC para la cámara y ICE con *WebSockets* para comunicar el navegador con el servidor JdeRobot para el drone (*ardrone_server*).
- Se ha desarrollado una conexión entre navegadores que transfieren en tiempo real y sin servidores intermedios la cámara y los datos necesarios para monitorar los sensores del drone y teleoperarlo. Para ello se ha utilizado *RTCPeerConnection* para el vídeo y *RTCDataChannel* para transmitir los datos de los sensores del drone desde el par local hacia el par remoto y en sentido opuesto las órdenes dadas por el usuario final.

- Se ha creado una interfaz web amigable que nos permite monitorar la cámara y los sensores del drone, así como teleoperarlo de una manera muy intuitiva y sencilla de usar. Esta interfaz consta de un vídeo de fondo a pantalla completa, unos relojes de navegación dónde se representan los datos recibidos de los sensores del drone, unos *joystick* virtuales para controlar el drone y un visor 3D.

A parte de estos subobjetivos se le han añadido unos extras, como poder teleoperarlo con un mando o que tanto la conexión local como remota, así como la interfaz sean multiplataforma y multidispositivo, lo que nos permite controlarlo desde un teléfono móvil o tableta, por ejemplo.

Todo lo desarrollado se ha validado experimentalmente tal y como se muestra en el capítulo 5.

Se puede encontrar tanto esta memoria, como el repositorio del código, vídeos, explicaciones, ejemplos y resultados obtenidos en la mediawiki oficial del proyecto¹[1].

6.2. Trabajos futuros

A todo proyecto hay que ponerle unos límites y este no es una excepción. Sirve como base y punto de partida para otros proyectos o trabajos con los que ampliar esta aplicación. Dentro de las líneas de desarrollo que se podrían seguir exponemos algunas de ellas:

Uno de las líneas más útiles de desarrollo podría ser incorporar diversas cámaras al drone y tener la posibilidad en el par remoto de cambiar entre ellas según nuestras necesidades.

Un paso más podría ser dotar al drone de autonomía, pudiendo indicarle desde el ordenador remoto unas coordenadas para que el drone se desplazase de una a otra como si de un circuito se tratase. Esto pasa por desarrollar un autopiloto dentro del cuadricóptero que le dote de capacidades de navegación autónoma.

Asimismo este proyecto podría integrarse en el repositorio oficial de JdeRobot[4] desde dónde se podrá seguir trabajando en estas líneas de desarrollo o cualquier otra según las necesidades que vayan surgiendo.

¹<http://jderobot.org/Irodmar-tfg>

Bibliografía

- [1] Mediawiki oficial del proyecto (WebRTC en un drone) <http://jderobot.org/Irodmar-tfg>
- [2] Repositorio oficial del proyecto (WebRTC en un Drone) <https://github.com/RoboticsURJC-students/2015-tfg-irodmar>
- [3] Proyecto JdeRobot <http://jderobot.org/>
- [4] Repositorio de JdeRobot <https://github.com/RoboticsURJC/JdeRobot>
- [5] Mediawiki de Alberto Martín (ArDroneServer) <http://jderobot.org/Amartinflorido-tfg>
- [6] Mediawiki Daniel Castellano (Surveillance 4.0) <http://jderobot.org/D.castellanob-pfc>
- [7] Mediawiki Edgar Barrero (Surveillance 5.1) <http://jderobot.org/Aerobeat-colab>
- [8] Mediawiki Aitor Martínez <http://jderobot.org/Aitormf-tfg>
- [9] Página oficial de WebRTC <https://webrtc.org>
- [10] W3C: Borrador de la norma WebRTC <http://www.w3.org/TR/webrtc/>
- [11] Real-Time Communication with WebRTC <http://shop.oreilly.com/product/0636920030911.do>
- [12] Capítulo 18 (WebRTC) del libro *High Performance Browser Networking* <http://chimera.labs.oreilly.com/books/1230000000545/ch18.html>
- [13] WebRTC-experiment <https://www.webrtc-experiment.com>
- [14] JSEP <https://rtcweb-wg.github.io/jsep/>
- [15] Wikipedia: JSEP <https://es.wikipedia.org/wiki/JSON>

- [16] Señalización en *Mozilla Foundation* https://developer.mozilla.org/es/docs/Web/API/WebRTC_API/Connectivity
- [17] HTML5Rocks.com: Señalización <http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/>
- [18] WebRTC-Experiment: Señalización <https://www.webrtc-experiment.com/docs/WebRTC-Signaling-Concepts.html>
- [19] Seguridad en WebRTC <https://rtcweb-wg.github.io/security-arch/>
- [20] Session Initiation Protocol https://es.wikipedia.org/wiki/Session_Initiation_Protocol
- [21] ORTC <http://ortc.org/wp-content/uploads/2015/10/ortc.html>
- [22] Relojes de navegación <http://sebmatton.github.io/flightindicators/>
- [23] Repositorio batería HTML5 y CSS3 <http://codepen.io/jkantner/pen/QybzKL>
- [24] Página oficial Gazebo <http://gazebosim.org/>
- [25] Página oficial ICE <http://www.zeroc.com/>
- [26] Ice 3.5.1 Documentation <https://doc.zeroc.com/display/Ice35/Home>
- [27] Uso de los compiladores de Slice <https://doc.zeroc.com/display/Ice36/Using+the+Slice+Compilers>
- [28] Página de Ice for Javascript <https://zeroc.com/labs/icejs/index.html>
- [29] Websockets en ICE <https://zeroc.com/labs/icejs/websocket.html>
- [30] Página de Three.js <http://threejs.org/>
- [31] Documentación de Three.js <http://mrdoob.github.io/three.js/docs/>
- [32] Curso de Three.js <http://stemkoski.github.io/Three.js/>
- [33] Página de JQuery <https://jquery.com/>