

Capítulo 1

Introducción

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 a distancia y en tiempo real desde un navegador web en un ordenador, tableta o teléfono.

El drone es un vehículo aéreo no tripulado al que podemos ubicar dentro de la robótica aérea. En las siguientes páginas se dará unas pinceladas sobre la robótica y su uso actual. También hablaremos sobre los sistemas actuales de control y manejo de drones, y para finalizar daremos una visión global sobre las tecnologías existentes dentro de las comunicaciones en tiempo real (RTC, Real Time Communications), que se pueden emplear para comunicar, por ejemplo, un drone con una estación en tierra.

En estos últimos años el desarrollo de vehículos no tripulados ha tenido un avance muy 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.

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

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

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

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

1.1.1. Clasificación y usos de 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 aterri-

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

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

- **Militares.** Blancos móviles aéreos, reconocimiento de terreno y combate, entre otras tareas.
- **Vigilancia.** Seguridad en hogares, vigilancia de autopistas, costas, etc.
- **Inspección y reparaciones.** Fotografiar torres eléctricas, oleoductos, presas, gaseoductos, molinos eólicos, puentes, plataformas petrolíferas, etc, con el objetivo de vigilar o buscar daños que deban repararse.
- **Filmación.** Grabación de 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 drone despegue, la suma de fuerzas provocadas por cada rotor debe superar su peso. Una vez en el aire, si la suma de fuerzas es igual al peso, el drone permanecerá en una altitud fija o cernido (*hovering*). Para aterrizar, o desplazarse hacia abajo, es necesario hacer que la fuerza resultante sea algo menor que la del peso.

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

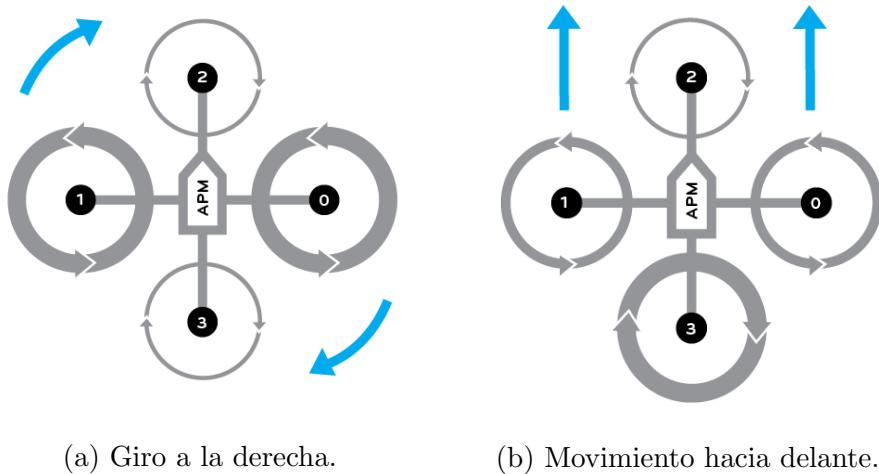


Figura 1.1: Distintas configuraciones de los motores del cuadricóptero para desplazarse.

no nula.

1.1.3. Aplicaciones

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 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, te detecta y sigue todos tus 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.

¹<https://www.lily.camera>

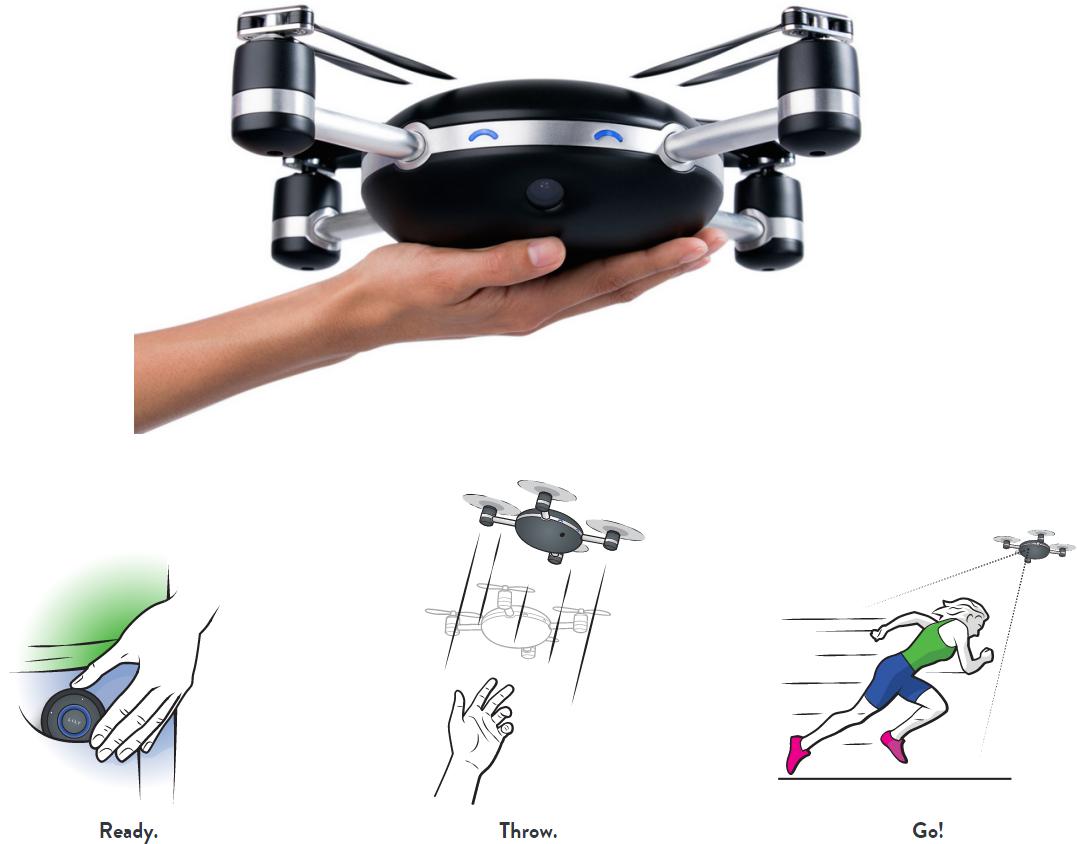


Figura 1.2: Lily (a) y funcionamiento en 3 pasos (b).

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.

Hemav, Agricultura de precisión.

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

²<http://hemav.com>

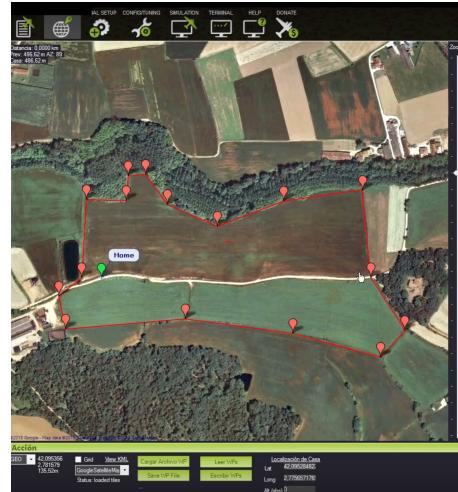


Figura 1.3: Drone de Endesa realizando labores de revisión.

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.



(a) Hemav.



(b) Finca acotada para el vuelo autónomo.

Entrega de paquetes con drones.

La entrega de paquetería usando vehículos aéreos no tripulados es el futuro, aunque ya muy presente 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.

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 Dron³ (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.

³trabajoscondrone.com



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 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.

1.1.4. Sistemas de control de drones

Los drones son vehículos aéreos no tripulados (*UAV, Unmanned Aerial Vehicle*), pero no autónomo, por lo que necesitan ser teleoperados desde tierra. Los sistemas actuales para ello se pueden dividir en dos grupos, los controlados mediante radiofrecuencia y los que usan sistemas alternativos.

⁴<http://friff.co>



Figura 1.7: Uso de un drone para grabación.

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

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

que es la que gestiona la 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 drone.

Empresas como DJI usa sistemas mixtos que consisten en teleoperar el drone 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 drone, así como tener otras funcionalidades como la grabación de vídeo, captura de imágenes y gestión de parámetros de vuelo como altura o velocidad máxima. En capítulos posteriores profundizaremos en este sistema implementado, ya que es el que usaremos para desarrollar el nuestro. 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.

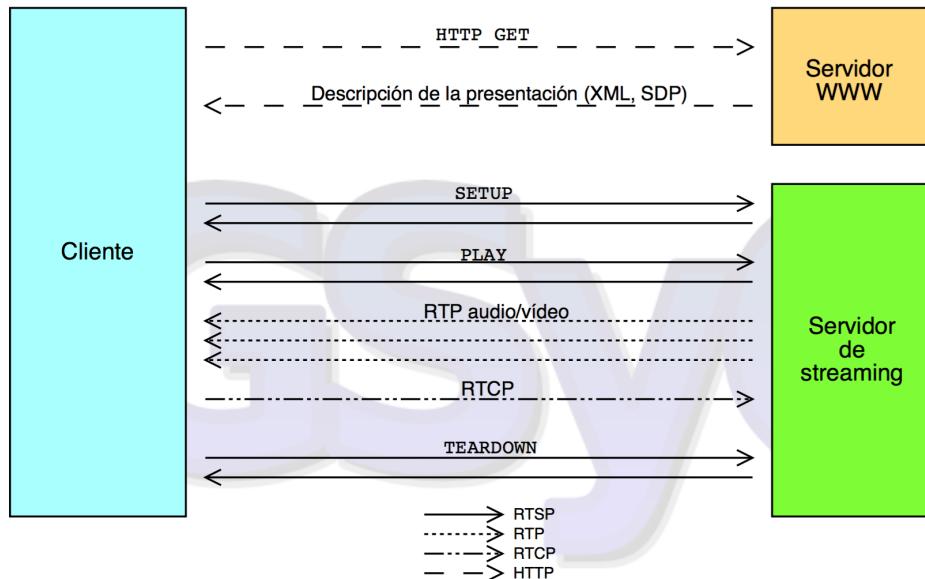


Figura 1.9: Ejemplo de conexión RTC, RTCP y RTSP

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.

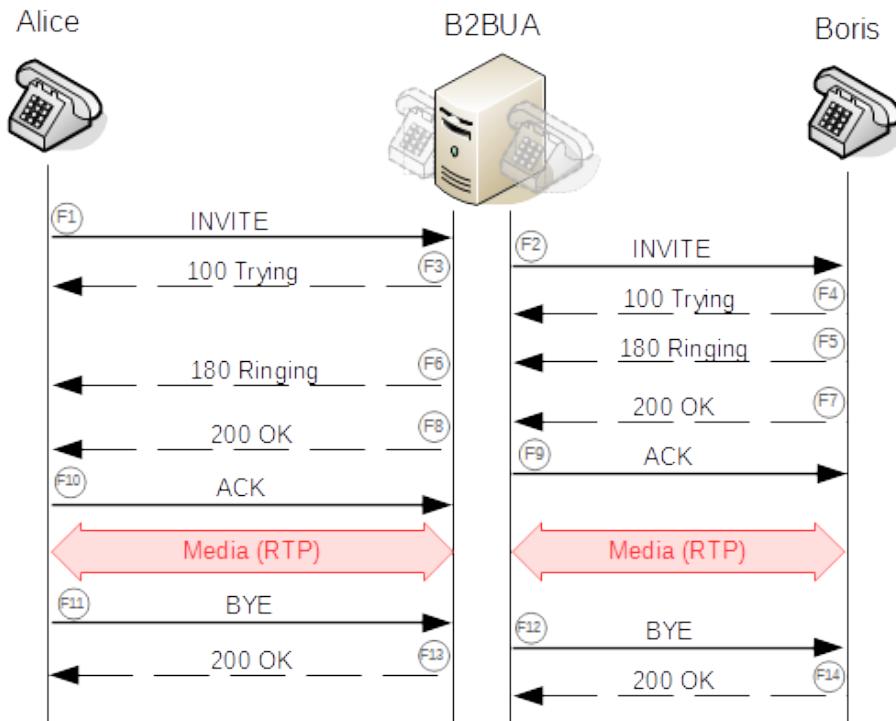


Figura 1.10: Ejemplo de conexión SIP

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*).

Wide Web Consortium).

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

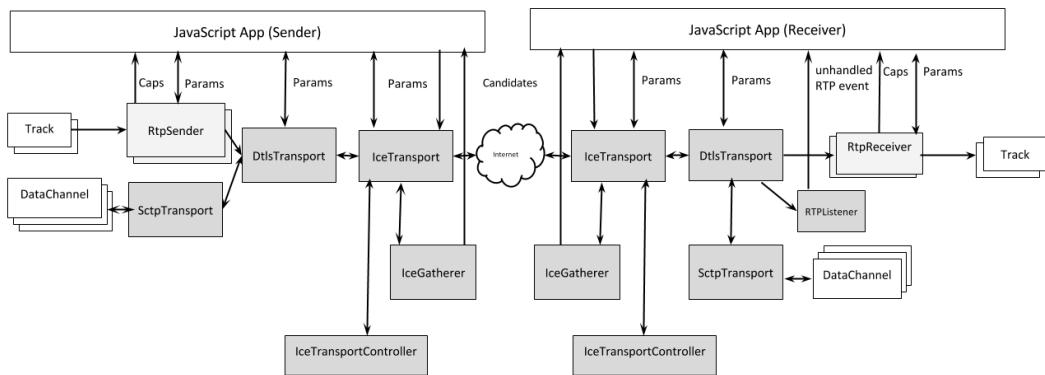


Figura 1.11: API de ORTC

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.



Figura 1.12: WebRTC

1.3. Antecedentes

Como base para el proyecto para desarrollar la aplicación web tenemos los siguientes proyectos, desarrollados también por alumnos de la URJC. Todos estos proyectos tienen en común que utilizan como base las herramientas que nos ofrece JdeRobot.

1.3.1. ArDroneServer

ArDroneServer es un envoltorio para la plataforma JdeRobot que pertenece a la SDK de ArDrone, desarrollado por Alberto Martín como parte de su Proyecto Fin de Carrera. Esta aplicación está inspirada en los paquetes ROS *ardrone_brown* y *ardrone_autonomy*. Implementa tres interfaces que nos permite 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.

1.3.2. Surveillance 4.0

Surveillance 4.0 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.

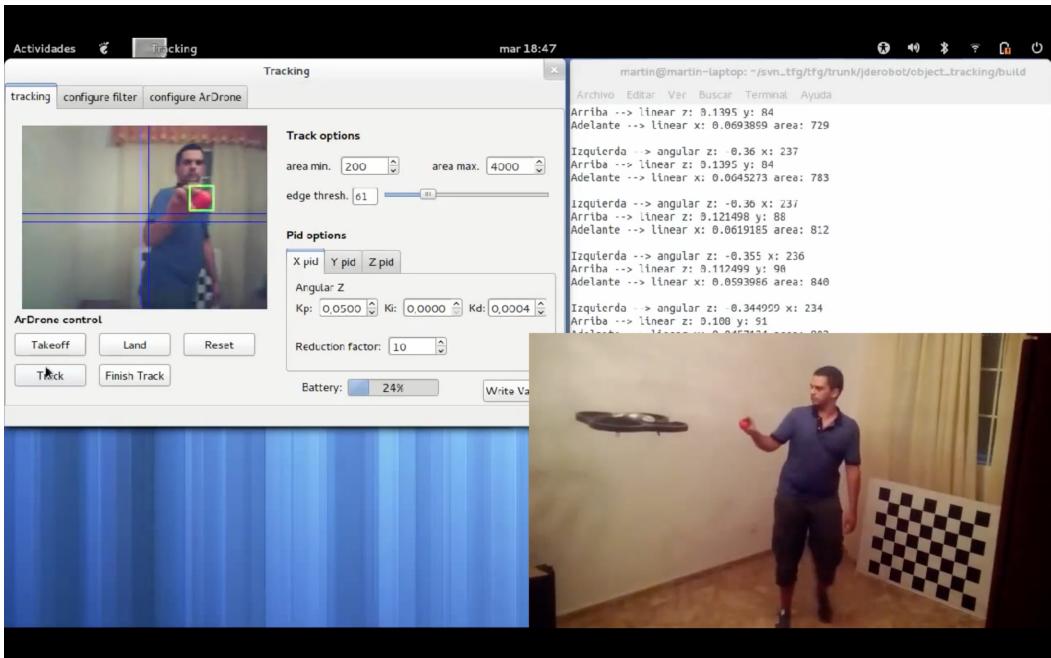


Figura 1.13: Seguimiento horizontal usando ardrone_server.

cuando era necesario. Además, esta versión incluía un streaming de vídeo utilizando el software de código abierto M-JPEG Streamer. En la figura 1.14 se puede ver la aplicación.

1.3.3. Surveillance 5.0

Otro ejemplo de conexión web con sensores y actuadores es Surveillance 5.1 desarrollado por Edgar Barrero como su Trabajo Fin de Grado. Esta aplicación obtenía un flujo de imágenes de una cámara web, un flujo de imágenes de profundidad de un sensor Kinect, además de datos de un sensor de humedad y de interaccionar con un actuador. La aplicación web se desarrolló en Ruby sobre Rails. En Surveillance 5.1, el servidor web se conectaba a los componentes de JdeRobot mediante sus interfaces ICE. La aplicación web refrescaba estos datos mediante peticiones AJAX. En la figura 1.15 se puede ver la aplicación.

1.3.4. Teleoperadores y visores Web en JdeRobot

Un ejemplo ya sin servidores intermedios, directo entre sensores, robots y navegadores web es esta plataforma desarrollada por Aitor Martínez como su Trabajo Fin de

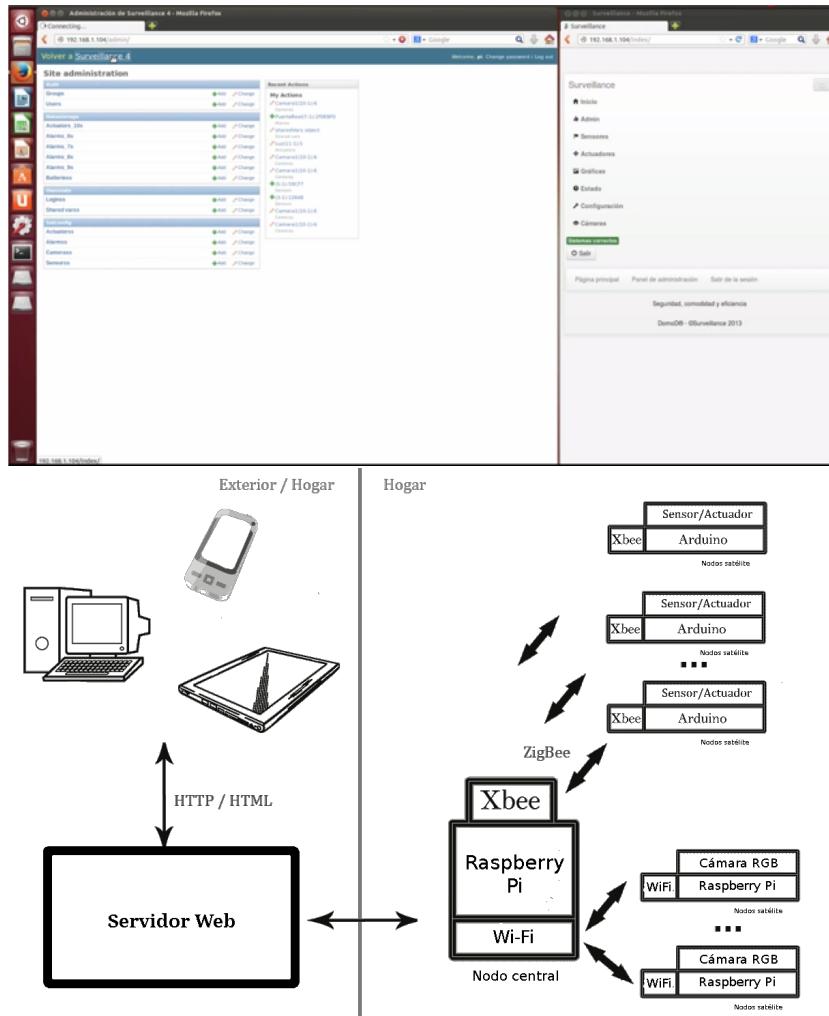


Figura 1.14: interfaz (a) y arquitectura (b).

Grado. Está compuesta por seis clientes Web: CameraViewJS, RGBDViewerJS, KobukiViewerJS, UavViewerJS, IntrorobKobukiJS e IntrorobUavJS, 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 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 Camerasyserver.
2. *RGBDViewerJS*: Cliente web similar a la herramienta RGBDViewer para visualizar datos de color y profundidad procedentes del servidor Openni1Server.



Figura 1.15: interfaz (a) y arquitectura (b).

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 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 cuál 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

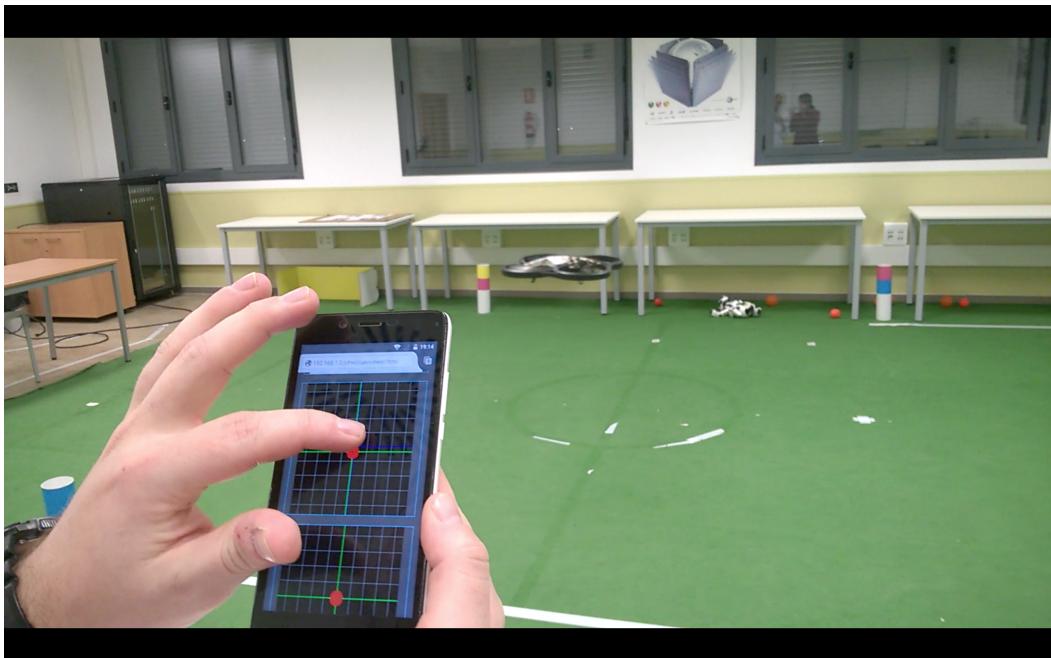


Figura 1.16: Cliente web UavViewer.js

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 este proyecto. En el capítulo tres se habla sobre las infraestructuras software en las que nos hemos ayudado para el desarrollo del proyecto. 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 sexto y 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 ultima generación. Este objetivo se ha desglosado en tres subobjetivos concretos que explicamos a continuación.

2.1.1. Conexión local

Como primer objetivo tenemos que desarrollar una conexión directa del navegador con el 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 ordenes 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.

2.1.2. 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 drone. 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 drone.

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

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.

2.1.3. Interfaz de usuario amigable

Uno de los requisitos 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 drone y la cámara para permitirnos conocer el estado de vuelo del drone en cada momento: altitud, inclinación, velocidad...

Esta interfaz deberá permitirnos el manejo del cuadricóptero de la forma más realista y similar posible a los sistemas de control de drones que hemos hablado en el capítulo 1.1.4.

Para que la satisfacción 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 del mismo. 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 subtareas 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.

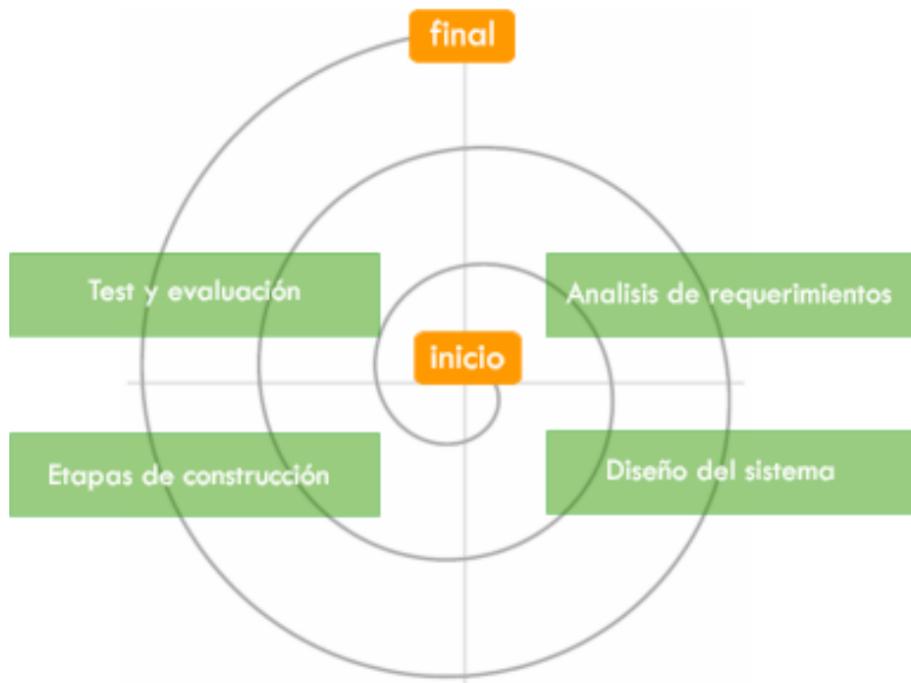


Figura 2.1: Esquema general del desarrollo del proyecto.

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

¹<http://jderobot.org/Irodmar-tfg>

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:

- **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 interconectara 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.

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

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

3.1. ArDrone de Parrot

El Parrot ARDrone es un drone que comercializa la marca Parrot el cuál puede ser teleoperado con una aplicación desde el dispositivos móviles. El drone crea una red WiFi, a la cuál conectas tu teléfono o tablet iOS ó Android y desde la aplicación, suministrada también por 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 simular este drone de la manera mas realista y realizar las pruebas de manera virtual, nos vamos a apoyar en el simulador robótico Gazebo.



Figura 3.1: ArDrone de Parrot

3.2. Simulador robótico Gazebo

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

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

Se ha usado la versión 5.1 del simulador, la cuál es última versión estable a la fecha

de comienzo y 100

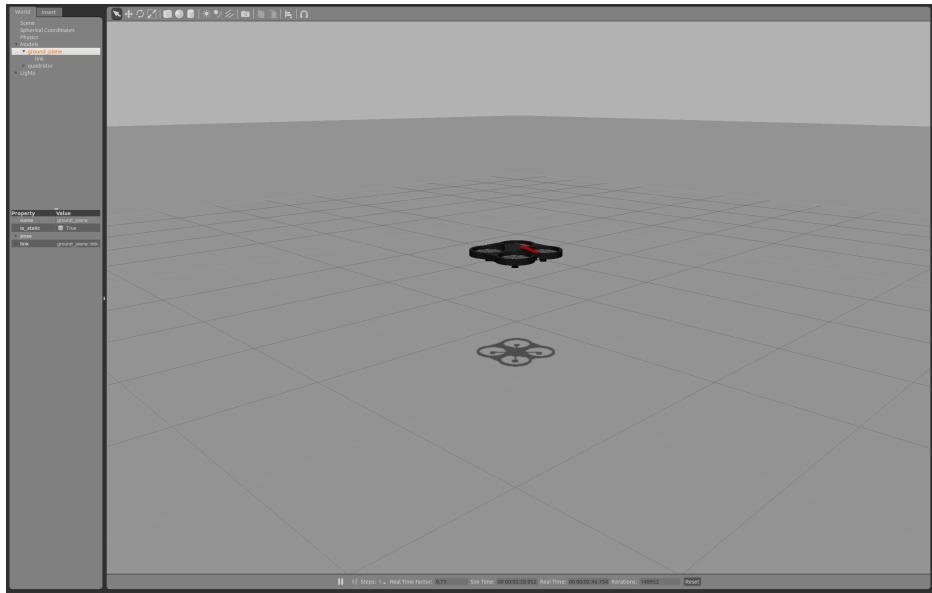


Figura 3.2: Simulador Gazebo

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

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

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

Incluye numerosas librerías y herramientas que podemos usar para desarrollar nuestra propia aplicación o para controlar nuestro propio robot. JdeRobot es software libre,

bajo licencia GPL y LGPL. También utiliza software de terceros como el simulador Gazebo, ArDroneServer ROS, GTK, OpenCV...

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

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

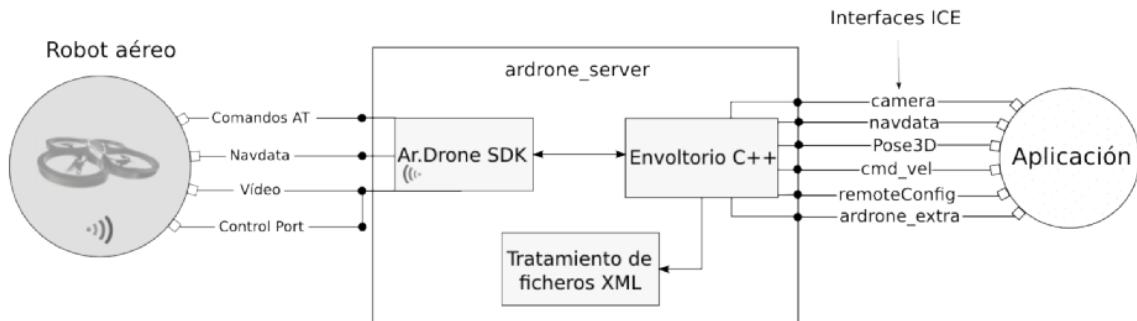


Figura 3.3: Estructura de ArDrone_Server

3.4. ICE

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

usando variedad de protocolos de red como TCP, UDP ó SSL/TTS

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

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



Figura 3.4: Esquema de ICE JS

3.5. WebRTC

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

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

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

control de la conexión, atravesar cortafuegos y NAT.

Con unas docenas de líneas de JavaScript podemos tener una videoconferencia entre pares con intercambio de archivos o datos en tiempo real. Ese es el potencial que WebRTC tiene. Pero aún así hay una serie de escollos como la señalización, descubrimiento de pares, negociación de la conexión o seguridad que debemos controlar para conseguir una llamada exitosa.

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

WebRTC está compuesto de 3 API's:

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

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

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

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 sesiones.



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 y Máquina de Estados

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

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

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

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

- Conocer si el descriptor de sesión pertenece a la parte local o remota.

- Conocer si el descriptor de sesión es una oferta o una respuesta.
- Permitir a la oferta ser especificada independientemente de la respuesta.

Para satisfacer estas premisas JSEP aborda esto añadiendo los métodos *setLocalDescription()* y *setRemoteDescription()* y teniendo un campo en los descriptores de sesión indicando el tipo de sesión que se suministra. En la figura 3.6 podemos ver el esquema de intercambio de paquetes SDP y el ajuste de los mismos en cada par según corresponde.

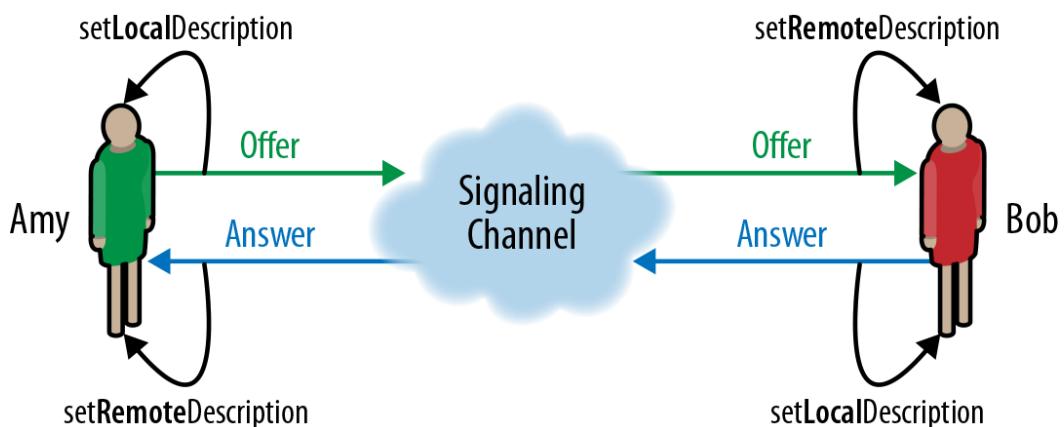


Figura 3.6: Intercambio y establecimiento de los SDP en cada par.

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

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

Formato de los Descriptores de Sesión

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

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

La forma que tiene un paquete SDP es la siguiente:

```

1 v=0
2 o=- 7729291447651054566 1 IN IP4 0.0.0.0
3 s=- 
4 t=0 0
5 a=group:BUNDLE a1 d1
6 a=ice-options:trickle
7 m=audio 9 UDP/TLS/RTP/SAVPF 96 0 8 97 98
8 c=IN IP4 0.0.0.0
9 a=rtcp:9 IN IP4 0.0.0.0
10 a=mid:a1
11 a=msid:QI39StLS8W7ZbQl1sJsWUXkr3Zf12fJUvzQ1
12 QI39StLS8W7ZbQl1sJsWUXkr3Zf12fJUvzQ1a0
13 a=sendrecv
14 a=rtpmap:96 opus/48000/2
15 a=rtpmap:0 PCMU/8000
16 a=rtpmap:8 PCMA/8000
17 a=rtpmap:97 telephone-event/8000
18 a=rtpmap:98 telephone-event/48000
19 a=maxptime:120
20 a=ice-ufrag:7sFvz2gdLkEwjZE
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:ssrc:4429951804 cname:Q/NWs1ao1HmN4Xa5
30
31 m=application 9 UDP/DTLS/SCTP webrtc-datachannel
32 c=IN IP4 0.0.0.0

```

```

33 a=mid:d1
34 a=fmtp:webrtc-datachannel max-message-size=65536
35 a=sctp-port 5000
36 a=fingerprint:sha-256 6B:8B:F0:65:5F:78:E2:51:3B:AC:6F:F3:3F:46:1B:35
37 :DC:B8:5F:64:1A:24:C2:43:F0:A1:58:D0:A1:2C:19:08
38 a=setup:active

```

Listing 3.1: Ejemplo paquete SDP

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.

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

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

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

1. El Agente ICE pide al sistema operativo las direcciones IP locales.
2. Consulta a un servidor *STUN* (*Session Traversal Utilities for NAT*) externo la tupla de dirección IP pública y puerto del par.
3. Consulta a un servidor *TURN* (*Traversal Using Relays around NAT*) como último recurso.

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

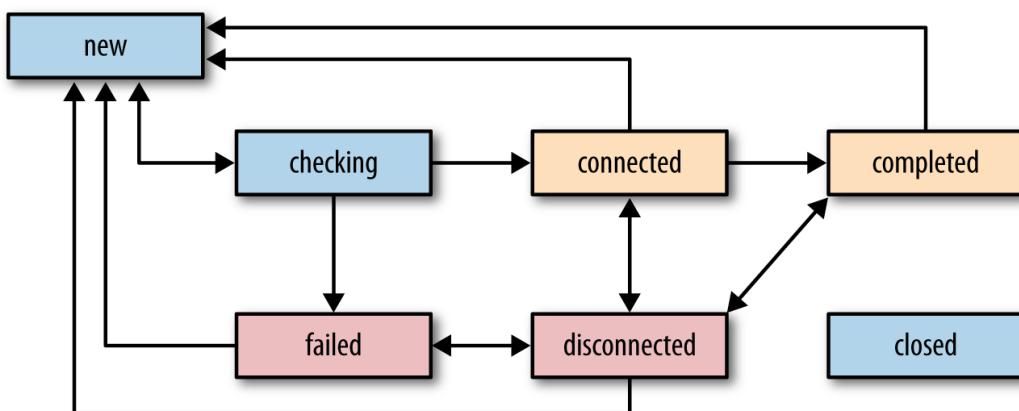


Figura 3.7: Maquina de estados y transiciones ICE.

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

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

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

Cuando una conexión se ha establecido correctamente cada Agente ICE continua

haciendo peticiones STUN periódicas al otro par, lo cual sirve también como *keepalives*.

Goteo de Candidatos ICE (*ICE Candidate Trickling*)

Recopilar IP's locales es rápido, pero conseguir las IP's 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 cual el llamante puede incrementar el número de candidatos para el llamado después de la primera oferta. Este proceso permite al llamado comenzar a establecer las conexiones ICE, sin tener que esperar a que el llamante recopile todos los posibles candidatos. Con esta técnica conseguimos un establecimiento del flujo audiovisual más rápido.

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

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

Formato de un Candidato ICE

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

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

Para tener una rica videoconferencia no es suficiente con obtener los flujos en crudo ó *formato raw* de la cámara o el micrófono. Cada flujo debe ser procesado para aumentar la calidad, sincronizarlos y ajustar el caudal (*bitrate*) de salida según las fluctuaciones del ancho de banda y la latencia entre pares. A la hora de recibir el flujo nos encontramos en la misma situación pero a la inversa. WebRTC nos da unos motores de procesado de audio y vídeo (figura 3.8) que harán todas estas cosas por nosotros.

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

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.

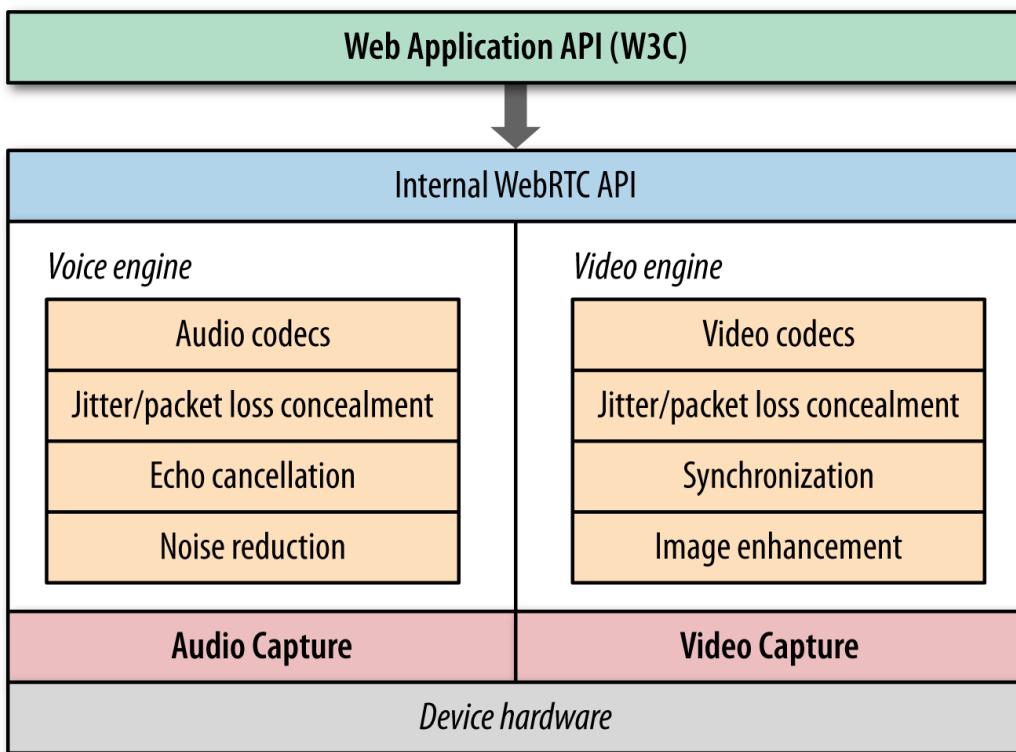


Figura 3.8: Motores de audio y vídeo de WebRTC

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

```

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

```

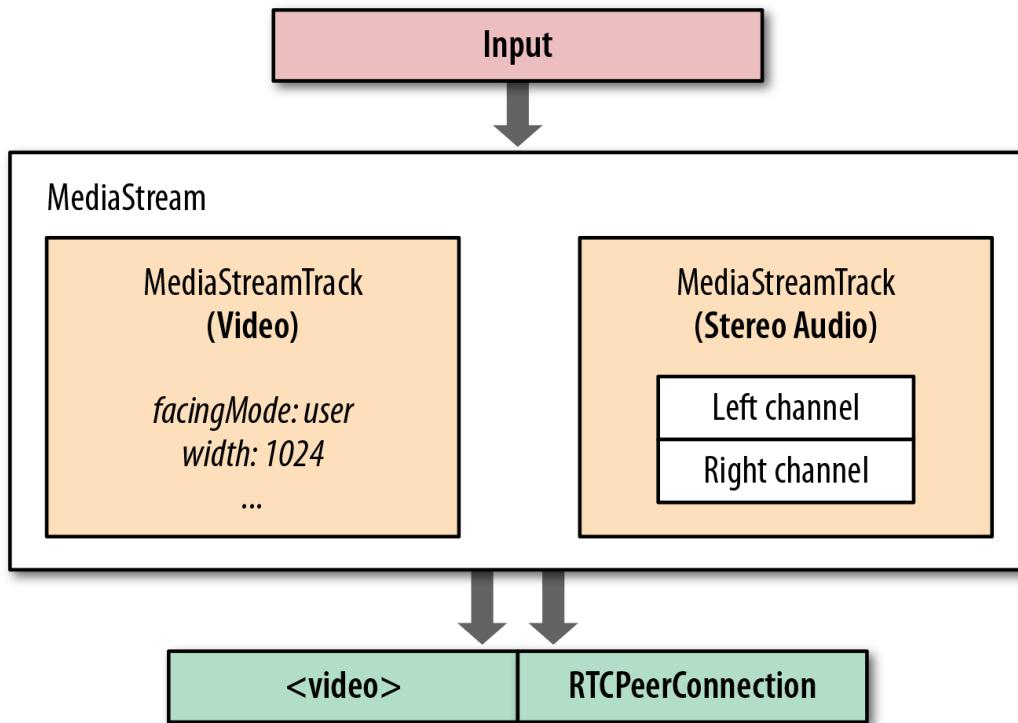


Figura 3.9: MediaStream

18 }

Listing 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 argumentosopcionales:

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

Listing 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 Trasversal. *pcConstraints* también es una variable a la que se le pueden añadir una serie de restricciones como *RtpDataChannels*, la cuál estará obsoleta en futuras versiones y se dejará de usar. Esta variable está para futuras mejoras.

Protocolos de transporte en tiempo real

El cerebro humano es muy bueno 'rellenando huecos' pero altamente sensible a los retardos. Si perdemos unas muestras de audio o vídeo no nos afecta demasiado en la percepción de lo que estamos recibiendo, pero en cambio añade un retardo al audio con respecto al vídeo y hará que ese material nos sea hasta molesto.

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

Este requerimiento de vicacidad antes que fiabilidad es la primera razón por la que el protocolo UDP es elegido para el envío de datos en tiempo real. TCP es fiable y tiene entrega ordenada de paquetes. Si uno de ellos se pierde, entonces TCP almacena los paquetes siguientes y para la retransmisión hasta que el paquete perdido es reenviado

y recibido. En cambio UDP no garantiza la entrega de paquetes, el orden de entrega, la ruta de los paquetes ni control de congestión de red.

WebRTC usa UDP como protocolo de transporte. Dadas las características de UDP, ¿podemos simplemente enviar cada paquete según llega y olvidarnos? no, ya que también necesitamos mecanismos para atravesar NAT's y cortafuegos, negociar los parámetros de cada flujo, encriptar los datos de usuario, congestión de red... Para abastecer estas necesidades WebRTC tiene una lista de protocolos y servicios que trabajan por encima de UDP [(figura 3.10)].

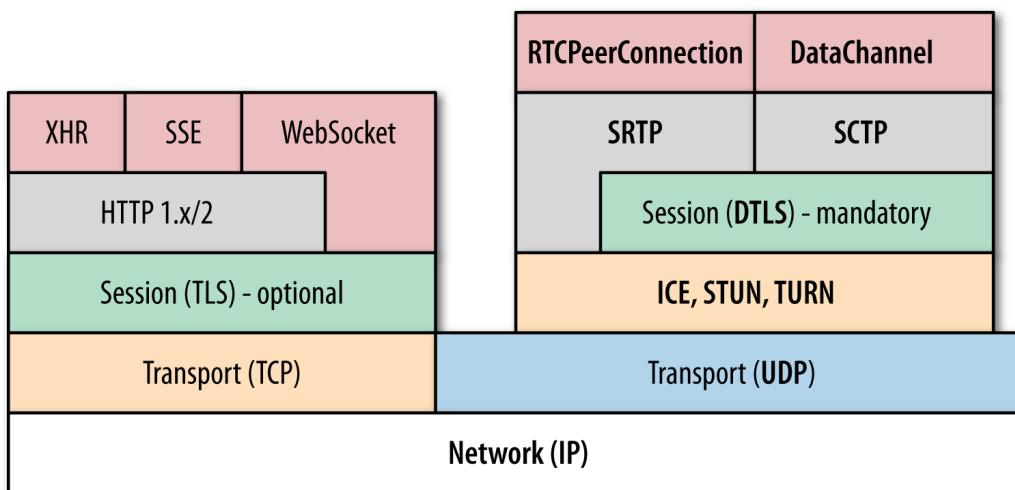


Figura 3.10: Pila de protocolos WebRTC

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

Entrega de vídeo con SRTP

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

El motor de red tiene su propio flujo de datos sin tener en cuenta desde el comienzo la capacidad de la red ni el caudal del flujo. Primero empieza enviando el vídeo y el audio a un bitrate bajo (por debajo de 500kbps) y luego comienza a ajustar la calidad

del flujo según la capacidad del ancho de banda. Según pueda variar el ancho de banda de la conexión de red así actúa el motor de red para ajustarlo.

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

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

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

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 mayola sobrecarga permitiéndonos tener una conexión más rápida. SCTP es un protocolo de transporte similar a TCP y UDP que puede funcionar directamente en la cima del protocolo IP. Sin embargo, en WebRTC, SCTP es construido sobre un túnel DTLS, el cuál corre encima de UDP.

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

Figura 3.11: Capacidades de RTCDataChannel.

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

- *Ordered*: Booleano para indicar si queremos que nos garantice la entrega ordenada de paquetes.
- *maxRetransmitTime*: Tiempo máximo para intentar retransmitir cada paquete si la entrega falla. (Fuerza el modo no fiable).
- *maxRetransmits*: Número máximo de veces que queremos que reenvíe cada paquete si la entrega falla. No puede usarse junto con *maxRetransmitTime*. (Fuerza el modo no fiable).
- *Protocol*: Permite el uso de un subprotocolo pero tiene que ser soportado por TCP/UDP.
- *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 debe estar encriptado cuando se transmite.. En *RTCDatachannel* todos los datos son codificados con *Datagram Transport Layer Security (DTLS)*. DTLS es un derivado de SSL por lo que los datos que intercambies irán igual de seguro que si usásemos SSL. Es obligatorio, para que un navegador pueda usar WebRTC, que tenga implementada esta tecnología.

Entrega de datos con SCTP

Como ya hemos visto en la figura 3.10, *RTCDatChannel* trabaja con un protocolo llamado *Stream Control Transmission Protocol (SCTP)*, el cual corre sobre DTLS, y este a su vez corre sobre UDP. Recalco esto ya que a diferencia del audio y el vídeo, para enviar data de la aplicación sí que necesitamos que lleguen todos los paquetes, por lo que si alguno se ha perdido hay que reenviarlo.

WebRTC requiere de 4 características que debe cumplir el protocolo:

1. El protocolo de transporte debe permitir tener varios canales independientes multiplexados.
 - a) Cada canal debe permitir entrega ordenada y desordenada.
 - b) Cada canal debe tener entrega fiable.
 - c) Cada canal debe tener niveles de prioridad definidos por la aplicación.
2. El protocolo debe proveer 'orientación al mensaje', por lo que debe tener fragmentación y reagrupacion de los datos.
3. El protocolo debe tener mecanismos control del flujo y de la congestión.
4. El protocolo debe tener seguridad y confidencialidad en los datos que se envían.

La última característica se cumple ya que SCTP corre sobre el túnel DTLS, por lo que los datos que envíemos 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

Solución

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

4.1. Estructura general

De manera general en cuanto a la estructura general el par local deberá establecer la conexión con el drone, acceder a sus sensores y a su 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 este 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. Este 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 llevase la batuta de la conexión fuese el ordenador local, ya que este a su vez también es el encargado de establecer la conexión con el drone. De

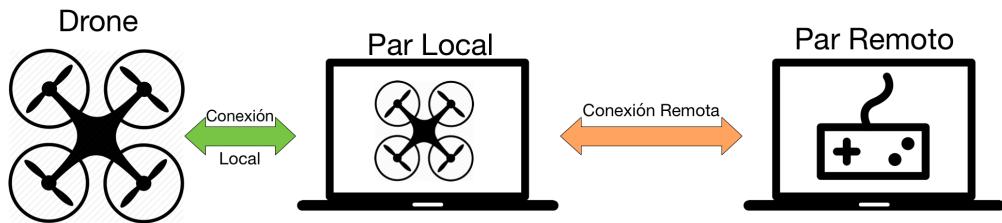


Figura 4.1: Esquema general del proyecto.

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 critico 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.

Como se puede observar el par local es el que lleva la batuta de la conexión. En primer término inicia la comunicación con el servidor de señalización. Una vez que este 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 a servidor de señalización, este 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 *RTCPeerConnection* 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

Esta primera parte la hemos solucionado dividiendo el problema en dos subproblemas. Por un lado la conexión con el servidor encargado de conectarse al cuadricóptero, *ardrone_server*, y 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

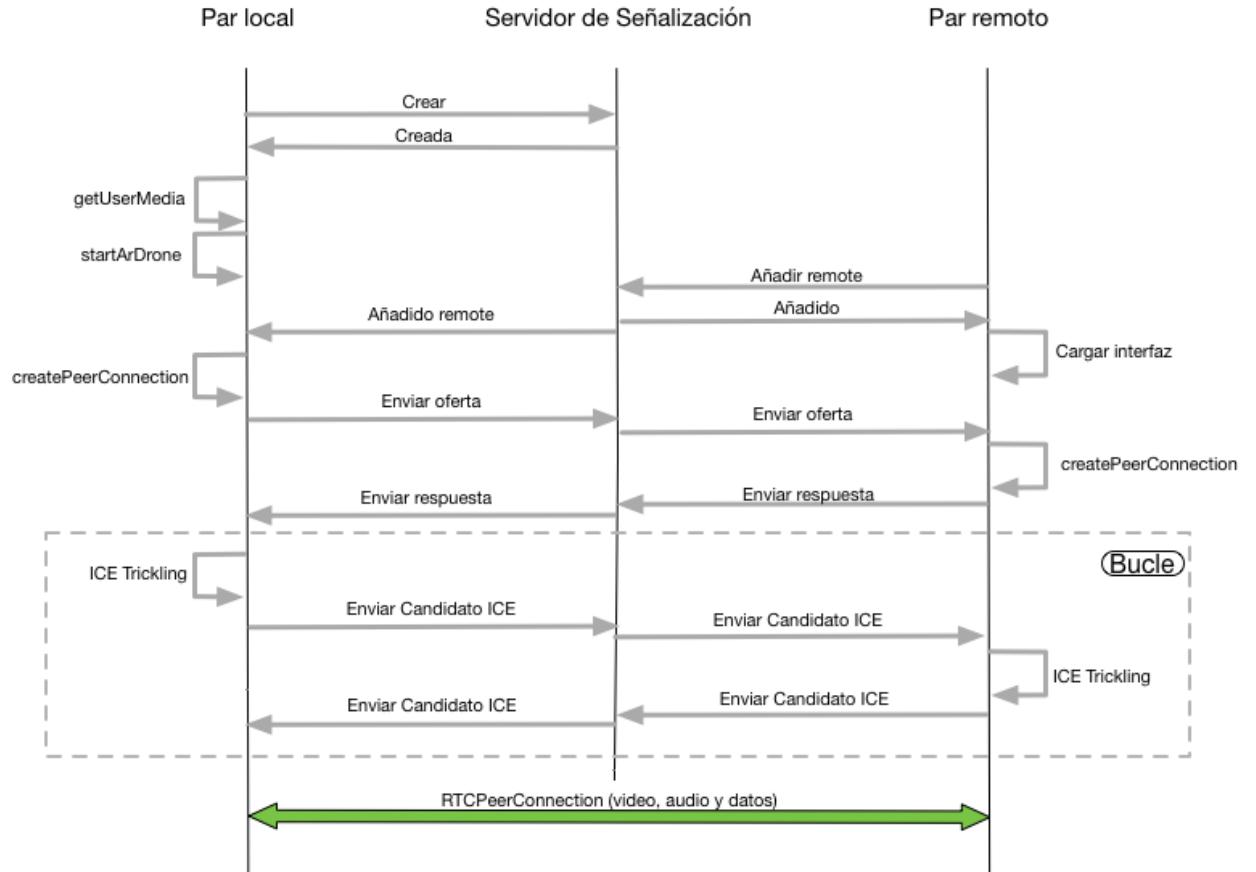


Figura 4.2: Flujo de llamada del proyecto

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, ICE y WebSockets

Para establecer la conexión con el drone se ha utilizado el componente `ardrone_server` de JdeRobot. Este componente tiene dos versiones, la componente simulada que es la que hemos instalado en forma de plugin en el simulador Gazebo y la componente real, que se conecta al drone real. Ambas componentes nos ofrecen las mismas interfaces de conexión, por lo que una única configuración de nuestra aplicación bastará para poder

trabajar con la versión simulada y la real.

En este punto podemos dividir el proyecto en dos partes. Servidor y Cliente. La parte de servidor trataremos básicamente la configuración, y la parte del cliente será la que se desarrolla desde cero.

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 más actual de JdeRobot². En esta versión ICE_JS es un plugin a parte 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. Para ello hay que añadir la siguiente línea en el archivo de configuración del servidor:

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

Listing 4.1: Activación del plugin ICEJS

Posteriormente a esto, en el mismo archivo hay que indicarle las direcciones IP y los puertos de cada interfaz de conexión. Cada conexión se corresponderá con un *WebSocket*, y la nomenclatura es la siguiente:

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

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

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

²<http://jderobot.org/Manual-5>

³<https://zeroc.com>

El servidor tiene hasta un máximo de seis interfaces diferentes a las que te puedes conectar. Cada uno 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 *pose* 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 que se encarga de recibir las órdenes de movimiento.
- **BaseExtra:** Esta interfaz nos da funcionalidades extra como el aterrizaje o el despegue del cuadricóptero.

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

```

Listing 4.3: Archivo de configuración

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.

Cliente

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

Para establecer una comunicación ICE lo primero es crear las variables ICE necesarias. Estas las creamos 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);

```

Listing 4.4: Formato *endpoints* de los *WebSocket* de ICEJS

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 que punto se produce el fallo.

Posteriormente es necesario crear una variable, que será la que actuará como *proxy*, por cada interfaz a la que necesitemos conectarnos. Esta es la nomenclatura que debe seguir:

```

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

```

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

Nótese en la nomenclatura 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*. 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 sincrónicos.

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 );
```

```

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 " +
41 ip + " -p " + cmdVelProxyPort);
42 jderobot.CMDVelPrx.checkedCast(basecmdVel).then(
43     function(ar){
44         console.log("cmdVelProxy connected: " + ar);
45         cmdVelProxy = ar;
46     },
47     function(ex, ar){
48         console.log("cmdVelProxy NOT connected: " + ex);
49     }
50 );
51
52 // Pose3D
53 var basepose3D = communicator.stringToProxy("ardrone_pose3d:ws -h " +
54 ip + " -p " + pose3DProxyPort);
55 jderobot.Pose3DPrix.checkedCast(basepose3D).then(
56     function(ar){
57         console.log("pose3DProxy connected: " + ar);
58         pose3DProxy = ar;
59         window.po = pose3DProxy;
60         resolve("Stuff worked!");
61         pose3DProxy.getPose3DDData().then(
62             function (ar){
63                 console.log("getPose3DDData() .");
64                 pose = ar;
65             },
66             function(ex, ar){
67                 console.log("Fail call getPose3DDData() .");
68             }
69         );
70     },
71     function(ex, ar){
72         console.log("pose3DProxy NOT connected: " + ex)
73     }
74 );

```

Listing 4.6: Nucleo ArDrone

En este punto ya estamos conectados con las interfaces del servidor, y por consiguiente, con el drone. Para poder teleoperar el drone hay que crear unas funciones que serán los manejadores. Por un lado hemos creado las funciones 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 }
```

Listing 4.7: Funciones aterrizaje y despegue.

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

```

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

```

21         console.log("Fail call getPoseDData(): " + ar2);
22     });
23 }

```

Listing 4.8: Variables actualización datos de los sensores.

Llamando a estas funciones periódicamente tenemos actualizados 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.

```

1
2 function sendVelocities () {
3     cmdVelProxy.setCMDVelData(cmd).then(
4         function(ar){
5             //console.log("sendVelocities.");
6         },
7         function(ex, ar){
8             console.log("sendVelocities failed.")
9         }
10    );
11 }

```

Listing 4.9: Función manejadora de las órdenes.

Dónde la variable *cmd* contiene los parámetros que necesita el drone para moverse. Esta variable es una variable CMDVel de JdeRobot y tiene esta estructura:

```

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

```

Listing 4.10: Variable CMD

4.2.2. getUserMedia

La interfaz que no hemos implementado de las que nos ofrece el servidor `ardrone_server` es la interfaz `cameraserver`. Esta interfaz se encarga de recoger las imágenes de la cámara integrada en el drone. Para tener una imagen con mas resolución y que nos permita

visualizar con mayor calidad se ha optado por colocar una cámara a bordo y acceder a ella con las herramientas que nos suministra WebRTC.

Esta cámara se conecta a nuestro ordenador local a través de una conexión USB. Camara y MiniPC van a bordo del drone. 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.
  webkit GetUserMedia || navigator.mozGetUserMedia;
```

Listing 4.11: Variable de getUserMedia.

Acceder a la cámara a con *GetUserMedia* se hace a través de una función que admite dos llamadas de vuelta o *callback*. Uno de ellos es el callback de éxito, y el segundo el de error. Según sea de exitosa 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 streaming y lo visualizaremos en un elemento vídeo de HTML5, si devuelve error mostramos un mensaje del error ocurrido.

```
1
2 // Manejador de éxito
3 function handleUserMedia(stream){
4   localStream = stream;
5
6   if (window.URL){
7     localVideo.src = window.URL.createObjectURL(stream);
8   } else{
9     localVideo.src = stream;
10 }
11 //console.log('Adding local stream.');
12 // Envío un mensaje al servidor como ack de éxito al llamar
13 // a getUserMedia()
14
15 // manejador de error
16 function handleUserMediaError(error){
17   console.log('getUserMedia error: ', error);
18 }
19
20 //Función getUserMedia
21 navigator.getUserMedia(constraints, handleUserMedia,
  handleUserMediaError);
```

Listing 4.12: getUserMedia.

El primer argumento de la función es una variable en la que le indicamos las restricciones que queremos: audio, vídeo, solo uno de ellos, resoluciones... Las restricciones elegidas para nuestro proyecto son las siguientes:

```

1
2 var constraints = {
3   audio: false,
4   video: {
5     width: { ideal: 1280, max: 1920 },
6     height: {ideal: 720, max: 1080 },
7   }
8 };

```

Listing 4.13: 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).

El flujo que nos proporciona *getUserMedia* lo utilizaremos con *RTCPeerConnection* en para enviárselo al par remoto.

4.3. Conexión multimedia entre navegadores

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

Como sub-objetivos para esta conexión tenemos los siguientes:

- *Visualización 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 deberán ser enviados al ordenador remoto donde se visualizaran.
- *Localización espacial del drone*. La brújula y GPS del cuadricóptero deberán ser recogidas por el ordenador local, y deberán enviarse al ordenador remoto para poder visualizar en este un mapa para tener localizada la posición del vehículo.

- *Ordenes.* Desde el ordenador remoto deberán enviarse hacia el ordenador local las órdenes dadas para el aterrizaje, despegue, y comandos de movimiento.

Para desarrollar la aplicación con WebRTC lo primero que hemos creado ha sido su servidor de señalización.

4.3.1. 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 en el lenguaje de programación *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, al servidor se le envían 4 tipos de paquetes. 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 diferentes tipos 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.

```

1 io.sockets.on('connection', function (socket){
2   // Manejador de mensajes genericos 'message' (intercambios SDP y
3   // Candidatos ICE)
4   socket.on('message', function (message) {
5

```

⁴<https://nodejs.org/>

⁵<http://socket.io>

```

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  // Si el que envia es remote hay que mandarselo al droner
11 } else if (socket.id== newPeer) {
12   io.sockets.socket(dronerID).emit('message', message);
13 }
14 });
15
16 // manejador de mensajes 'create' enviados por Droner
17 socket.on('create', function () {
18   //log('Server --> Droner has sido conectado');
19   socket.join();
20   dronerID = socket.id;
21   socket.emit('created');
22 });
23
24
25 // Manejador de mensajes 'join remote ' enviados por remote
26 socket.on('join remote', function () {
27   //log("Server --> Un 'remote' se ha unido.");
28
29   io.sockets.in().emit('join remote');
30   socket.join();
31   newPeer = socket.id;
32   socket.emit('joined');
33 });
34 });

```

Listing 4.14: Nucleo servidor de señalización

Una vez tenemos el servidor operativo vamos a explicar como hemos creado la conexión entre pares y como hemos usado esta conexión para transmitir el vídeo de la cámara del drone hasta el par remoto.

4.3.2. RTCPeerConnection: Transmisión de la cámara a bordo

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 esta 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.
2      mozRTCPeerConnection ||
3          window.webkitRTCPeerConnection || window.
4              msRTCPeerConnection;
5
6  RTCPSessionDescription = window.RTCSessionDescription || window.
7      mozRTCSessionDescription ||
8          window.webkitRTCSessionDescription || window.
9              msRTCSessionDescription;
10
11 RTCIceCandidate = window.RTCIceCandidate || window.mozRTCIceCandidate
12     ||
13         window.webkitRTCIceCandidate || window.
14             msRTCIceCandidate;

```

Listing 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.
- *RTCPSessionDescription*: 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
2 var ICE_config = {
3     'iceServers': [
4         {
5             'urls': 'stun:stun.l.google.com:19302'
6         },
7         {
8             'urls': 'stun:23.21.150.121'
9         },
10        {
11            'urls': 'turn:192.158.29.39:3478?transport=udp',
12            'credential': 'JZE0Et2V3Qb0y27GRntt2u2PAYA=',
13            'username': '28224511:1379330808'
14        },
15        {

```

```

16     'urls': 'turn:192.158.29.39:3478?transport=tcp',
17     'credential': 'JZE0Et2V3Qb0y27GRntt2u2PAYA=',
18     'username': '28224511:1379330808'
19   }
20 ]
21 };

```

Listing 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
);

```

Listing 4.17: RTCPeerConnection.

Esta función admite dos argumentos. El primero es la configuración ice que ya hemos visto 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 que el intercambio de paquetes en la señalización hay que 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 en el par remoto tienen 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 encargara de notificárnoslo en la consola del navegador.

```

1
2 // Funcion de exito
3 function gotLocalDescription(sessionDescription){
4   PeerConnection.setLocalDescription(sessionDescription,
5     successLocalSDP, errorLocalSDP);
6   sendMessage(sessionDescription);
7 }

```

```

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

```

Listing 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));

```

Listing 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);

```

Listing 4.20: Manejador de respuestas SDP

Las funciones de éxito y error *gotLocalDescription* y *onSignalingError* son las mismas que en el otro par, hemos mostrado su código en el listing 4.18.

Para manejar los Candidatos ICE utilizamos un método manejador *onicecandidate*, el cuál llama a la función indicada, *handleIceCandidate()*, 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.

```

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

```

```

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

```

Listing 4.21: Manejador de los Candidatos ICE locales.

Los candidatos son recibidos en los 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,
2                                     candidate:message.candidate});
3 PeerConnection.addIceCandidate(candidate);

```

Listing 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);
```

Listing 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 esta se encarga de configurar el flujo en una etiqueta vídeo de HTML5.

```

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

```

```

11    }
12    //console.log('Remote stream attached!!.');
13    remoteStream = event.stream;
14}
15
16// Método manejador
17PeerConnection.onaddstream = handleRemoteStreamAdded;

```

Listing 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 hacia ambos lados.

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 drone como las órdenes de movimiento dadas.

Para crear esta conexión basta con llamar al método *createDataChannel()* de *RTCPeerConnection* en el par local ya que es quien ha establecido 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 viene 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 // Método de error para la creacion del dataChannel
6 dataChannel.onerror = function (error) {
7   console.log("Data Channel Error:", error);
8 };

```

Listing 4.25: Establecimiento conexión RTCDDataChannle en el par local.

Si al crear la conexión ocurre un error salta un evento en el método *onerror* que nos comunica a través de la terminal el error 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, 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 actualización
6             con el drone si el canal se cierra
7     }
8 }
9 dataChannel.onopen = handleReceiveChannelStateChange;
10 dataChannel.onclose = handleReceiveChannelStateChange;
11 dataChannel.onmessage = handleMessage;
12 };

```

Listing 4.26: Manejadores de RTCdataChannle.

La función *handleMessage* es la encargada de manejar los mensajes recibidos, pero hablaremos más en profundidad de ella en la sección 4.3.3

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 una función que maneja 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. La función manejadora *handleSendChannelStateChange* es la misma que la mostrada en el listing 4.27.

```

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 comunicación datachannel del
9 // par local.

```

```

9 PeerConnection.ondatachannel = gotReceiveChannel;
10 };

```

Listing 4.27: Establecimiento de RTCDatChannlel en el par remoto.

Una vez establecida la comunicación de datos RTCDataChannel podemos enviar los datos que necesitemos con el método *dataChannel.send()* y recibirlas con el método manejador *onmessage*. A continuación vamos a ver cómo hemos utilizado estos métodos para enviar y recibir los sensores del dron 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 para recoger los datos. Esta llamada la hacemos de manera periódica con una función *setInterval()* de JavaScript.

```

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

```

Listing 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 *RTCDatChannlel*. 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 variable *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 };

```

Listing 4.29: Envio de los datos de los sensores en el par local.

Para enviar los datos 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 RTCDatChannlel.

⁶<http://www.json.org>

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

```

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

Listing 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 navdata para su posterior representación en pantalla. Estos datos los visualizaremos en unos relojes de navegación, los cuales estudiaremos con más detalle en la sección 4.4.3.

Órdenes

El par remoto es el encargado de transferir las órdenes dadas por el usuario al par local, y este a su vez al drone. El usuario tiene distintas maneras de introducir o hacer saber al equipo las órdenes que quiere dar. La primera es con unos *joysticks* multi-dispositivo e las esquinas inferiores de la pantalla, y la segunda mediante un mando conectado por USB al ordenador.

En la sección 4.4.2 se analizará 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, estas son las llamadas especiales, que son la orden de despegue y la orden de aterrizaje. Por este motivo hemos 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 }
```

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

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

sendCMDVel() es llamada cuando 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 drone. La función *sendAltYaw()*. Esta función también admite dos argumentos, la altura (o componente espacial *z*) y el giro del drone. Enviamos de la misma manera, creando una variable JavaScript con ellas, reconvertimos a objeto JSON y enviamos.

Todos estos datos los recibimos en la función *handleMessage()* del par local. En esta función reconvertimos el objeto JSON a objeto JavaScript y llamamos a una función manejadora 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
21 // de datos
22 function handleMessage(event) {
23     //console.log('Received message: ' + event.data);
24     var data = JSON.parse(event.data);
25     handleReceiveData(data);
26 }
```

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

La función manejadora *handleReceiveData()* se encarga de discernir que tipo de dato ha llegado y llamar a la función correspondiente del módulo ArDrone.

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

4.4. Interfaz web de usuario

4.4.1. Visualización de las imágenes

4.4.2. Joysticks y Gamepad

4.4.3. Relojes de navegación

Localización espacial del drone