ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

Comunicación remota con un sistema ROS mediante WebRTC

Remote communication with ROS system through WebRTC

Realizado por
Carlos Cuenca García
Tutorizado por
Vicente Manuel Arévalo Espejo
Departamento
Ingeniería de Sistemas y Automática

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO 2018

Fecha defensa:

El Secretario del Tribunal

A mis padres Isabel y Juan Carlos, por su incondicional apoyo y fe inquebrantable,
a mi hermana Alba, por su capacidad de sacar una sonrisa cuando más se necesita,
y a mi abuelo Juan, que con su compañía ilumina hasta los senderos más oscuros.

Resumen

El objetivo de este proyecto consiste en establecer una conexión directa mediante p2p (peer to peer) entre un usuario usando una aplicación web y un robot gobernado por una arquitectura robótica basada en el framework ROS, con el fin de realizar un "streaming" de audio y vídeo, así como intercambiar datos.

El establecimiento de la comunicación se basa en la tecnología WebRTC, desarrollada por Google en colaboración con Mozilla y Opera.

La aplicación web cuenta con un servidor Django, que gestiona las diferentes funcionalidades de ésta, así como el acceso de los usuarios y la comunicación inicial con el robot remoto. Dicha comunicación inicial es posible gracias al uso de webSockets, que permiten el intercambio de mensajes entre aquellos usuarios o dispositivos que se encuentren en una misma "room". El chat integrado en la web se nutre de este sistema de igual forma.

El ecosistema ROS del robot gestiona diversos nodos, que se encargan de establecer la comunicación con el usuario remoto, gestionar los recursos multimedia locales y remotos, así como del control del hardware del robot.

Por tanto, el usuario podrá controlar el respectivo robot de forma remota, visualizando lo que ve el robot y recibiendo la información que proporcionan sus sensores, con el único requisito de que tanto el usuario como el robot dispongan de conexión a internet, sin importar la ubicación de cada uno de ellos.

Palabras Clave

WebRTC, Robot, ROS, Django, Python, Web, WebSockets.

Abstract

This project has been developed for the porpose of establishing a direct connection by p2p (peer to peer) between an user using a web-based application and a robot which is framework ROS based, to create a video/audio streaming and data exchange.

This direct communication is based on WebRTC technology, developed by Google, in collaboration with Mozilla and Opera.

The web application works with Django as server, which manages all its funcionalities, users authentification and initial remote robot communication. Such communication is created using websockets, which creates different rooms, and all users in each room share their messages. Integrated web chat uses this system as well.

Robot ROS ecosystem leads all required nodes, which manage the connection establishment, local and remote multimedia resources and robot hardware.

So, users can control the robot remotely, visualizing everything the robot sees and receiving robot sensors information, with the only requirement that both of them have internet connection, no matter where is each one.

Key Words

WebRTC, Robot, ROS, Django, Python, Web, WebSockets.

Índice

1. Introducción 9
1.1 Contexto 9
1.2 Objetivos del proyecto 10
1.3 Estructura del documento 11
2. Planteamiento del problema 13
3. Sistema ROS y su aplicación en el proyecto 18
3.1 Gestión del vídeo local y remoto 19
3.2 Gestión del audio local y remoto 20
3.3 Gestión de los datos locales y remotos 21
4. WebRTC y su aplicación en el proyecto 23
4.1 WebRTC aplicado al navegador 26
4.2 WebRTC aplicado al robot 27
5. Metodología 29
5.1 1º Iteración: Desarrollo del servidor web 29
5.2 2º Iteración: Desarrollo de la aplicación web 30
5.3 3º Iteración: Desarrollo del canal de comunicación WebSocket 30
5.4 4º Iteración: Desarrollo del cliente WebRTC del navegador 31
5.5 5º Iteración: Desarrollo del cliente WebRTC del robot 31
5.6 6º Iteración: Desarrollo de los gestores multimedia del robot 32
5.7 7º Iteración: Creación del nodo ROS-NXT 32
6. Solución propuesta 33
6.1 Aplicación Django 33
6.1.1 Front End (Navegador) 33
6.1.2 Back End (Servidor) 35
6.2 Aplicación WebRosTC 43
6.2.1 Nodo Bridge 43
6.2.2 Nodo ROS 53
7. Pruebas y resultados 62
8. Conclusiones y futuras mejoras 65
9. Referencias y bibliografía 68
10. Índice de ilustraciones 70
11. Anexo 72
11.1 Manual de instalación 72

11.1.1 Aplicación Django 72
11.1.2 Aplicación WebRosTC 72
11.2 Dependencias 74
11.2.1 Dependencias del servidor Django 74
11.2.2 Dependencias WebRosTC 74

1. Introducción

1.1 Contexto

Desde que el ser humano comenzó a crear los primeros utensilios en la prehistoria para ayudarse a cazar hasta nuestros días, pasando por inventos como los vehículos a vapor y posteriormente de combustión, la bombilla, la radio, el teléfono, internet, etc., ha tenido como propósito desarrollar herramientas que le permita realizar las mismas tareas, pero de forma más rápida y eficiente.

El siglo XX, en parte debido a las dos guerras mundiales que han tenido lugar, ha sido un período de grandes descubrimientos, que han permitido, entre otras cosas, hacer del mundo un lugar más pequeño, gracias a las mejoras en las comunicaciones y los transportes. Y es que basta con echar la vista apenas 20 años atrás, cuando internet se encontraba aún en pañales, para ver la vertiginosa evolución que ha tenido en un período tan reducido de tiempo.

En la actualidad, y gracias a la inminente implantación de la tecnología 5G y de las eSIM, nos encontramos a las puertas de la denominada era del "internet de las cosas" o "IoT" donde todo, absolutamente todo, va a estar conectado a internet, llevando a un nuevo nivel el día a día de miles de personas.

Del mismo modo, la evolución de la informática ha ido de la mano del desarrollo de las telecomunicaciones, no sólo a nivel de hardware, creando dispositivos cada vez más compactos, pero no por ello menos potentes, sino a nivel de software, obteniendo unos resultados espectaculares, fácilmente reconocibles en ramas como los videojuegos, la inteligencia artificial o el *Big Data*.

Asimismo, se ha podido apreciar un gran avance en el mundo de la *robótica,* llegando a significar una nueva *revolución* industrial, llevando a otro nivel el sector de la industria, ya que los robots permiten agilizar tareas repetitivas, automatizar procesos industriales o reducir costes en la fabricación.

Pero no sólo se han aplicado a la industria, sino que han sido partícipes de proyectos muchos más ambiciosos, como la exploración espacial teleoperada, teniendo como ejemplo más calificador el caso del robot "Curiosity", enviado por la NASA a Marte en el 2011.

No es de extrañar que actualmente haya múltiples líneas de investigación abiertas con el fin de desarrollar nuevas tecnologías que permitan operar robots de forma remota para salvar limitaciones humanas (como es el caso de la exploración marciana) o evitar su participación en operaciones de alto riesgo (como los robots empleados por los Tedax).

El manejo de robots a distancia, o teleoperación de robots, plantea una serie de problemas, tales como la latencia de las comunicaciones, el radio de control del mismo, el desarrollo de interfaces hombre-máquina intuitivos, sistemas capaces de utilizar redes de comunicaciones de propósito general, etc.

1.2 Objetivos del proyecto

El propósito fundamental de este TFG es aprovechar las nuevas facilidades que brindan las recientes tecnologías de comunicación, como es WebRTC [1], para la teleoperación de robots móviles, los cuales contarán con una arquitectura software ampliamente utilizada, como es ROS [2], de tal forma que el usuario pueda enviar y recibir tanto vídeo y audio como datos del robot.

Para lograr dicho propósito, es necesario satisfacer un conjunto de sub-objetivos, entre los que se contemplan:

- Desarrollar un nodo ROS con soporte WebRTC que haga las funciones de puente (nodo *bridge*) entre un cliente WebRTC externo y ROS.
- Desarrollar un servidor que dé soporte al cliente remoto y al nodo *bridge* para iniciar el protocolo de comunicación.
- Desarrollar una aplicación web con soporte WebRTC que se conecte al sistema ROS mediante el nodo puente del robot y muestre información sobre el estado de sus sensores y permita interactuar con sus actuadores (por ejemplo, los motores).

1.3 Estructura del documento

La presente memoria consta de los siguientes capítulos:

- En el capítulo 2, se aborda la principal problemática que ha desencadenado en este proyecto. Se expone la actual situación de las telecomunicaciones, concretamente la de internet, y en cómo afecta en la teleoperación de los robots.

- En el capítulo 3, se expone el *framework* ROS [2], sus principales ventajas y su aplicación en el proyecto.

- En el capítulo 4, se expone la tecnología WebRTC [1], sus principales ventajas, su funcionamiento y su aplicación en el proyecto.

- En el capítulo 5, se detalla la metodología seguida para investigar y desarrollar este proyecto, los diferentes pasos seguidos, así como el por qué de tal cronología.

- En el capítulo 6, se explican cada una de las partes de este proyecto, su desarrollo y función dentro del conjunto del mismo. Además, se han añadido diferentes capturas del código que refuerzan las explicaciones dadas y facilitan su comprensión.

- En el capítulo 7, se muestran las diferentes pruebas realizadas para comprobar el funcionamiento del software desarrollado, así como los diferentes equipos utilizados para llevarlas a cabo.

- En el capítulo 8, se aporta un punto de vista personal sobre el proyecto, tanto de la etapa de investigación como de la etapa de desarrollo. Así como posibles mejoras y/o proyecciones futuras del mismo, posibilidades, etc.

- En el capítulo 9, se encuentran las diferentes referencias y recursos utilizados para desarrollar el proyecto, tanto enlaces de consulta, como paquetes utilizados y sus respectivas APIs.

- En el capítulo 10 se muestra un índice de las diferentes ilustraciones usadas en esta memoria.

- Por último, en el capítulo 11 se encuentra el anexo. Por un lado, se adjunta un manual de instalación de cada una de las partes diferenciadas del software. Y, por otro lado, se encuentran todas las dependencias necesarias de las que depende el proyecto. Si se deseara recrear este software en otros equipos, tales dependencias tienen que suplirse para el correcto funcionamiento de éste.

2. Planteamiento del problema

La base del manejo de un robot de forma teleoperada radica en gran medida en el sistema de comunicación entre el operador y el robot. En aquellos casos en los que la conexión con el robot es directa (local), ya sea por cable o por radiofrecuencia, las posibles dificultades se limitan a las características del medio usado (radio de acción de la señal, ruido en la señal, longitud del cable, etc.).

Sin embargo, cuando se extrapola la comunicación a un modelo remoto, donde el robot y el operador no comparten una misma red ni siquiera están cerca, entran en juego otros factores que complican la ecuación de forma notoria. En un sistema ideal de comunicación remota, el usuario y el robot se intercomunican mediante usando internet como medio (Ilustración 2-1).

No obstante, la realidad es bien distinta. Debido a la sobrecarga de las direcciones IPv4, la única forma de continuar asignando direcciones IP consiste en ir creando subredes, y cada subred es asignada a una dirección IP original, normalmente realizada por un firewall.

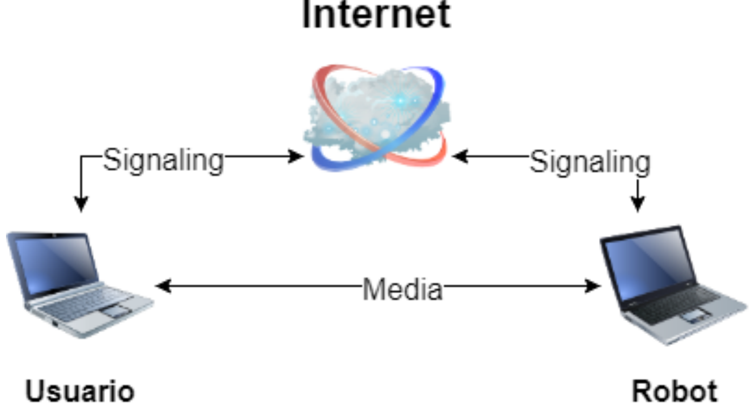

Ilustración 2-1. Modelo conexión ideal.

Este proceso es denominado *NAT* o *Network Address Translation*. Por tanto, las conexiones entrantes y salientes de cada dispositivo conectado a internet deben pasar, al menos, por un firewall (Ilustración 2-2). En función del tipo de configuración de la NAT, éste permitirá más o menos conexiones entrantes. Para más información, se puede consultar estos enlaces: NAT [3] y RFC 3489 [4].

Entonces, el lector se estará preguntando, si obligatoriamente tanto el usuario como el robot se tienen que conectar a través de una NAT, ¿cómo se va a solucionar el problema?

Realmente la cuestión no radica en el hecho de tener o no NAT, sino en cómo interviene en el proceso de comunicación, y cómo responder ante los casos de NAT más restrictivos. Para poder entender el papel que juega la NAT en el modelo de comunicación aquí expuesto, es necesario explicar muy brevemente en qué consiste WebRTC, tecnología sobre la que se basa este trabajo.

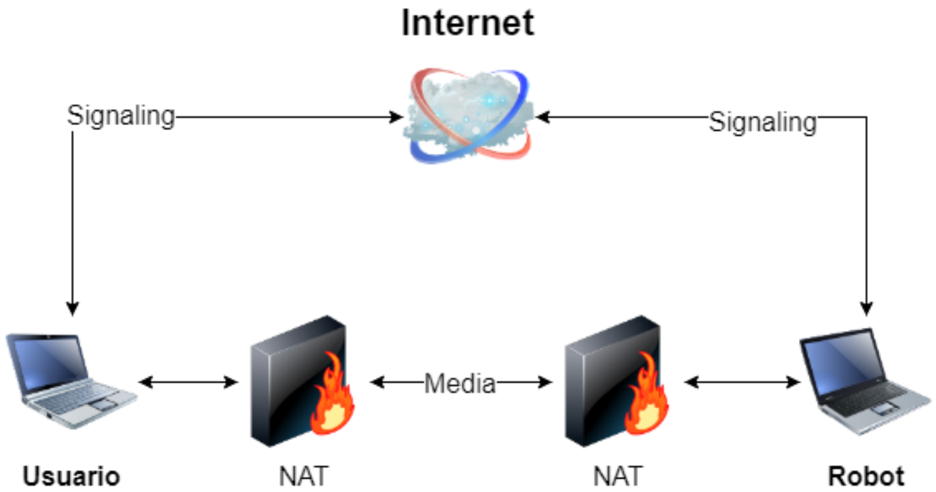

Ilustración 2-2. Modelo conexión real.

WebRTC consiste en un proyecto abierto, llevado a cabo por Google, Mozilla y Opera entre otros, con el objetivo de facilitar la comunicación en tiempo real de vídeo/audio y/o datos de forma instantánea entre navegadores y/o aplicaciones móviles. Este sistema se basa en la creación de una comunicación p2p (*peer to peer*) entre ambos usuarios, eliminando cualquier intermediario y reduciendo la latencia en la transferencia de información.

Para llevar a cabo la conexión p2p, inicialmente cada peer requiere saber cuál es su IP global para poder especificarle posteriormente al otro peer a dónde tiene que enviar los paquetes (más adelante se detallará el procedimiento de "*signaling*" entre ambos *peers*).

Para obtener dicha información, cada peer realiza una consulta a un servidor ***STUN***, que en respuesta proporciona la IP global a la que está conectado, así como la NAT con la que se conecta. (Ilustración 2-3). Y es aquí donde se encuentra el kit de la cuestión.

En muchos casos, la NAT se encuentra configurada como "NAT Simétrica". *A grosso modo,* esta NAT es la más restrictiva, debido a que el mapeo entre (IP, puerto) privado e (IP, puerto) público no se conserva, por lo que para cada conexión saliente se asigna un nuevo puerto aleatorio. Por consiguiente, cuando el host destino trata de enviar un paquete a la dirección origen, el *router* NAT rechaza el paquete porque no reconoce al que envía el paquete como un host "autorizado".

Para solventar este problema, es necesario contar con la intervención de un servidor ***TURN***.

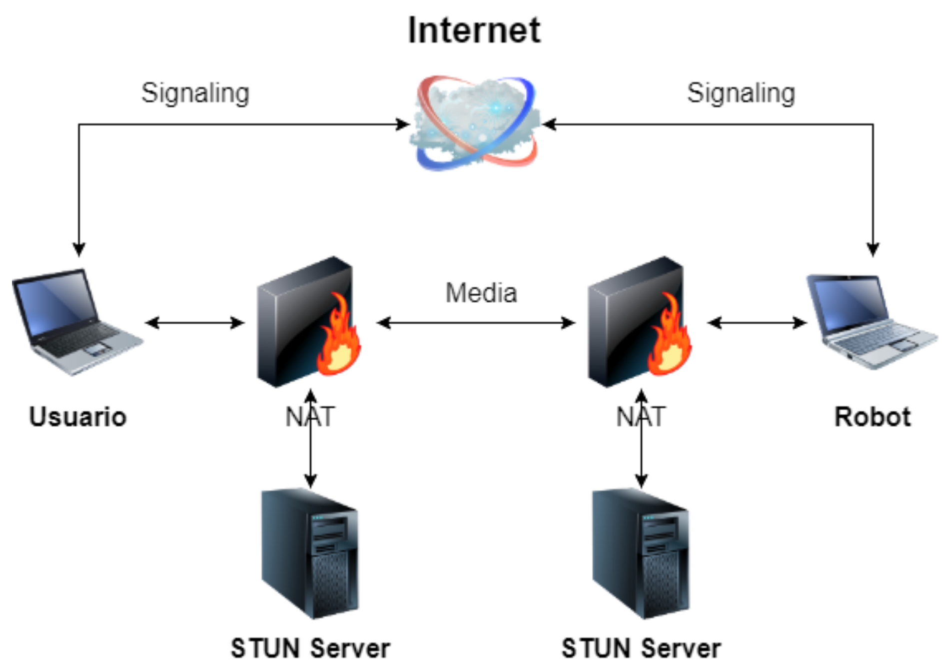

Ilustración 2-3. Usando servidores STUN.

Inicialmente, ambos *peers* tratan de establecer una comunicación directa mediante UDP. Si falla, lo vuelve a intentar mediante TCP. Y si la comunicación sigue sin establecerse, ambos *peers* tratan de conectarse usando un servidor TURN. (Ilustración 2-4).

El servidor TURN permite a los peers hacer *streaming* de audio, vídeo y datos, lo que implica que todo el tráfico pasa por ellos. Al contrario que los servidores STUN, los servidores TURN requieren autentificación por parte de los peers, y ambos peers deben conectarse al mismo servidor TURN.

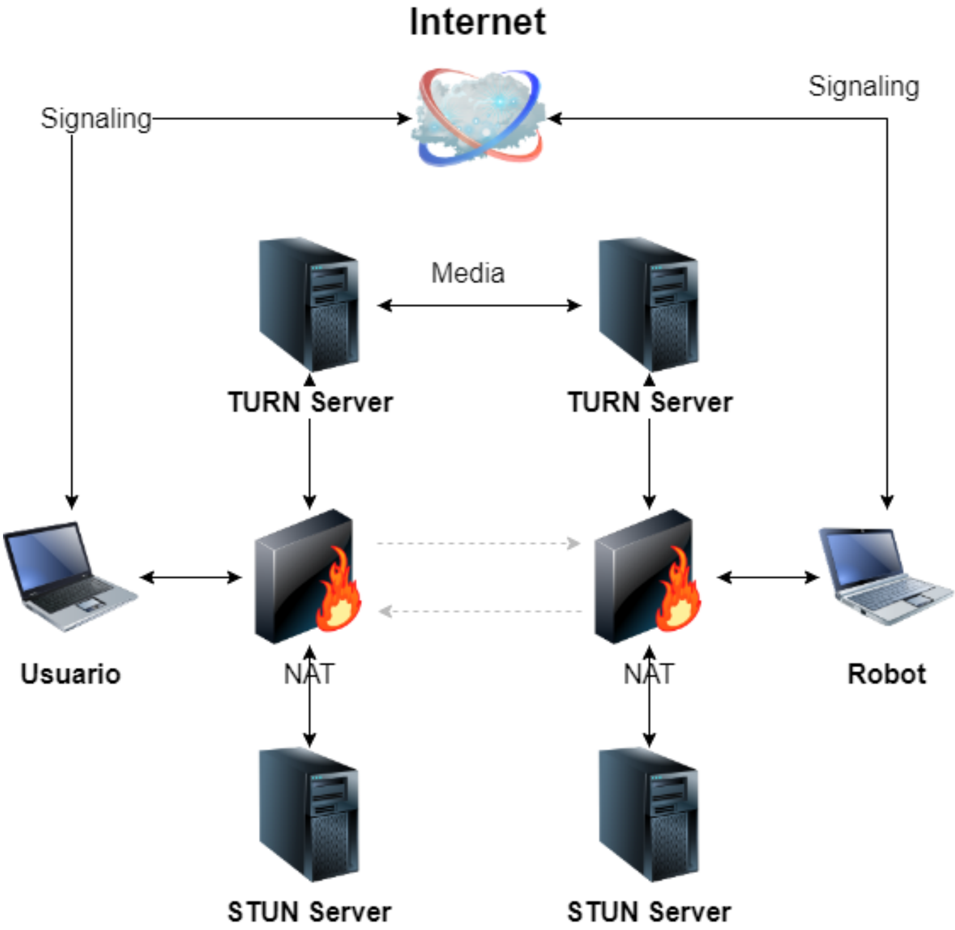

Ilustración 2-4. Usando servidores STUN/TURN.

WebRTC gestiona internamente el uso de los diferentes servidores y protocolos en función de los escenarios en los que se ejecuta, lo que facilita su implementación en cualquier aplicación.

Sin embargo, al ser una tecnología que aún se encuentra en desarrollo, implementar una aplicación estable supone una dificultad adicional. Al fin y al cabo, se trata de una tecnología sumamente reciente, que apenas tiene unos años de vida.

Pero no por ello merece menos interés, puesto que está considerado como un sistema de comunicación con un gran potencial, respaldado por grandes empresas como Google. Interconectar dos o más dispositivos estén donde estén, cuyo único requisito sea tener conexión a internet ya es motivo más que suficiente, y más considerando que en breve todo estará conectado a internet.

3. Sistema ROS y su aplicación en el proyecto

El sistema ROS [2] (acrónimo de *Robot Operating System*) es un *framework* libre cuya finalidad es proveer a los desarrolladores de librerías y herramientas para crear aplicaciones para robots. ROS se basa en nodos, donde cada herramienta conforma un nodo, y dichos nodos se comunican entre sí mediante *topics.* Cada nodo puede suscribirse a los *topics* que sean necesarios, al igual que puede publicar en tantos *topics* como requiera.

La gestión de estos canales de comunicación es llevada a cabo por el núcleo de ROS (*roscore).*

La elección del sistema ROS como base para gestionar el sistema robot no ha sido fortuita, ya que dicho *framework* aporta múltiples ventajas al desarrollador, tales como:

- Abstracción del hardware
- Versatilidad: permite desarrollar tanto en C++ como en Python.
- Multiplataforma: Aunque inicialmente se desarrolló para Linux, se está adaptando a otros sistemas operativos como Fedora, Mac OSX, Debian, etc.
- Soporte: existe una gran comunidad que mantiene el software actualizado continuamente, y solventan cualquier duda o problema que se presente.

Para llevar a cabo la comunicación remota, se ha tenido que dotar al sistema ROS de un nodo que haga de puente entre los demás nodos del sistema y el sistema remoto.

Dicho nodo puente o *bridge* tiene como propósito permitir al robot establecer una conexión de *streaming* entre el usuario remoto y el propio robot para intercambiar video, audio y datos bidireccionalmente. (Ilustración 3-1).

Por tanto, el nodo bridge tiene tres tareas principales:

- Gestión del video local y remoto
- Gestión del audio local y remoto
- Gestión de los datos locales y remotos

3.1 Gestión del vídeo local y remoto

Si bien ROS incluye un nodo gestor de vídeo, concretamente, “*video_stream_opencv*” [5] para el *streaming* de vídeo, se ha optado por desarrollar una herramienta a medida para el proyecto. La razón por la cual se ha tomado esta decisión es la siguiente:

El nodo “video_stream_opencv” dispone de un *topic* de comunicación, basado en mensajes de tipo “*ROS Images” (sensor_msgs/Image.msg)* [6]. Internamente, gestiona los mensajes usando los datos de cada *frame* para mostrarlos por pantalla, al mismo tiempo que captura la imagen del dispositivo de vídeo, lo encapsula en un mensaje de tipo “*ROS Images*” y lo publica.

La herramienta implementada en su lugar se basa en el mismo proceso y, de hecho, son muy similares, con la principal diferencia de que éste se ha implementado en Python. Entonces, ¿cuál es el propósito de dicha herramienta si son prácticamente similares?

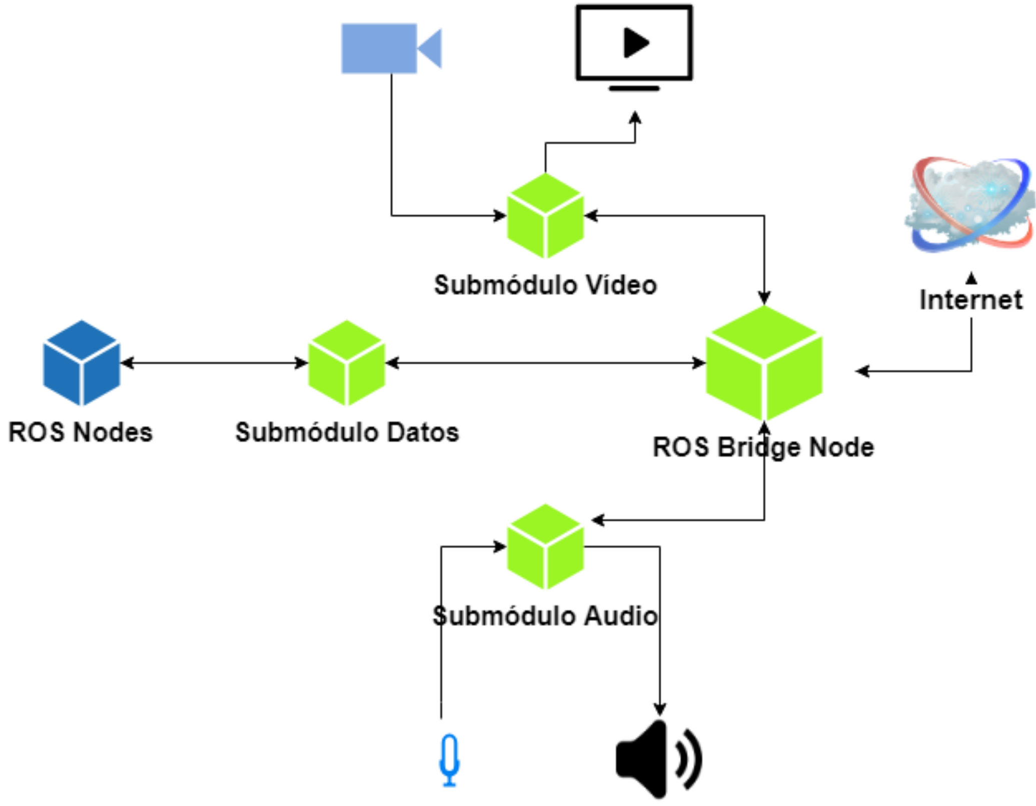

Ilustración 3-1. Estructura nodo ROS Bridge.

La señal de vídeo remota es recibida por el robot como un *array* de *bytes*, que propiamente estructurada proporciona una señal de vídeo en formato *YUV420* [7], la cual requiere ser transformada posteriormente a formato *RGB* (tanto si se usa para el nodo como si se usa para *openCV* [8]). En caso de utilizar el nodo extra "*video_stream_opencv*", la señal tendría que reconvertirse a *ROS Image message*, formato aceptado por el *topic* del nodo extra, mientras que, al usar la herramienta implementada, directamente se usa la información transformada a *RGB* para mostrarla por pantalla.

Es importante recalcar que los procesos anteriormente mencionados se efectúan para cada *frame*, por lo cual, cualquier operación extra puede suponer una diferencia de rendimiento considerable.

Con la gestión del vídeo en el otro sentido ocurre una situación similar. La señal del vídeo es recogida directamente por el nodo puente mediante *openCV*, reconvertida de *RGB* a *YUV420* y encapsulada para ser enviada al usuario remoto.

Ya que este subproceso es, sin prácticamente lugar a dudas, la rutina que más recursos consume del robot, en función de las características hardware de éste, se puede variar la calidad de la imagen enviada/recibida, pudiendo modificar parámetros como la resolución de la imagen, imagen a color o escala de grises, *frames* por segundo, etc.

3.2 Gestión del audio local y remoto

En este caso ocurre una situación similar al submódulo gestor de vídeo. ROS incorpora diversos nodos dentro del paquete "*audio_common"* [9] entre los cuales se encuentra uno para capturar el sonido del micrófono, "*audio_capture*", y otro para reproducir audio en formato *wav* y *ogg*, llamado "*sound_play*".

Ambos gestionan internamente la capa con el hardware del robot para acceder al micrófono y al altavoz. El nodo "sound_play", al recibir los datos por su *topic* mediante mensajes "*audio_common_msg*", extrae el audio y lo reproduce. Asimismo, el nodo "audio_capture" obtiene el audio del micrófono, y lo publica por su *topic* mediante mensajes "*audio_common_msg*".

La causa de la implementación de una herramienta extra para gestionar el audio, en vez de usar los dos nodos es una mera cuestión de ahorrar procesos.

Aunque es cierto que la herramienta implementada sí que difiere ligeramente de los nodos aportados por ROS puesto que se ha desarrollado en Python usando el paquete *pyAudio* [10], la esencia es la misma, puesto que dicha herramienta trata directamente con los datos provenientes del usuario remoto, al igual que encapsula el audio local directamente en el formato admitido por el nodo *bridge*.

3.3 Gestión de los datos locales y remotos

Inicialmente, y aunque el canal empleado para los datos puede usarse hasta para intercambiar archivos, se ha limitado el uso de este módulo para transferir datos locales del robot, tales como la velocidad de los motores o la información de los sensores, así como para recibir instrucciones de movimiento.

Sin embargo, este gran abanico de posibilidades no hace más que resaltar el potencial de esta tecnología, pudiendo usarse en el futuro para enviar datos de telemetría, GPS, datos de diagnóstico, etc.

Es necesario añadir que, en este caso en concreto, se ha añadido un módulo extra de comunicación interna entre el resto de los nodos de ROS y el nodo bridge, debido a un problema de compatibilidades. Concretamente, tanto el sistema interno como los nodos de ROS se ejecutan conjuntamente en C++ y Python. La versión de Python corresponde con la 2.7.

Sin embargo, el nodo bridge cuenta con diversas librerías que, dado a que requieren métodos añadidos a partir de la versión 3, resulta imposible de ejecutar en la versión 2.7. Por ello, se ha recurrido a un pequeño módulo que recrea una comunicación interna (basada en sockets UDP) para comunicar ambos módulos. Esta medida es temporal, y será prescindible tan pronto como ROS incorpore Python 3 de forma nativa.

Respecto a los aspectos técnicos del por qué se ha optado por usar módulos propios en vez de los proporcionados por ROS (aparte de los ya expuestos), así como del por qué del uso de dichos formatos de vídeo, audio y datos, se analizarán posteriormente en más profundidad.

4. WebRTC y su aplicación en el proyecto

Como ya se explicó brevemente en el capítulo 2, *WebRTC* [1] (acrónimo de *Web Real-Time Communications*) es un proyecto abierto que permite compartir en tiempo real datos, vídeo y audio entre dos dispositivos o navegadores.

Además, WebRTC provee a cualquier navegador de las herramientas necesarias para llevar a cabo la comunicación sin necesidad de instalar software adicional o de terceros. Dichas herramientas son accesibles mediante JavaScript de forma sencilla y permiten adaptarse a los diferentes navegadores que soportan esta tecnología, como Chrome, Opera o Firefox.

La elección de dicha tecnología como sistema de comunicación no ha sido fruto del azar, ya que cuenta con interesantes características que hacen que la balanza se decante en su favor, tales como:

- Se trata de un proyecto abierto, con una comunidad de desarrolladores sumamente amplia, lo que facilita, en primera instancia, su uso al ser gratuito y, por otro lado, la consulta en foros habilitados ante cualquier duda con respecto al mismo.
- Multiplataforma, facilitando su integración en cualquier equipo y evitando problemas de incompatibilidad.
- Transmisión de vídeo y audio codificada (mediante el protocolo SRTP).
- Calidad alta de vídeo y audio gracias al uso de los códecs Opus (para el audio) y VP8 (para el vídeo).
- Ancho de banda adaptativo, regulando la calidad del audio y el vídeo en función de la calidad de la conexión, ofreciendo siempre la mejor calidad posible.

Pero… ¿exactamente cómo funciona WebRTC? En el capítulo 2 se ha expuesto la problemática existente en la comunicación remota con los robots, y que éste se basa en p2p, sin embargo, no se ha hecho mención del proceso por el cual WebRTC crea dicha conexión entre dos puntos cualesquiera que quieran conectarse.

Aunque la API de esta potente herramienta está en constante cambio, (afortunadamente) el proceso del establecimiento de conexión se mantiene prácticamente igual:

1. En primer lugar, cada uno de los *peers* debe declarar qué desea compartir (audio, vídeo, etc.) y qué desea recibir. (Ilustración 4-1).

```
// WebRTC Constraints
// What media is going to send
var constraints = {
    audio: true,
    video: true
};

// What media is going to receive
var offerOptions = {
    offerToReceiveAudio: true,
    offerToReceiveVideo: true
};
```

Ilustración 4-1. Ejemplo JavaScript para definir qué medios se van a enviar y cuáles se van a recibir.

2. Se crea el objeto *RTCPeerConnection,* que hace referencia al propio *peer*. Si se desea compartir datos mediante un *dataChannel*, uno de los dos usuarios debe crear dicho canal a su vez. (Ilustración 4-2).

```
// Creating peer object
console.log("Creating Peer Client ...")
djangoPeerClient = new RTCPeerConnection(djangoPeerConfiguration);

// Creating data channel
console.log('Creating data channel')
dataChannel = djangoPeerClient.createDataChannel("data");
```

Ilustración 4-2. Ejemplo JavaScript para crear un peer y un dataChannel.

3. Una vez definido el usuario, en caso de que desee compartir audio y/o vídeo, dichos recursos deben ser añadidos al peer. (Ilustración 4-3).

```
// Add local stream to peer
console.log('Adding local media to django peer... ');
localStream.getTracks().forEach(track => djangoPeerClient.addTrack(track, localStream));
```

Ilustración 4-3. Ejemplo JavaScript para añadir los medios locales al peer.

4. Llegados a este punto, uno de ellos crea una oferta, donde se detallan múltiples campos, como su dirección IP global (es aquí donde el peer realiza la consulta al servidor STUN), los medios que desea compartir, formatos y *rate* de audio/vídeo, etc. (Ilustración 4-4).

```
djangoPeerClient.createOffer(offerOptions).then(function (sdp){
    console.log("Sending offer...\n" + JSON.stringify(sdp));
    sendNegotiation("offer", sdp);
    return djangoPeerClient.setLocalDescription(new RTCSessionDescription(sdp));
}, function(err) {
    console.log("Error creating offer... " + err);
});
```

Ilustración 4-4. Ejemplo JavaScript para crear una oferta y mandársela al peer remoto.

5. Dicha oferta se corresponde con la descripción local del peer. Por lo que se la manda al peer remoto para que la tome como descripción remota, cree su propia descripción (o respuesta) y se la vuelva a mandar al peer local. Este proceso se denomina "*Signaling*".
6. Por otro lado, y de forma prácticamente paralela, ambos *peers* generan *ICE Candidates*, para enviárselos al peer remoto. (Ilustración 4-5).

```
// Defining onIceCandidate event -> This handler is called
// when network candidates become available
djangoPeerClient.onicecandidate = function(event) {
    if(!djangoPeerClient || !event || !event.candidate) return;
    var candidate = event.candidate;
    console.log('Sending candidate...');
    sendNegotiation("candidate", candidate);
};
```

Ilustración 4-5. Ejemplo JavaScript para crear ICE Candidates cuando se encuentren disponibles.

¿Y qué son los *ICE Candidates*? ICE es el acrónimo de "*Interactive Connectivity Establishment*", y hace referencia a los diferentes métodos usados en NAT para establecer comunicaciones VOIP, p2p, mensajería instantánea, etc.

Los diferentes candidatos aportan información sobre la dirección IP y el puerto donde es posible establecer una comunicación, así como el protocolo a usar, ya sea UDP o

TCP. Además, indican qué tipo de NAT tienen y si es necesario recurrir a un servidor TURN para crear la conexión. Ambos *peers* generan múltiples candidatos, y cuando se crean un par compatible, se establece la conexión.

Es posible que se sigan generando candidatos aunque la comunicación ya esté establecida, y si surge un par con mejores características, la conexión se actualiza para mejorar la calidad de la misma.

El modelo usado para llevar a cabo la comunicación en este TFG difiere ligeramente del modelo original proporcionado por WebRTC. En este caso no se trata de conectar dos usuarios usando sus navegadores, sino de conectar un usuario usando su navegador con un robot. (Ilustración 4-6). Por consiguiente, el uso de WebRTC en este proyecto se divide en dos grandes bloques:

- WebRTC aplicado al navegador
- WebRTC aplicado al robot

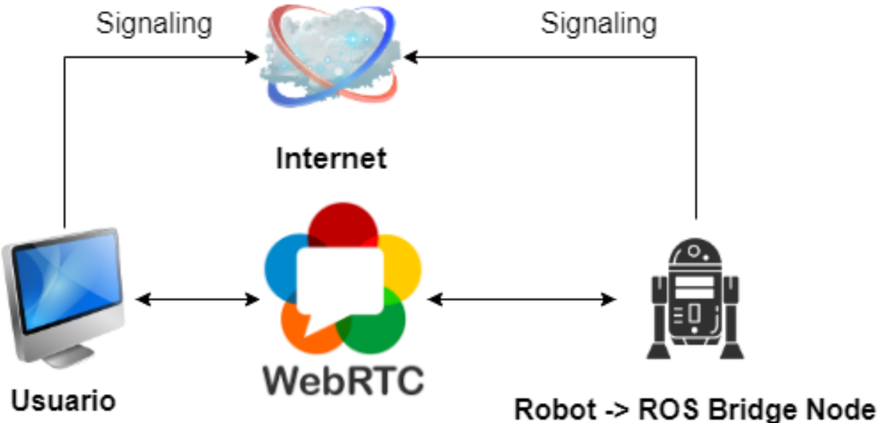

Ilustración 4-6. Estructura de los nodos WebRTC.

4.1 WebRTC aplicado al navegador

Gracias a la sencilla API de WebRTC para el navegador, el desarrollo del peer del usuario no supone un gran reto.

Capturar el vídeo y el audio del navegador se realiza con una simple función, "*getUserMedia*", (Ilustración 4-7) que recoge el vídeo de cualquier cámara conectada

al equipo, ya sea *USB* o integrada y el audio del micrófono de forma totalmente transparente para el programador.

Asimismo, los procedimientos necesarios para realizar el “*Signaling”* y el intercambio de “*ICE Candidates”* están contemplados en la API, por lo que el único requisito es realizar el proceso en el correcto orden para que la oferta y los candidatos se generen de forma apropiada.

```
// To get local media
function getLocalMedia() {
    console.log('Getting local stream... ')
    navigator.mediaDevices.getUserMedia(constraints).then(gotStream).catch(handleError);
};
```

Ilustración 4-7. Ejemplo JavaScript donde se capturan los medios locales (audio/vídeo).

4.2 WebRTC aplicado al robot

La creación de un nodo WebRTC que se ejecute en un robot con ROS ya no supone un aspecto tan trivial como el anterior. El tratamiento de la información, así como la creación de candidatos o el proceso de *signaling,* ya no se encuentra tras la capa del navegador.

Si bien es cierto que WebRTC aporta tanto su código fuente como su código compilado en C++, usarlo supone integrar ROS en dicho código, lo que conlleva a entender dicho código en su totalidad, además de adaptarlo a este caso particular. De hecho, la investigación inicial para este TFG se invirtió en gran medida en usar el código fuente de WebRTC.

Sin embargo, el tener que compilarse con *cmake*, tener problemas con la versión de C++ usada (C++98 y C++11), y prácticamente tener que prescindir de *debugger*, hicieron de esta línea de investigación una vía muerta.

Afortunadamente, gracias a la gran comunidad que apoya esta tecnología, se ha desarrollado un nodo WebRTC en Python, que pese a estar aún en desarrollo, cumple con las funciones básicas necesarias. El proyecto, llamado “*aiortc”*, contiene varias clases que gestionan el peer, el encapsulamiento de los datos o el proceso de “*signaling”,* entre otros.

No obstante, al estar aún en desarrollo, carece de ciertas funcionalidades, como el intercambio de "*ICE Candidates*" o establecer los servidores STUN/TURN apropiados, con lo que ha sido necesario modificar ciertos parámetros del código para poder llevar a cabo la comunicación remota con éxito.

Cabe aclarar, ya que el lector se lo estará preguntando, cómo es posible que ambos peers se intercambien tanto los "*ICE Candidates*" como los datos para el "*Signaling*", si hasta que dichos datos no son compartidos, la comunicación no se puede establecer.

Si bien es cierto que WebRTC menciona en numerosas ocasiones un "*Signaling Channel*" por el que intercambiar toda esta información, no proporciona ningún medio para llevar a cabo tal comunicación, por lo que corre a cargo del desarrollador el escoger la mejor opción que cumpla tales requisitos.

Es en este momento donde entra en escena el intermediario, que crea el canal de comunicación temporal entre ambos *peers*. Pero, en contra de lo que pueda parecer, no es requisito indispensable implementarlo de forma independiente, puesto que este intermediario ya existía desde el principio.

Basta con recordar que ya se dispone de una aplicación web, la cual requiere de un ***servidor*** para poder ejecutarse. Concretamente, para este proyecto se ha optado por un servidor Django [11], el cual, al ser un servidor de código abierto, dispone de múltiples paquetes complementarios, que le aportan infinidad de nuevas funciones.

De entre las diferentes opciones posibles (HTTP, nodeJS, sockets, etc.) la elección que más aporta a este proyecto es "*WebSockets*", dado a que se adapta a la tecnología implementada en el servidor con su paquete "*Django-channels*" [12] y permite implementar nodos de múltiples formas, facilitando la creación de un canal inicial de comunicación entre ambos *peers.*

"*Django-channels*" se basa en la creación de "*rooms*" donde los usuarios que se encuentran en la misma "*room*" comparten los mensajes. Por tanto, al publicar un mensaje, todos los demás usuarios lo reciben.

Más adelante se explicará detalladamente la implementación de dicho canal tanto en el servidor como en el robot.

5. Metodología

El presente proyecto es el resultado de la unión de un conjunto de bloques o secciones, cada una dependiente de la anterior, cuyo funcionamiento está directamente ligado a la correcta ejecución de todas sus partes.

Es por ello por lo que el método de desarrollo software elegido para este proyecto ha sido el desarrollo iterativo e incremental (IDD) [13]. En particular, se han realizado una serie de **iteraciones** en la que cada iteración ha constituido en sí un pequeño proyecto con las siguientes fases:

- Análisis de requisitos
- Diseño
- Implementación
- Prueba
- Mantenimiento

Las iteraciones que definen el desarrollo completo de este proyecto son las siguientes:

5.1 1º Iteración: Desarrollo del servidor web

Esta primera etapa o iteración, sin lugar a duda, ha resultado ser la más complicada de decidir, ya que se trata de la base del proyecto.

Para llevar a cabo el desarrollo de la aplicación, se requería un servidor web, en efecto, pero dado a que no gestionaría únicamente las diferentes páginas web, sino el control de usuarios, dispositivos remotos o la conexión inicial entre los *peers*, los candidatos que cumplían todos los requisitos eran escasos. Además, dependiendo de la elección del servidor, dependería la implementación de herramientas necesarias para suplir los objetivos del proyecto.

Por todo esto, se ha optado por desarrollar un servidor web usando Django [11], al que se le ha implementado una base de datos que gestiona los diferentes dispositivos remotos y su uso por parte de los usuarios.

5.2 2º Iteración: Desarrollo de la aplicación web

Una vez implementado el servidor, el siguiente paso ha sido el desarrollo de la aplicación web. Gracias al uso de Django, el desarrollo de la misma se ha visto sumamente simplificado, tanto a nivel de diseño web, gracias al uso de plantillas y bloques, como a nivel de *dataflow*, autentificación de usuarios, o manejo de formularios.

Para este proyecto, se requiere una aplicación web que facilite la interacción del usuario con el robot, por lo que, en primer lugar, es necesario que los usuarios que la usaran estuvieran correctamente identificados y que, además, los usuarios puedan interactuar entre sí mediante un chat.

De esta forma, una vez implementada la aplicación, se ha probado a intentar a acceder a la aplicación más allá de la página de *login*, sin éxito. Además, con cada entrada del usuario se guarda un registro en la base de datos, llevando un control de los accesos.

5.3 3º Iteración: Desarrollo del canal de comunicación WebSocket

Contando con la aplicación web desarrollada, así como el servidor, la siguiente etapa en la implementación de la aplicación ha sido la creación del canal de comunicación webSocket. Esta etapa constituye un paso crucial en el desarrollo de la aplicación, pues supone la conexión inicial entre el usuario y el *peer* remoto. Sin dicha conexión, la comunicación p2p no puede establecerse.

Por ello, ha sido necesario implementar diferentes apartados para satisfacer esta etapa:

1. En primer lugar, el servidor Django ha requerido la adición de un paquete adicional que gestionara una comunicación entre el *front end* (navegador) y el *back end* (servidor). Concretamente, el paquete "*Django-channels*" [12], que gestiona los clientes webSocket y sus respectivos mensajes.
2. Crear el cliente webSocket en la aplicación web mediante JavaScript, junto con sus respectivos eventos.

3. Crear un cliente externo webSocket (inicialmente ejecutado en consola), implementado en Python y basado en el paquete "*websockets*" [14].

Una vez testeado que los mensajes enviados desde la aplicación web son recibidos por el cliente externo y viceversa, se ha dado por concluida esta etapa.

5.4 4º Iteración: Desarrollo del cliente WebRTC del navegador

Ya que se ha definido el canal inicial de comunicación entre el navegador y el peer remoto, se ha podido proceder a la construcción del cliente WebRTC del navegador.

Como se ha podido apreciar en el apartado 4.1, el cliente JavaScript requiere de un "signaling channel" para intercambiar la oferta y la respuesta, así como los *ICE Candidates.*

Dado a que el robot es el queda a la espera de que un usuario desee conectarse, es el cliente WebRTC del usuario el que inicialmente crea la oferta y se la envía al robot para que la procese.

5.5 5º Iteración: Desarrollo del cliente WebRTC del robot

Siendo el cliente *webSocket* la base de este nodo, una vez lograda una comunicación, el objetivo de esta etapa consiste en manejar los mensajes entrantes por el canal *webSocket*, con el fin de esperar a que el usuario remoto requiera establecer una comunicación y envíe su oferta, para procesarla, enviarle una respuesta y establecer la comunicación p2p.

La creación del peer del robot, así como los procesos para gestionar la oferta y generar la respuesta son proporcionados por el paquete *aiortc*, el cual supone una adaptación de WebRTC para Python.

Sin embargo, debido a que este paquete aún se encuentra en fase de desarrollo, no cuenta con todos los procesos necesarios, y se han tenido que suplir algunos métodos para poder llevar a cabo la comunicación con éxito.

5.6 6º Iteración: Desarrollo de los gestores multimedia del robot

Una vez establecida la conexión p2p con el usuario remoto, el siguiente objetivo consiste en gestionar correctamente los recursos multimedia, tanto locales como remotos.

Por ello, se han implementado módulos encargados de recibir y mostrar los datos multimedia remotos, así como obtener los datos multimedia locales, encapsularlos y enviarlos.

Inicialmente se ha probado a simplemente rebotar la información entrante del usuario remoto, a modo de espejo, con el propósito de verificar que:

1. El robot recibe correctamente el vídeo/audio y los datos.
2. El robot procesa correctamente la información recibida.

Posteriormente, se han terminado de implementar los métodos para que, en vez de rebotar la información entrante, el robot envíe su propia información.

5.7 7º Iteración: Creación del nodo ROS-NXT

Finalmente, el objetivo ha sido integrar el nodo desarrollado del robot con ROS, de tal forma que sirviera de puente con el usuario remoto. Además, se ha decidido usar como robot un *lego NXT* [16], por lo que se ha añadido como requisito la implementación del nodo ROS que gestione dicho robot.

Por tanto, se ha implementado un nodo ROS que gestiona el robot, basado en "*NXT-Core ROS"* [17] y en "*NXT-Python*" [18], que se comunica con el nodo *bridge.*

De esta forma, el usuario puede actuar sobre el robot de forma remota, al mismo tiempo que recibe información proveniente de los sensores de éste.

6. Solución propuesta

Como ya se podrá figurar el lector, este proyecto se divide en dos grandes bloques, cada uno representando a los dos *peers* que se pretenden conectar.

A continuación, se van a desgranar ambas partes, exponiendo con detalle cómo se han implementado, las librerías o paquetes usados y el propósito de cada sección, con el fin de facilitar la comprensión al lector del funcionamiento en conjunto de cada módulo. Además, se añadirán algunos extractos de código con el fin de complementar la explicación dada.

6.1 Aplicación Django

Gestionando el módulo del usuario, se encuentra una aplicación basada en Django. Si bien es cierto que existen otros *framework* que realizan funciones similares, Django cuenta con ciertas características distintivas que han decantado la balanza a su favor, tales como:

- Desarrollado en Python.
- Gestión sencilla de la base de datos.
- Control y gestión de usuarios incorporado.
- Sencilla implantación de módulos externos para añadir funcionalidades extra.
- Soporte de servidor Web.
- Código abierto.

6.1.1 Front End (Navegador)

Con el fin de hacer más amigable la interacción del usuario con la aplicación y, por consiguiente, con la comunicación remota del robot, se ha desarrollado una aplicación web, usando Django como soporte.

Como se puede apreciar en el *dataflow* de dicha aplicación (Ilustración 5-1), cualquier usuario anónimo que acceda a la aplicación únicamente tendrá acceso a la página inicial (donde se muestra un pequeño resumen del proyecto en sí, así como otros datos de interés) y a la página de acceso. El resto de la aplicación se encuentra disponible únicamente para usuarios registrados.

La razón por la cual los usuarios necesitan disponer de credenciales para poder acceder a las funciones remotas del robot, así como del chat, no es más que garantizar un control sobre quién usa cada recurso en cada momento, evitar que dos usuarios accedan a la vez al mismo robot, etc.

Además, el chat se nutre del nombre del usuario para mostrar quién ha escrito cada mensaje, y evitar mensajes de usuarios ajenos a la aplicación.

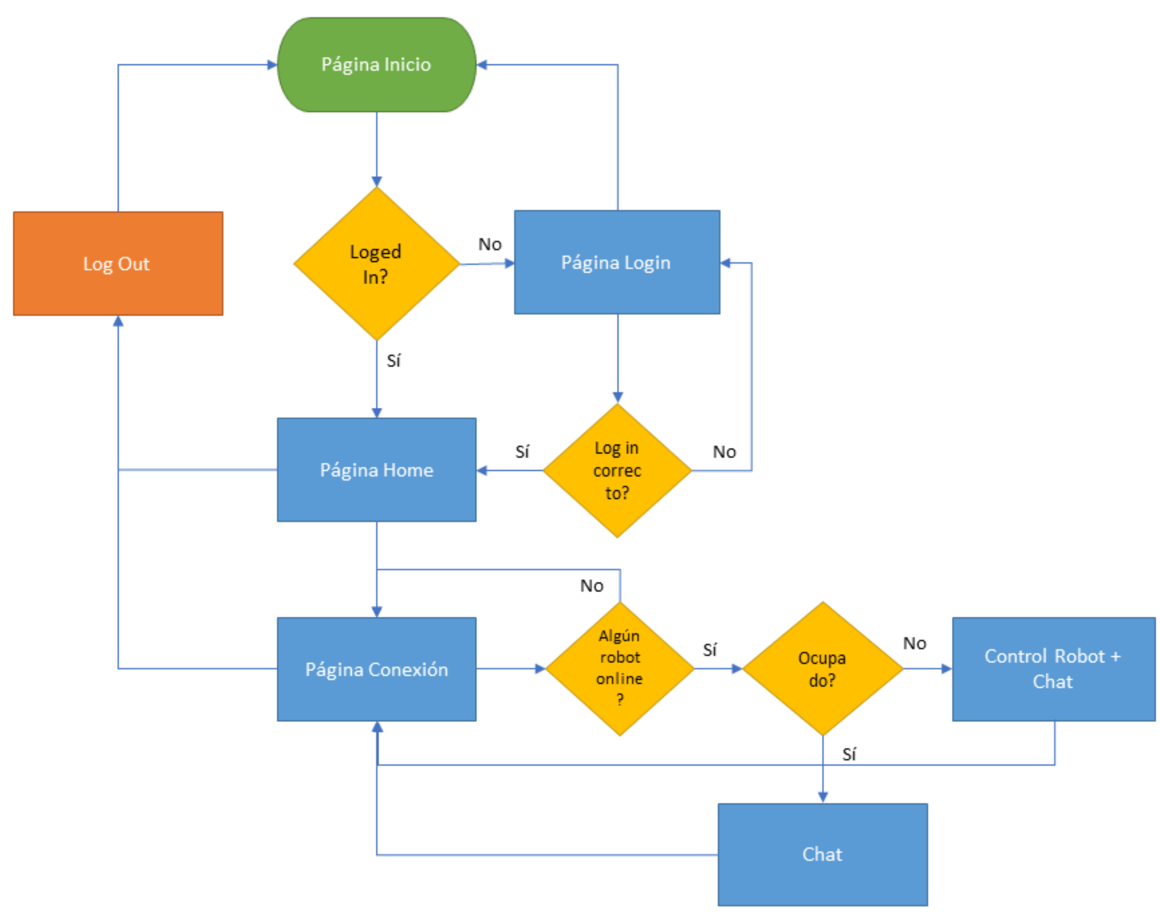

Ilustración 6-1. Django FrontEnd DataFlow.

En la página *home*, el usuario tiene a simple vista un resumen de su actividad en la aplicación, visualizando, entre otros, sus últimos accesos, últimas conexiones, etc. Para acceder a los diferentes robots disponibles, el usuario accede a la página "*conexiones*", donde se muestran los robots conectados.

En caso de que el robot ya esté en uso, el usuario únicamente podrá acceder al chat correspondiente a dicho robot, para interactuar con los demás usuarios que estén interesados en su actividad. Tan pronto como el usuario que esté usando el robot abandone la página, libera el recurso permitiendo que otro usuario pueda usarlo.

6.1.2 Back End (Servidor)

La implementación del servidor se ha planteado de forma modular, de tal forma que cada parte cumpla una función diferente, facilitando posibles futuras modificaciones y/o ampliaciones del mismo. En el conjunto del servidor se contemplan los siguientes paquetes, cada uno con una función claramente definida:

- Templates
- Sockets
- Exceptions
- Definitions
- Database
- Security
- Static files

6.1.2.1 Templates

Este módulo contiene todos los ficheros "*html*" que constituyen el front end. Para facilitar su desarrollo, se ha construido un fichero plantilla, con los elementos básicos de todas las páginas, como la barra superior o el pie de página, para evitar la redundancia del código.

El uso de plantillas, así como de otras secciones de código se simplifica enormemente gracias a la sencilla gestión que ofrece Django, mediante la división por "bloques". De esta forma, cada página constituye un bloque, que se inserta en la plantilla general, transformándose en la página final deseada para cada circunstancia.

6.1.2.2 Sockets

Una definición rápida para este apartado podría ser "gestor de conexiones del webSocket", aunque seguramente, no aporte gran información al lector.

Como se ha comentado anteriormente, Django permite definir de forma sencilla tanto la capa de "*front end*" como la capa "*back end*". Pero no sólo eso, sino que permite gestionar la capa intermedia, o "*middleware*" usada a menudo para comunicar el servidor con el navegador. Django incluye varios paquetes, cada uno con una función específica como, por ejemplo, un gestor de las peticiones al servidor realizadas por los usuarios (*requests)*, vinculándolos usando sesiones (gracias a las credenciales de los usuarios). En este caso, el propósito de este paquete es gestionar la conexión entre el cliente webSocket del navegador (creado mediante JavaScript) (Ilustración 5-2) y el cliente remoto webSocket del robot.

```
// Get current scheme
var ws_scheme = window.location.protocol == "https:" ? "wss" : "ws";
var chatWSPath = ws_scheme + '://' + window.location.host + "/ws/chatRoom/" + deviceName + "/";
var deviceWSPath = ws_scheme + '://' + window.location.host + "/ws/devices/" + deviceName + "/";

console.log("Connecting to " + deviceWSPath);

webrtcWS = new WebSocket(deviceWSPath);

// Defining all webSocket events
webrtcWS.onopen = function(e) {
    console.log('Device websocket opened');
    getLocalMedia();
};

webrtcWS.onclose = function(e) {
    console.log("Device websocket closed");
};

webrtcWS.onmessage = function(e) {
    // Handling message
    message_handler(JSON.parse(e.data));
};

webrtcWS.onerror = function(e) {
    console.log('Websocket error');
};
```

Ilustración 6-2. Cliente WebSocket JavaScript.

El paquete sockets contiene tres ficheros. El primero, "*routing.py*" tiene como finalidad redirigir las conexiones entrantes de los clientes webSocket en función de su URL de conexión. Como se ha podido observar en la ilustración 5-2, se definen dos *url*, una para la conexión del chat, y otra para la conexión con el dispositivo remoto.

Las conexiones a la *url "/ws/devices/"* son gestionadas por el paquete "*devicesConsumer.py*", mientras que las conexiones a la *url "/ws/chatRoom/"* lo son por el paquete "*usersConsumer.py*". De este modo, se puede definir un comportamiento diferente en función de la *url* a la que se conecte cada usuario.

Cada *consumer* hereda de la clase *"AsyncWebsocketConsumer",* de la cual es necesario definir las funciones "*connect*", "*receive*" y "*disconnect*", con el propósito de especificar qué hacer cuando un usuario se conecta, manda un mensaje o se desconecta. (Ilustración 5-3).

Además, gracias a las herramientas de Django, se puede incluso diferenciar si el usuario que envía una solicitud de conexión corresponde a un usuario registrado en el sistema o no.

```
'''
    This class handles devices websocket connections
'''
class DevicesConsumer(AsyncWebsocketConsumer):

    ####################################################################################
    ######################### Web Socket Event Handlers ##################################
    ####################################################################################

    # Control message exchange stage
    isAsymmetric = True
    isDevice = False
    username = None
    roomName = None

    async def connect(self):
        '''
        Called when the websocket is handshaking as part of initial connection.

        '''

        # Check device name from url
        self.deviceName = self.scope['url_route']['kwargs']['deviceName']
        self.roomName = 'room_%s' % self.deviceName
```

Ilustración 6-3. Parte de "devicesConsumer.py".

En un principio, las conexiones webSocket están ideadas para conexiones entre el navegador y el servidor, aunque permite conexiones externas de igual forma. Sin embargo, éstas últimas, al no encontrarse dentro del entorno de la aplicación, sino que provienen de dispositivos externos, conectados a través de internet, han requerido añadir un sistema de identificación y seguridad para evitar intrusiones tanto en extremo del servidor Django como en el extremo del robot.

Para ello, y usando el paquete "*security*", explicado más adelante, se ha ideado un sencillo método que permite verificar la identidad de ambos extremos de la comunicación, así como establecer una comunicación cifrada de forma segura y dinámica, tal y como se puede observar en la ilustración 5-4.

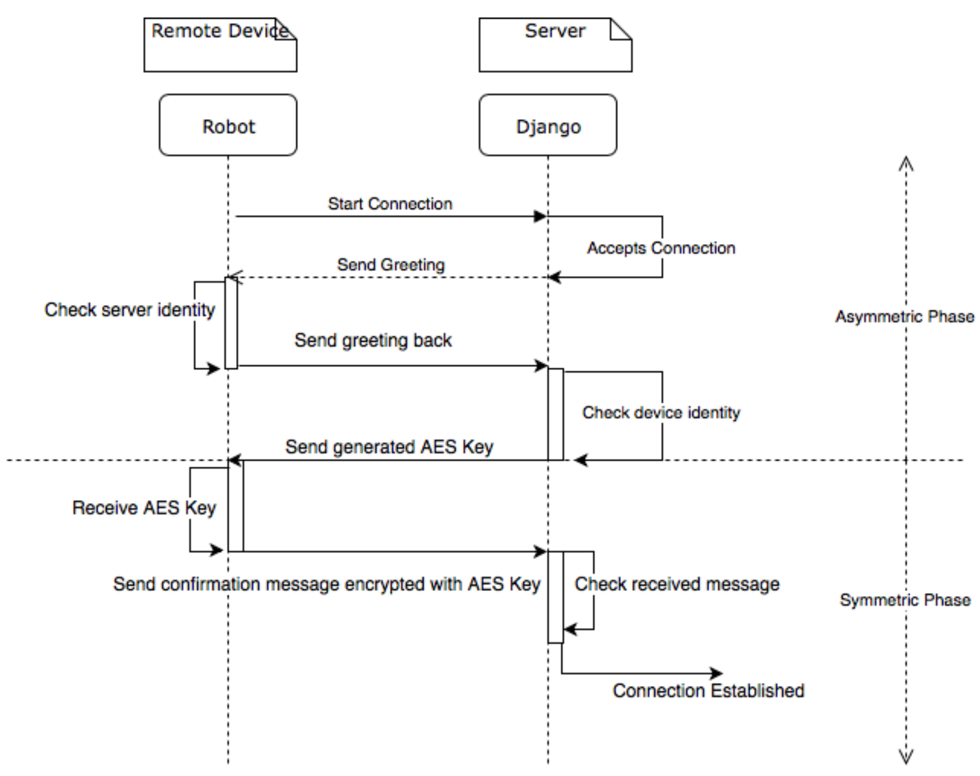

Ilustración 6-4. Diagrama de conexión remota.

Esta conexión se basa en una comunicación simétrica/asimétrica, donde el servidor y el dispositivo remoto disponen de su par propio de claves pública/privada y la clave pública del otro dispositivo (el servidor debe disponer de todas las claves públicas de los dispositivos remotos que quieran conectarse a él).

Como se puede ver en el diagrama de la conexión remota, en cada conexión el servidor crea una clave aleatoria nueva, concretamente AES-256, que posteriormente comparte con el dispositivo remoto en cuestión para establecer la conexión simétrica.

Una vez finalizada la comunicación, dicha clave se pierde. Este sistema, aunque sencillo, permite crear un método seguro de transmisión de datos entre ambas partes, necesario al usar internet como medio.

6.1.2.3 Exceptions

Este pequeño módulo contiene clases que heredan de "*Exception*" y permiten distinguir el tipo de excepción que se ha producido. Las clases en sí son sumamente simples, como se puede apreciar en la ilustración 5-5.

```
'''
To manage authentication errors
'''

class AuthError(Exception):

    def __init__(self, code):
        super().__init__(code)
        self.code = code
```

Ilustración 6-5. Ejemplo de una de las clases para manejar excepciones.

6.1.2.4 Definitions

Como se puede intuir por el título, este paquete contiene las definiciones de las variables globales usadas en la aplicación. Por un lado, se especifican los códigos de cierre de conexión del servidor webSocket, que permiten especificar los diferentes motivos que han llevado a dichos cierres.

Y, por otro lado, se encuentran las definiciones de los diferentes tipos de mensajes *json* usados en la comunicación. La mayoría usados en el proceso de establecimiento de la misma.

6.1.2.5 Database

Para regularizar y controlar los diferentes aspectos de la aplicación, se ha optado por implementar una sencilla base de datos que, nutriéndose de la propia base de datos interna de Django que gestiona los usuarios, permite tratar de igual modo con los dispositivos registrados y su uso.

Tal y como podemos ver en el esquema de la base de datos creada (Ilustración 5-6), partiendo de la tabla "*Users*", gestionada internamente por Django, se encuentra un registro de actividad de los usuarios, concretamente de cada log.

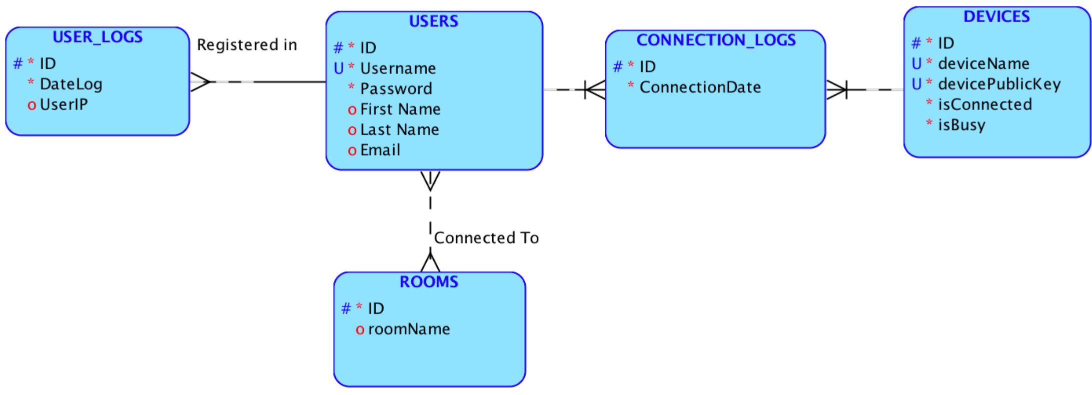

Ilustración 6-6. Base de datos de la aplicación Django.

Por otro lado, se tiene un registro de los dispositivos, de tal forma que cualquier dispositivo no registrado no puede tener acceso al servidor, ya que en dicha tabla se almacenan datos como su clave pública, necesaria para la comunicación asimétrica.

Además, se lleva un control de conexiones de los usuarios con los diferentes dispositivos, así como de las diferentes salas que se crean dinámicamente, y de los usuarios contenidos en ellas.

Para gestionar la base de datos, se ha creado una clase por cada tabla que contiene los métodos necesarios para realizar la inserción, actualización y eliminación de datos en un fichero llamado "*manager.py*".

6.1.2.6 Security

Como se mencionó con anterioridad en el apartado de "*Sockets*", el paquete "*Security*" se encarga de gestionar la encriptación y verificación de los mensajes intercambiados con el usuario remoto. En este apartado se encuentran varias clases que engloban las diferentes funcionalidades necesarias para llevar a cabo la comunicación. Todas estas clases usan como base el paquete Python "*pycryptodome*" [19], que facilita la interacción con claves simétricas y asimétricas.

En primer lugar, el fichero "asymmetric.py" contiene los métodos necesarios para firmar, encriptar, desencriptar y verificar firmas con claves públicas y privadas, permitiendo llevar a cabo la comunicación de forma asimétrica que, como se explicó con anterioridad, constituye la primera etapa de la conexión. Para intercambiar mensajes, tantos simétricos como asimétricos, se ha establecido como estándar los mensajes de tipo *json*, que permiten añadir tantos campos como el mensaje requiera (Ilustración 5-7).

```
JSON Message Format:
    encrypted message = {'coded':'encrypted data here', 'codedSign': 'Encrypted data sign here'}
    decoded_data = {'type': 'type here', 'message': 'message here'}
```

Ilustración 6-7. Mensajes JSON usados en la encriptación de los datos.

Por otro lado, el archivo "*symmetric.py*" contiene métodos muy similares a los definidos anteriormente, pero aplicados al intercambio de mensajes cifrados de forma simétrica. Para este tipo de cifrado se requiere una clave simétrica, solicitada para

crear el objeto de esta clase. En este caso se ha optado por una clave AES-256, que ofrece un alto nivel de seguridad, y es generada de forma dinámica cada vez que se establece una nueva comunicación. Además, se utiliza un encriptado *EAX* [20], que añade otra capa de seguridad verificando, mediante un hash, que el fichero encriptado no ha sido alterado por un tercero.

Por último, este paquete añade un módulo que, aunque no se usa en el programa de forma directa, sí es necesario. Concretamente, se hace referencia al fichero "*createPublicKeys.py*" cuya misión es crear pares de claves pública/privada. De esta forma, se puede invocar esta clase de forma independiente para crear las claves que sean necesarias.

6.1.2.7 Static Files

Este último bloque contiene, como su propio nombre indica, los ficheros estáticos de la aplicación, principalmente usados por la capa del navegador. Esta organización de los ficheros permite a Django y, por consiguiente, al navegador, acceder de forma sencilla a los archivos.

En efecto, desde el propio navegador, se puede acceder a estas carpetas haciendo únicamente referencia a "*Static*", sin necesidad de usar el path del fichero completo dentro de la aplicación y/o el sistema (como se puede apreciar en la Ilustración 5-8). Pueden encontrarse principalmente ficheros propios de programación web, tales como archivos JavaScript, css, imágenes usadas en la web, etc.

```
        <!-- Add children content -->
        <div class="container body-content" style="padding-top: 80px" id="content"> <!-- Added a top padd
            {% block content %} {% endblock %} <!-- Includes block content from htmls -->
            <hr/>
            <footer class="text-center">
                <p>&copy; {{ year }} - ROS WebRTC Bridge</p>
            </footer>
            </div>
        <script src="{% static 'javascript/jquery-3.3.1.js' %}"></script>
        <script src="{% static 'javascript/jquery-3.3.1.min.js' %}"></script>
        <script src="{% static 'javascript/popper.js' %}"></script>
        <script src="{% static 'bootstrap/js/bootstrap.min.js' %}"></script>
        {% block endScripts %} {% endblock %}
    </body>
```

Ilustración 6-8. Ejemplo de uso de ficheros estáticos (extracto de template.html).

6.2 Aplicación WebRosTC

En el otro extremo de la comunicación, gestionando el sistema robot, se encuentra una aplicación escrita en Python. Esta aplicación está compuesta por varios módulos y submódulos que se detallarán a continuación. La razón por la cual se ha elegido desarrollar esta aplicación en Python y no otro lenguaje viene, como ya se ha expuesto anteriormente, determinado por varios factores:

- Inicialmente se ha tratado de desarrollar esta misma aplicación en C++, sin éxito.
- ROS trabaja con C++ y Python únicamente.
- El paquete usado para interactuar con WebRTC se ha desarrollado en Python.
- Algunas librerías usadas en el servidor Django se han podido reutilizar en esta aplicación (tal es el caso de la librería "*security*").

6.2.1 Nodo Bridge

Tal y como ya se pudo apreciar en la Ilustración 3-1, el nodo "*bridge*" es el encargado de gestionar el establecimiento de la conexión, así como de enviar y recepcionar la información. Este nodo cuenta con diversos módulos, con el fin de tratar correctamente tanto el vídeo y el audio como los datos, tanto locales como remotos, así como de gestionar el enlace con el usuario remoto. Los módulos son los siguientes:

- Connections
- Definitions
- MediaManager
- WebRTC
- Security

6.2.1.1 Connections

Este primer módulo es el encargado de gestionar las comunicaciones del nodo en general. Por un lado, se encarga de establecer la conexión inicial con el servidor webSocket remoto, con el fin de intercambiar la información necesaria con el otro usuario.

Y, por otro lado, dada la incompatibilidad entre el nodo ROS ejecutándose en Python 2.7 y este nodo ejecutándose en Python 3.5, ha sido necesario implementar un sencillo canal de comunicación UDP interno, que se comunica con el nodo ROS. Veámoslos con más detalle:

El submódulo encargado de la comunicación con el servidor webSocket, "*signaling.py*", se basa en la creación de un cliente webSocket haciendo uso del paquete "*webSockets".* Al igual que en el lado del servidor, ha sido preciso implementar un sistema seguro de comunicación basado en cifrado simétrico/asimétrico, como ya se puedo representar en la Ilustración 5-4. De esta forma, este paquete gestiona (junto con el módulo de *Security)* la encriptación y desencriptación de paquetes.

Asimismo, el encargado de conectar con el nodo ROS en otra versión de Python, *"channel.py*", está implementado mediante sockets (Ilustración 5-9). Se crea un socket para enviar la información y otro para recibirla, trabajando cada uno en un puerto diferente. De esta forma se evitan conflictos entre sockets.

6.2.1.2 Definitions

De igual forma que en el servidor, en el nodo *bridge* se ha definido los tipos de cierre de la conexión con el webSocket en el módulo "*closeCodes.py"*, con el fin de identificar las diferentes razones por la que se puede cerrar el enlace.

Por otro lado, se han definido los tipos de mensajes *Json* que se pueden enviar y recibir en el nodo (Ilustración 5-10). Concretamente, los diferentes mensajes del servidor webSocket.

```
'''
To allow message exchanging, sender and receivers sockets must be on different ports
'''
# UDP Socket
import socket

class Channel():

    def __init__(self, name):
        self.server_address = '127.0.0.1'
        if name == 'Client':
            self.sender_port = 5678
            self.receiver_port = 8765
        else:
            self.receiver_port = 5678
            self.sender_port = 8765

        self.buffer = 1024
        self.name = name

        # Creating UDP socket
        self.receiver = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        # Make socket reusable
        self.receiver.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        # Allow incoming broadcast
        self.receiver.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
        # Set socket to non-blocking mode
        self.receiver.setblocking(False)

        self.sender = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Ilustración 6-9. Inicialización de los sockets (extracto de channels.py).

```
'''
Enumeration class with different used json types
'''
from enum import Enum

class JsonMessages(Enum):

    # WebRTC
    WEBRTC_MESSAGE = 'webrtc_message'

    SDP_OFFER = 'offer'
    SDP_ANSWER = 'answer'

    CANDIDATE = 'candidate'

    # WebSocket HeartBeat
    PING = 'ping'
    PONG = 'pong'

    # Authentication
    GREETING = 'greeting'
    KEY = 'key'
    AES_CONFIRMATION = 'AESConfirmation'

    # Disconnect
    LEAVING = 'leaving'
```

Ilustración 6-10. Diferentes tipos de mensajes Json usados (extracto de jsonMessages.py).

6.2.1.3 MediaManager

Al exponer inicialmente las diferentes tecnologías usadas en el proyecto, y concretamente la de ROS, se explicó que se habían optado por implantar de forma independiente submódulos que gestionaran el vídeo y el audio local y remoto, en vez de usar los proporcionados por ROS.

Una de las razones por la cual se optó por esta solución fue la de la eficiencia, al ahorrarse un paso de procesamiento de los datos. Otra fue por la facilidad de tratar con los datos que ofrecía el módulo de *aiortc* [15]. Veamos el por qué:

6.2.1.3.1 Módulo de Vídeo

En primer lugar, la gestión del vídeo, tanto local como remoto es llevada a cabo por "*videoHandler.py*". Para tratar con los datos de vídeo remotos, ha sido necesario analizar la trama de video que gestiona *aiortc*.

Concretamente, los paquetes de vídeo entrante son tratados como un objeto de la clase *VideoFrame,* la cual contiene:

- Data (bytes): correspondientes a un *frame* de vídeo. El formato del vídeo se corresponde con el *YUV420* [7].
- La altura de la imagen.
- La anchura de la imagen.
- Implícitamente, por el número de bytes por *frame*, así como por la resolución, se obtiene que cada píxel está definido por 1,5 bytes.

Con esta información, a partir del *array* de bytes recibido, se reconstruye la matriz de la imagen. Dado que, para mostrar la imagen por pantalla usando openCV [8], éste requiere una imagen en formato *BGR* (o *RGB*), se transforma la imagen de YUV a BGR mediante *openCV* para posteriormente mostrarla.

Por tanto, con una simple reconstrucción de los datos y conversión de formato, se obtiene la imagen remota deseada (Ilustración 5-11).

```python
async def handleIncomingVideo(track):
    '''
        Handles incoming video
        track.recv() returns an aiortc.mediastreams.videoframe object with:
        - data (bytes) -> Video frame in YUV420 format
        - height
        - width

        Get remote video data and displays it

    '''
    print('Handling incoming video...')

    while True:
        # Get video from remote peer
        video = await track.recv()
        # Transform it into BGR video
        frame = dataInYUVToBGR(video.data, video.width, video.height)
        # Displays it using OpenCV
        cv2.imshow('Remote Peer Video', frame)
        if cv2.waitKey(5) & 0xff == ord('q'):
            cv2.destroyAllWindows()
            break
```

Ilustración 6-11. Mostrando vídeo mandado por el peer remoto (extracto de videoHandler.py).

En el sentido opuesto, se encuentra el gestor del vídeo local. Dado a que el paquete que acepta *aiortc* es, al igual que el entrante, de tipo "*VideoFrame*", el proceso es muy similar, aunque a la inversa. (Ilustración 5-12).

Inicialmente, es preciso obtener el vídeo (*frame a frame*) de la cámara correspondiente. Afortunadamente, openCV facilita enormemente dicha tarea, aportando un sencillo método que accede al dispositivo de vídeo y obtiene los *frames (cv2.VideoCapture())*.

```python
    async def recv(self):
        if self.capture.isOpened():
            #self.last = await self.pause(self.last, 0.04)
            # Get frame by frame
            ret, frame = self.capture.read()

            # aiortc works with YUV420 video, so frame must be transformed
            yuv = cv2.cvtColor(frame, cv2.COLOR_BGR2YUV_I420)

            # Transform matrix data to byte array
            data = np.array(yuv, dtype = frame.dtype).tobytes()
            return VideoFrame(width = int(self.capture.get(cv2.CAP_PROP_FRAME_WIDTH)),
                              height = int(self.capture.get(cv2.CAP_PROP_FRAME_HEIGHT)),
                              data = data)
```

Ilustración 6-12. Captura y encapsulamiento del vídeo (extracto de videoHandler.py).

Una vez superada la barrera de obtener la información, ya es cuestión de adaptar los datos al formato necesario para *aiortc.* Es decir, transformar la imagen obtenida de la cámara, transformarla a formato *YUV420* y transformar la matriz a un *array* de *bytes*. Por último, se encapsulan los datos, así como los parámetros de altura y anchura de la imagen en un objeto *VideoFrame* y se manda al peer remoto.

6.2.1.3.2 Módulo de Audio

La gestión tanto del vídeo entrante como saliente la podemos encontrar en el módulo "*audioHandler.py*". Al igual que en el caso del vídeo, *aiortc* dispone de una clase específica para el audio, tanto entrante como saliente. Concretamente, la clase *AudioFrame*, que se caracteriza por:

- Número de canales del audio
- Datos: array de bytes (PCM – 16)
- Sample-rate (44100, 48000, etc.)
- Sample-width (16 bits, 24 bits, etc.)

Para tratar correctamente los datos entrantes y encapsular los datos salientes, se ha recurrido al paquete "*pyAudio*" [10] que, al igual que openCV, permite interactuar de forma sencilla con los elementos hardware.

En primer lugar, para tratar los datos entrantes y reproducirlos por cualquier medio del robot (ya sea un altavoz o una salida de audio), tan sólo ha sido necesario abrir un *streaming* de audio con las características del audio entrante. Los datos necesarios para abrirlo, como el formato, los canales o el *sample rate* son proporcionados por la propia clase *AudioFrame.* Una vez abierto, cada paquete de datos recibidos se transmite al *stream* para ser reproducido (Ilustración 5-13).

Por otro lado, para obtener los datos locales de audio (como, por ejemplo, los de un micrófono, y encapsularlos para poder mandarlos a través de *aiortc, pyAudio* vuelve a simplificar enormemente el proceso:

1. Se especifican inicialmente los parámetros del audio que se va a enviar, y se abre el *streaming* de audio.

2. Posteriormente se crea un nuevo objeto de la clase *AudioFrame* con los parámetros anteriormente especificados, y se leen los datos del *stream* (Ilustración 5-14).

```python
async def handleIncomingAudio(track):
    '''
        Handles incoming audio
        16-bit (2 bytes width) PCM audio
        track.recv() returns an aiortc.mediastream.audioframe object with:
        - channels
        - data
        - sample-rate: 44100, 48000...
        - sample-width (2)
    '''
    print('Handling incoming audio...')
    try:
        # Instantiate pyAudio
        audio = pyaudio.PyAudio()

        audioFrame = await track.recv()
        # Open stream
        stream = audio.open(format = audio.get_format_from_width(audioFrame.sample_width),
                            channels = audioFrame.channels,
                            rate = audioFrame.sample_rate,
                            output = True)

        while True:
            audioFrame = await track.recv()
            stream.write(audioFrame.data)

    finally:
        stream.stop_stream()
        stream.close()
        audio.terminate()
```

Ilustración 6-13. Reproduciendo audio remoto (extracto de audioHandler.py).

```python
async def recv(self):
    '''
        Get audio from stream (microphone) and return it as audio frame
    '''
    self.last = await self.pause(self.last, self.pauseTime)

    return AudioFrame(
        channels = self.channels,
        data = self.stream.read(self.buffer),
        sample_rate = self.framerate
    )
```

Ilustración 6-14. Capturando audio del micrófono y encapsulándolo (extracto de audioHandler.py).

6.2.1.4 WebRTC

De entre los diferentes módulos que componen este nodo, podría decirse que éste es el principal, pues agrupa prácticamente a todos los módulos anteriores y conforma los procedimientos para llevar a cabo la comunicación directa con el otro peer de forma apropiada.

Tomando como base el paquete *aiortc*, este módulo es el encargado de intercambiar la información necesaria con el peer remoto para establecer la comunicación p2p. Además, gestiona la comunicación con el nodo ROS mediante el paquete *channel*, tratando de esta forma con los datos entrantes y salientes.

El procedimiento para establecer la conexión con el sistema remoto es muy similar a la ya vista en el lado del navegador. Inicialmente se crea un peer y se le añaden los recursos locales. Además, se crea el *dataChannel* y los eventos para manejar los recursos remotos. (Ilustración 5-15).

```
async def createPeer(self):
    '''
        Creates a new peer, with audio, video and data handlers
    '''
    # Creating local peer
    self.localPeer = RTCPeerConnection()

    # Creating data channel
    self.dataChannel = self.localPeer.createDataChannel('data')

    # Data channel event handlers
    @self.dataChannel.on('message')
    def on_message(message):
        print('Received message via dataChannel: ' + str(message))
        # Sending message through internal channel
        self.internalChannel.send(message)

    # Creating local medias (audio and video)
    self.localAudio = AudioHandler()
    self.localVideo = VideoHandler()
```

Ilustración 6-15. Creación del peer, dataChannel y sus manejadores (extracto de webrtc.py).

Sin embargo, en este caso, el robot, una vez conectado al servidor webSocket remoto, se queda a la espera de recibir cualquier oferta de cualquier peer que requiera conectarse.

Una vez recibida la oferta, el robot la procesa y almacena como definición remota. Luego crea una respuesta, que almacena como definición local y se la manda al peer remoto. Como ya se comentó durante la explicación del proceso del establecimiento de conexión en el navegador, dado que el paquete *aiortc* se encuentra aún en desarrollo, ha sido necesario repetir el proceso de almacenar la oferta remota y crear una nueva, ya que es en la segunda oferta donde el *peer* remoto incluye "*ICE candidates*" en la oferta. (Ilustración 5-16).

```
async def handleRemoteOffer(self, remoteOffer):
    '''
        Set remoteOffer as remote Description, creates an answer,
        set it as local description and send it to remote peer.
    '''
    print('Remote offer received. Handling...')
    #print(str(remoteOffer))
    # Set remoteOffer as remoteDescription
    await self.localPeer.setRemoteDescription(RTCSessionDescription(**remoteOffer))

    # Now creates an answer
    answer = await self.localPeer.createAnswer()
    #print('Answer: ' + str(answer))

    # Setting answer as local description
    await self.localPeer.setLocalDescription(answer)

    # Send answer to remote peer
    print('Sending answer to remote peer')
    await self.signaling.send({
        'type': 'answer',
        'data': {
            'type': answer.type,
            'sdp': answer.sdp
        }
    })

    # For debug
    # asyncio.ensure_future(self.statusChecker())
```

Ilustración 6-16. Manejo de oferta remota y creación de respuesta (extracto de webrtc.py).

Tan pronto como se agregue la funcionalidad de agregar "*ICE Candidates*" de forma dinámica al peer, este "arreglo" no será necesario, a la vez que asegurará la estabilidad de la aplicación.

Como se puede apreciar, tanto el vídeo como el audio cuentan con métodos independientes que lo gestionan. Sin embargo, en el caso de los datos locales/remotos, son gestionados desde este mismo módulo, enlazando los eventos de datos con el canal local creado para comunicarse con el nodo ROS (Ilustración 5-17).

```
async def internalChannelHandler(self):

    while True:
        if not self.localPeer: break

        if self.dataChannel and self.dataChannel.readyState == 'open':
            message = self.internalChannel.recv()
            if message:
                # Send message through data channel
                self.dataChannel.send(message)

        await asyncio.sleep(0.1)
```

Ilustración 6-17. Gestor de mensajes entre channel interno y dataChannel (extracto de webrtc.py).

6.2.1.5 Security

Al igual que en el servidor, el paquete "*Security*" se encarga de gestionar el manejo de claves simétricas y pública/privada. De hecho, se trata del mismo paquete, ya que tanto el sistema de cifrado/descifrado como la estructura de los mensajes deben ser idénticos. En el caso del robot, dispone en una carpeta de las claves pública y privada propias, así como de la clave pública del servidor.

De esta forma, al inicializar el módulo de seguridad asimétrica (Ilustración 5-18), automáticamente se cargan las claves necesarias para firmar y desencriptar (clave privada) y para encriptar y verificar la firma remota (clave pública del servidor).

```
def __init__(self):
    '''
    Load own private/public keys and remote public key

    Raises file not found exception if files do not exist
    '''
    try:
        self.privateDeviceKey = RSA.import_key(open('keys/privateDeviceKey.pem').read())
        #self.publicDeviceKey = RSA.import_key(open('keys/publicDeviceKey.pem').read())
        self.publicServerKey = RSA.import_key(open('keys/publicServerKey.pem').read())
    except FileNotFoundError as err:
        print('Error opening public keys: ' + str(err))


def signMessage(self, message):
    '''
    Sign message using PKCS1_PSS Signature
    Using SHA3_512 as well
    '''
    if isinstance(message, bytes) or isinstance(message, bytearray):
        # Generate message hash object
        messageHash = SHA3_512.new(message)
        # Return signature
        return pss.new(self.privateDeviceKey).sign(messageHash)
    else:
        raise Exception('Error creating message signature: message MUST be either bytes or bytearray type')
```

Ilustración 6-18. Constructor y método para firmar mensajes (extracto de asymmetric.py).

6.2.2 Nodo ROS

Conjuntamente con el nodo bridge que se acaba de exponer, se encuentra el nodo ROS, que se integra dentro de la arquitectura ROS y hace de enlace con el nodo bridge. Tal y como se explicó en el apartado 3, el sistema ROS [2] consiste en un *framework* que permite la utilización de diversas herramientas de forma simultánea y comunicarlas entre sí mediante *topics*.

El propósito de esta herramienta no es más que comunicar los *topics* necesarios con el peer remoto, con el fin de, por un lado, controlar los motores del robot y, por otro lado, obtener información del robot como, por ejemplo, la velocidad de cada uno de los motores o las lecturas del sensor de ultrasonidos.

Para el ejemplo de robot actual, se ha optado por usar un robot lego NXT (Ilustración 5-19), por las facilidades que aporta para programarlo y por los diversos actuadores y sensores que dispone. Por tanto, para adaptar el nodo ROS al esquema de este tipo de robot, se han requerido ciertas herramientas adicionales, que solventarán la capa entre el software y el propio robot.

Ilustración 6-19. Robot Lego MindStorms NXT.

Concretamente, se han utilizado las librerías "*NXT-Python"* [18] (que en este momento sólo dispone de librerías para Python 3) y "*NXT ROS – Core"* [17] (diseñado para *ROS Kinetic*).

NXT-Python tiene como finalidad crear una capa invisible para el desarrollador, haciendo de intermediaria entre el hardware del NXT y el software. Permite gestionar los diferentes puertos de conexión del NXT, configurarlos para usarlos según convenga, etc.

Por otro lado, *NXT ROS-Core,* recurriendo a al paquete anterior, transforma el robot NXT en un nodo más de ROS, definiendo unos *topics* concretos para cada sensor, *encoder* y motor. Además, establece los diferentes tipos de mensajes necesarios para usar en cada *topic.*

Cabe aclarar que ROS ya contiene un paquete encargado de gestionar los robots NXT, llamado de esa forma, "*nxt".* Sin embargo, este paquete únicamente tuvo soporte para la versión *electric* de ROS, una versión del 2011, razón por la cual se ha optado por una herramienta más actualizada.

Por tanto, conociéndose qué *topics* son los que se corresponden con cada sensor, motor, etc., tan sólo es cuestión de publicar y suscribirse a los *topics* adecuados. Y ésta es precisamente la misión del nodo ROS creado. Para construir este nodo, inicialmente ha sido necesario (aparte de instalar el propio motor ROS) crear una carpeta que va a funcionar como entorno de trabajo (la famosa carpeta "catkin_ws").

En la carpeta "*src*" se encuentran todos los nodos del entorno de trabajo. Por tanto, por un lado, es necesario añadir el nodo *NXT-ROS Core* a dicha carpeta. (Ilustración 5-20). Por otro lado, se añade el nodo ROS que se ha implementado acorde con las especificaciones necesarias para este proyecto, el cual está compuesto por:

- Código fuente: escrito en Python, contiene la funcionalidad del nodo propiamente dicha.
- Fichero de configuración: contiene la configuración aplicada al robot NXT sobre el que se va a ejecutar.
- Fichero de ejecución: indica a ROS cómo ejecutar el/los nodo/s, así como los parámetros necesarios para su ejecución.
- Dependencias y compilación

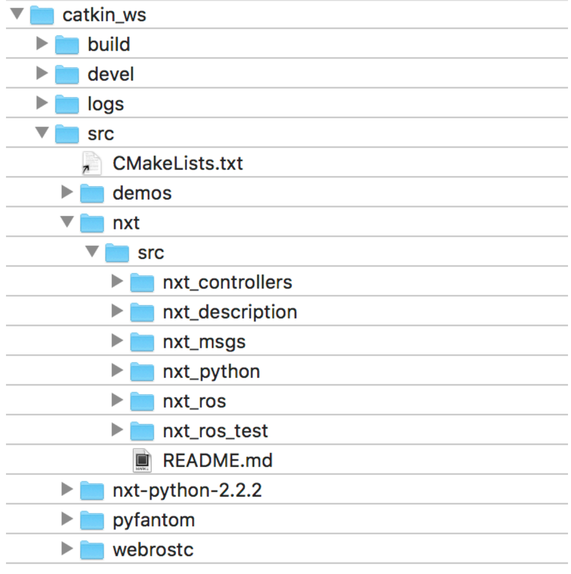

Ilustración 6-20. Contenido de la carpeta "catkin_ws", donde se encuentran los nodos ROS.

6.2.2.1 Código Fuente

Para acceder a los diferentes *topics* necesarios del robot, es necesario crear un nodo, definiendo un "*Publisher*" y un "*Subscriber*". El *Publisher* tiene como misión controlar los motores del NXT enviando instrucciones de potencia a los mismos. (Ilustración 5-21).

```python
def start(self):
    '''
        Start publisher node and subscribers
    '''
    # Defining publishers topics and message types
    print('Starting publisher node...')
    self.publisherLeft = rospy.Publisher(self.motorsTopic, JointCommand, queue_size = 10)
    self.publisherRight = rospy.Publisher(self.motorsTopic, JointCommand, queue_size = 10)

    # Defining subscribers
    self.motorsSubscriber = rospy.Subscriber('joint_states', JointState, self.motorsCallback)

    self.sensorSubscriber = rospy.Subscriber('ultrasonic_sensor', Range, self.sensorCallback)

    rospy.init_node('publisher', anonymous = True)
    # Start channel
    self.channel.startChannel()
    rate = rospy.Rate(self.rate)

    while not rospy.is_shutdown():
        # Check channel messages
        command = self.channel.recv()
        # If data received...
        if command:
            self.setMovement(command)
        self.publisherLeft.publish(self.left)
        self.publisherRight.publish(self.right)
        rate.sleep()
```

Ilustración 6-21. Inicio del nodo 'publisher' para comunicarse con el robot (extracto de topicsHandler.py).

Como ya se ha comentado anteriormente, para solventar la incompatibilidad de los programas por la versión de Python en la que se ejecutan, se ha implementado un sencillo canal UDP basado en sockets. Por tanto, en este nodo ha sido necesario implementar el otro extremo del canal. De esta forma, los mensajes recibidos por el canal se clasifican y, en función del mensaje, se envía la correspondiente instrucción por el *topic* adecuado.

Para facilitar la tarea, se ha creado una clase adicional auxiliar "*Movements*" (Ilustración 5-22), que traduce la instrucción en el mensaje aceptado por el *topic*, con la referencia al motor correspondiente, así como la potencia necesaria para llevar a cabo tal instrucción.

```
class Movements():
    '''
        Defines all possible robot movements
    '''
    def __init__(self):
        self.leftMotor = 'left_motor'
        self.rightMotor = 'right_motor'
        # Left motor command
        self.left = JointCommand()
        self.left.name = self.leftMotor
        # Right motor command
        self.right = JointCommand()
        self.right.name = self.rightMotor

    def goForward(self):
        '''
            Both motors with same power
        '''
        self.left.effort = 10.00
        self.right.effort = 10.00
        return (self.left, self.right)

    def goBack(self):
        self.left.effort = -10.00
        self.right.effort = -10.00
        return (self.left, self.right)

    def goRight(self):
        self.left.effort = 10.00
        self.right.effort = 5.00
        return (self.left, self.right)

    def goLeft(self):
        self.left.effort = 5.00
        self.right.effort = 10.00
        return (self.left, self.right)

    def stop(self):
        self.left.effort = 0.00
        self.right.effort = 0.00
        return (self.left, self.right)
```

Ilustración 6-22. Clase auxiliar para manejar los comandos entrantes (extracto de topicsHandler.py).

Por otro lado, se ha creado un *subscriber* por cada *topic* que se quiere escuchar. Concretamente, se han necesitado tres: uno por cada *encoder* de los motores y un tercero para el sensor de ultrasonidos.

Para tratar la información y enviarla por el mismo canal, cada método "*callback*" recoge la información respectiva a su *topic*, la encapsula en un archivo Json y la manda por el canal UDP para que el nodo ROS lo reenvíe al peer remoto. (Ilustración 5-23).

```python
def motorsCallback(self, data):
    # MotorCounter is an integer counter. Divider is set to 5 in init
    if self.motorCounter is self.divider:
        print('Left Motor Velocity: ' + str(round(data.velocity[0],2)))
        print('Right Motor Velocity: ' + str(round(data.velocity[1],2)))

        # Send it through internal channel
        message = json.dumps({
            'dataType': 'motor_info',
            'message': {
                'left_motor': str(round(data.velocity[0],2)),
                'right_motor': str(round(data.velocity[1],2))
            }
        })
        self.channel.send(message)
        self.motorCounter = 0
    self.motorCounter = self.motorCounter + 1
```

Ilustración 6-23. Captura de uno de los dos subscribers (extracto de topicHandler.py).

6.2.2.2 Fichero Configuración

Dado que el software que controla el robot NXT necesita conocer las especificaciones concretas del robot que se va a usar, es necesario construir un fichero de configuración con extensión "*.yaml*" en el que se declaran los componentes exactos que contiene el robot, así como la propia configuración de éstos.

El fichero de configuración debe tener un formato específico. Afortunadamente, el propio ROS nos proporciona una guía [21] en la que nos indica, entre otras cosas, cómo crear el fichero.

Por tanto, dado que el NXT usado en este caso posee dos motores, el izquierdo en el puerto B y el derecho en el puerto C, un sensor de ultrasonidos en el puerto 4 y un sensor de luz en el puerto 1, el archivo de configuración es el que podemos ver en la ilustración 5-24.

```yaml
nxt_robot:
  - type: motor
    name: left_motor
    port: PORT_B
    desired_frequency: 10.0

  - type: motor
    name: right_motor
    port: PORT_C
    desired_frequency: 10.0

  - type: touch
    frame_id: touch_sensor_frame
    name: my_touch_sensor
    port: PORT_1
    desired_frequency: 10.0

  - type: ultrasonic
    frame_id: light_sensor_frame
    name: ultrasonic_sensor
    spread_angle: 30.0
    min_range: 0.07
    max_range: 2.54
    port: PORT_4
    desired_frequency: 10.0
```

Ilustración 6-24. Configuración del robot NXT (extracto de nxt_robot.yaml).

6.2.2.3 Fichero de ejecución

Por regla general, cuando se desea ejecutar un nodo ROS, basta con ejecutar en consola el comando *rosrun* + *nombre_paquete* + *nombre ejecutable.* Sin embargo, cuando se requiere ejecutar más de un nodo de forma conjunta, este método no es válido. Es necesario definir un fichero de ejecución con extensión "*.launch*" de estructura similar al *xml,* donde se especifican los diferentes nodos que se van a ejecutar, así como los diferentes parámetros que éstos requieran, como la ubicación de archivos necesarios, o valores de variables.

Para la ejecución de este nodo en conjunto, se ha diseñado un fichero de ejecución que lanza el nodo "*nxt-ros*" que controla el robot con el fichero de configuración visto

en el apartado anterior, el nodo “joint_state_publisher”, encargado de gestionar los topics del robot, y el nodo explicado en el apartado 5.2.2.1, “webrostc”. La Ilustración 5-25 muestra el fichero resultante.

```
<!-- Launch file for WebRosTC project -->

<launch>
  <!-- Defining nodes... -->
  <node pkg="nxt_ros" type="nxt_ros.py" name="nxt_ros" output="screen" respawn="true">
    <rosparam command="load" file="$(find webrostc)/config/nxt_robot.yaml"/>
  </node>
  <node pkg="nxt_ros" type="joint_states_aggregator.py" name="joint_state_publisher" output="screen"/>
  <node pkg="webrostc" type="webrostc.py" name="WebRosTC" output="screen"/>
</launch>
```

Ilustración 6-25. Fichero de lanzamiento de los nodos ROS correspondientes (extracto de webrostc.launch).

6.2.2.4 Dependencias y compilación

Todos los nodos ROS pertenecientes a un entorno de trabajo se compilan de forma conjunta mediante la herramienta ROS “*catkin_make*” o “*catkin build”.* Sin embargo, como es normal, cada nodo requiere de unos paquetes específicos, librerías, etc. Por ello, es necesario especificar qué dependencias tiene cada nodo, y cómo debe compilarlo el compilador de ROS.

En un primer momento, al crear un nodo con ROS mediante el comando “*catkin_create_pkg”,* posteriormente se indica, además del nombre del nodo, las dependencias que va a tener, como explica la propia guía de ROS para la creación de nodos [22]. Por ejemplo, “*catkin_create pkg example_node std_msgs rospy roscpp”*. Esta instrucción crea, aparte de las carpetas correspondientes del paquete, unos ficheros que especifican dichas dependencias. Concretamente, los ficheros “CMakeLists.txt” y “package.xml”.

El fichero *package.xml* permite especificar parámetros como el nombre del paquete, su versión o las dependencias necesarias para compilar, exportar y ejecutar el paquete. La configuración del *package.xml* específica para este nodo se puede observar en la Ilustración 5-26.

```xml
<?xml version="1.0"?>
<package format="2">
  <name>webrostc</name>
  <version>0.0.0</version>
  <description>The webrostc package</description>

  <maintainer email="ubuntu@todo.todo">ubuntu</maintainer>

  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>nxt_ros</build_depend>
  <build_depend>nxt_msgs</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>nxt_ros</build_export_depend>
  <build_export_depend>nxt_msgs</build_export_depend>
  <build_export_depend>sensor_msgs</build_export_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>nxt_ros</exec_depend>
  <exec_depend>nxt_msgs</exec_depend>
  <exec_depend>sensor_msgs</exec_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>


  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

Ilustración 6-26. Captura del fichero package.xml con las dependencias del nodo webrostc.

Por otro lado, el fichero *CMakeLists.txt* constituye una “guía” para el compilador *cmake,* en el cual le indica cómo compilar y construir los paquetes.

Estos ficheros siguen un formato específico, en el cual se indican parámetros como la versión mínima necesaria del compilador para compilar dicho paquete, el nombre del paquete, las dependencias o el fichero que contiene la rutina principal (*main*), entre otros.

En este caso el orden de los parámetros sí debe ser tenido en cuenta. Para aclarar las dudas de cualquier usuario que desee crearse un fichero *CMakeLists* personalizado a sus necesidades, existen múltiples tutoriales que explican paso a paso como construir dichos ficheros, tales como la propia guía de *CMake* [23] o la propia guía de ROS para configurar correctamente los ficheros *CMakeLists.txt* [24].

Normalmente, al compilar un nodo escrito en C++, es necesario especificar en "add_executable", el fichero cpp que contiene el main. Sin embargo, este nodo está escrito en Python, y esto cambia ligeramente la forma en la que el compilador debe trabajar con el nodo. Veamos cómo:

1. Para comenzar, es necesario especificar que se trata de un programa en Python en el propio fichero *CMakeLists.txt* añadiendo la línea "*catkin_python_setup()"*.
2. Es necesario crear un fichero "*setup.py",* que detalla qué ficheros Python se va a compilar y cuál es su "*main*". De igual forma, ROS proporciona una sencilla guía [25] que no solo explica cómo crear el fichero *CMakeLists.txt* para un nodo creado en Python, sino cómo crear el fichero *setup.py.*

En la Ilustración 5-27 se encuentra un extracto del fichero *CMakeLists.txt* utilizado para este robot.

```
cmake_minimum_required(VERSION 2.8.3)

project(webrostc)

find_package(catkin REQUIRED COMPONENTS
  nxt_ros
  nxt_msgs
  sensor_msgs
  roscpp
  rospy
  std_msgs
)

catkin_python_setup()

catkin_package(
  CATKIN_DEPENDS nxt_ros nxt_msgs sensor_msgs roscpp rospy std_msgs
)
```

Ilustración 6-27. Extracto del fichero CMakeLists.txt para construir el proyecto "webrostc".

7. Pruebas y resultados

Dado a que este proyecto tiene como finalidad conectar dos dispositivos para intercambiar datos, ha sido necesario contar con diversos equipos. Durante prácticamente la totalidad de la etapa de desarrollo del proyecto, se ha usado un pc personal, en el cual se han ejecutado tanto el servidor Django como el robot y el navegador.

Posteriormente, para facilitar la última etapa de desarrollo, se ha usado un segundo portátil, con *Ubuntu 16.04.3 LTS* como sistema operativo, con el propósito de emular al robot y al mismo tiempo programarlo.

El equipo final elegido para ejecutar el software del robot ha sido una *raspberry pi 3 model b* (del año 2015) con *Lubuntu 16.04.3 LTS* como sistema operativo, usando una imagen con *ROS Kinetic* preinstalado [26]. La elección de este hardware se debe a sus ya conocidas características, tales como su gran versatilidad, su reducido tamaño y sus opciones de conexión (redes inalámbricas, ethernet, USB, etc.). Sin embargo, como todo hardware, tiene sus limitaciones. Para verificar el alcance del software/hardware, se han realizado las siguientes comprobaciones:

- En primer lugar, se ha testeado la comunicación con el *webSocket* del servidor, realizando la encriptación/desencriptación y establecimiento de clave simétrica de forma prácticamente inmediata.
- Por otro lado, se ha testeado el sistema ROS con el manejo del robot. La interfaz de conexión entre la *raspberry pi* y el *NXT* es mediante *USB*, para evitar las posibles latencias en la comunicación.
- La ejecución del nodo de forma independiente en la *raspberry pi*, estableciendo la conexión con el NXT e inicializando los *topics* no presenta problema alguno. En ningún momento se ha experimentado pérdida de conexión o problema de incompatibilidades.
- Se ha probado la comunicación interna mediante UDP, usada para “parchear” el problema de las versiones Python en los diferentes programas usados. La comunicación se establece sin problemas. Además, no es preciso que se realice la inicialización de la comunicación de una forma concreta. Cuando se

inicializa uno de los extremos, éste comienza a escuchar y enviar información, independientemente de que el otro extremo esté activo.

- Por último, se ha testeado el conjunto de las rutinas, tanto en el servidor y el navegador como en el robot.

Gracias a la flexibilidad de WebRTC, se ha podido testear las diferentes combinaciones de *streaming* de información, probando a transferir audio/video, datos únicamente y todo a la vez:

- La transmisión de datos únicamente, debido a que se trata de ficheros *json* (enviados como tipo *String*), apenas suponen carga de procesamiento para la raspberry pi ni carga de datos para el canal. Por lo que la transmisión de igual forma se realiza sin problemas.
- Se ha probado a manejar el robot desde el navegador (estando ambos conectados en la misma red local para descartar retrasos debidos a la red) y no se aprecia ningún lag.
- Sin embargo, al testear la transferencia de audio/vídeo sí que se han apreciado retrasos en la comunicación, atribuibles, con toda seguridad, a los recursos limitados de los que dispone la *raspberry pi* en cuanto a potencia de procesamiento se refiere.

Es preciso recordar que el procesamiento del audio/vídeo requiere dos procesos para el vídeo y otros dos para el audio. Cada proceso de vídeo realiza una conversión *frame a frame* de *RGB* a *YUV420* y viceversa.

Y, en segundo lugar, y no por ello menos importante, el paquete *aiortc* constituye una implementación de la tecnología WebRTC que, como la propia tecnología original, se encuentra aún en proceso de desarrollo.

Y es que como se explicó en el funcionamiento de WebRTC, aunque la conexión entre ambos *peers* se establezca, ambos siguen generando *ICE Candidates*, con el fin de mejorar la conexión existente (siempre que sea posible), ya que inicialmente se establece una conexión limitada y se va mejorando gradualmente hasta ocupar el ancho de banda disponible. Por tanto, si el *ICE Candidate* inicial se desconecta, es altamente probable que haya otro disponible, incluso con mejor conexión.

Sin embargo, el paquete *aiortc* aún no dispone de dicha característica, por lo que únicamente dispone de los *ICE Candidates* iniciales para realizar la conexión. Si por alguna razón dicho candidatos se desconectan, se pierde la conexión por completo. Además, tampoco permite la mejora de la conexión al no admitir nuevos candidatos.

A esto se suma la incompatibilidad de ROS con Python 3, que obliga a ejecutar un socket UDP en ambos programas que se ejecuta en la raspberry, haciendo que, en conjunto, se cargue en exceso al procesador.

8. Conclusiones y futuras mejoras

Desde el planteamiento de este proyecto como TFG, abordarlo en su totalidad ha supuesto un gran reto personal. Al ser una tecnología tan reciente, no sólo dispone una API que cambia continuamente, así como su código, sino que resulta complejo encontrar la solución a ciertos problemas, como de compilación, compatibilidad, etc.

Hay que tener en cuenta que el proyecto original de WebRTC está enfocado a la comunicación entre dos navegadores, muy lejos del propósito de este proyecto. Por si fuera poco, se ha tenido que adaptar al ecosistema ROS, lo que supone una dificultad adicional.

Y es que, como se comentó anteriormente, inicialmente se trató de abordar el nodo puente con ROS usando C++, pero el código fuente aportado por WebRTC era tan sumamente difuso que, unido al desconocimiento de la tecnología, supuso un reto imposible de afrontar. A esto se le sumó el hecho de que WebRTC no incluye el canal de comunicación inicial, mediante el cual se ponen de acuerdo ambos *peers*. Por tanto, se requería un servidor que hiciera de intercomunicador entre ellos.

Esto desencadenó en la adición de requisitos a la ya extensa lista. Dependiendo de cómo se planteara la solución, desarrollar el software completo podría suponer una inversión de tiempo demasiado amplia. La elección de implantar un servidor usando el *framework* Django supuso un ahorro sustancial de recursos, teniendo en cuenta las posibilidades y facilidades de las que dispone, permitiendo, no sólo desarrollar el servidor que daría soporte a la web, sino incluir un módulo que gestionaría el canal de comunicación inicial, vital para el proyecto que aquí se expone.

Asimismo, al ser un *framework* en Python, no sólo disponía de multitud de paquetes de código abierto, sino que parte de su código podía ser reutilizado en el nodo *bridge* del robot, como se ha podido ver, por ejemplo, en el paquete *"Security"*.

Si bien el requisito base de disponer de un canal de comunicación para la primera toma de contacto entre los *peers* se había satisfecho, aún quedaba por implantar el cliente WebRTC en el robot. Aunque existían diversas opciones, tales como "*EasyRTC", "node-WebRTC"* o "*OpenWebRTC"*, ninguna cumplía los requisitos necesarios, pues o bien no se podían implantar correctamente en el soporte que

requería este caso (en un entorno robótico) o bien no contaban con el soporte necesario para mantener el software necesario, por lo que se corría el riesgo de adoptar una solución en breve obsoleta (en caso de que no lo estuviera ya).

La solución adoptada, mediante el paquete *aiortc,* es la que mejor se amolda a los requisitos del proyecto. Desarrollada para Python 3, se basa en la librería "*asyncio*" para manejar los eventos de entrada salida de forma mucho más eficiente, ya que permite gestionar de forma interna las diferentes hebras encargadas de recibir y enviar la información a través de los puertos correspondientes. A esto se le añade de que el autor mantiene el software actualizado al ritmo de las novedades de WebRTC.

Una vez logrado el objetivo principal, o lo que es lo mismo, establecer la comunicación p2p entre dos clientes WebRTC, la etapa final del desarrollo ha consistido en interconectar el módulo anterior con el módulo gestor del robot. Es decir, permitir que el nodo *bridge* pueda interactuar con los demás nodos que se ejecutan en el entorno ROS. Dado que se decidió integrar el software del robot en un *Lego NXT,* se incorporaron los nodos ROS que tratan con el hardware del robot para crear una capa sencilla de usar que permita interactuar con el mismo.

A pesar de que el desarrollo del proyecto ha supuesto una gran inversión de tiempo y energía, la impresión general ha sido sumamente positiva. La búsqueda incansable de posibles soluciones, así como el trato con nuevas tecnologías como son Django junto con Python o WebRTC ha posibilitado la exploración de nuevos campos, muchos de ellos con gran potencial de cara al futuro.

Y es que se trata de crear una conexión entre un dispositivo y un navegador. Lo que quiere decir que se puede, por ejemplo, manejar un robot, sin importar dónde esté, con el único requisito de que disponga de conexión a internet. Si hoy en día esto supone algo sencillo de suplir, en breve será algo común, tan pronto como se implante el "IoT" (internet de las cosas) y se normalice que cualquier dispositivo tenga conexión a la red.

Si ahora la cantidad de posibilidades de esta tecnología es más que amplia, en el futuro se verá multiplicada sustancialmente.

Es por ello por lo que es cuestión de tiempo (a corto plazo incluso) que se solventen los pequeños problemas que se presentan actualmente al tratar de adaptar esta

tecnología a otros ámbitos. Por ejemplo, la implementación de "ICE Trickle" (que consiste en comenzar a intercambiar "*ICE Candidates"* antes incluso de crear la oferta – respuesta) ya se encuentra disponible en los navegadores, sin embargo, *aiortc* aún la está desarrollando. Tan pronto como se añada esta funcionalidad, el rendimiento de la comunicación se verá mejorada en gran medida.

Cabe añadir, como mejora futura a este proyecto, la implementación del servidor Django en un dominio registrado, de tal forma que se pueda acceder de forma global. Además, dado a que, por seguridad, si la conexión no es segura, el navegador no permite acceder a los recursos multimedia (cámara y micrófono), es más que recomendable añadirle un certificado para proporcionarle *SSL* a la página web (https).

Respecto a la conexión, sería interesante añadir la funcionalidad de que un mismo robot, en vez realizar una sola conexión con un peer remoto, se pueda conectar a varios a la vez, con el propósito de realizar un *streaming* de vídeo con varios usuarios al mismo tiempo (aunque luego se limite el canal de datos a un solo usuario para que el robot únicamente sea manejado por un usuario).

Como se puede apreciar, hay infinidad de opciones posibles, la imaginación es la que establece el límite.

9. Referencias y bibliografía

- [1] Browsers and Mobile apps Real – Time Communication WebRTC. (https://webrtc.org)
- [2] Robot Operating System (ROS) (http://www.ros.org)
- [3] Network Address Translation (NAT) (https://en.wikipedia.org/wiki/Network_address_translation)
- [4] RFC 3489. (https://tools.ietf.org/html/rfc3489)
- [5] “video_stream_opencv” (ROS Node) (http://wiki.ros.org/video_stream_opencv)
- [6] ROS Image (sensor_msgs/Image.msg) (http://docs.ros.org/kinetic/api/sensor_msgs/html/msg/Image.html)
- [7] YUV420 Video (https://en.wikipedia.org/wiki/YUV)
- [8] openCV – Python (http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_setup/py_intro/py_intro.html#intro)
- [9] “audio_common” ROS Package (http://wiki.ros.org/audio_common)
- [10] PyAudio (http://people.csail.mit.edu/hubert/pyaudio/)
- [11] Django (https://docs.djangoproject.com/en/2.0/)
- [12] Django channels (https://channels.readthedocs.io/en/latest/)
- [13] Desarrollo iterativo e incremental (IDD) (https://es.wikipedia.org/wiki/Desarrollo_iterativo_y_creciente)
- [14] WebSockets (http://websockets.readthedocs.io/en/stable/api.html)
- [15] aioRTC (https://aiortc.readthedocs.io/en/latest/api.html)
- [16] Lego NXT (https://www.lego.com/es-es/mindstorms)
- [17] NXT-ROS Core (https://github.com/NXT-ROS/nxt)
- [18] NXT-Python (https://github.com/Eelviny/nxt-python)
- [19] PyCryptodome (https://pycryptodome.readthedocs.io/en/latest/)
- [20] EAX Mode (Encriptación Simétrica) (https://en.wikipedia.org/wiki/EAX_mode)
- [21] Guía Fichero Configuración NXT-ROS (http://wiki.ros.org/nxt_ros/Tutorials/Creating%20a%20full%20robot)

- [22] Guía ROS creación de paquetes (http://wiki.ros.org/ROS/Tutorials/CreatingPackage)
- [23] Guía CMake CMakeLists.txt (https://cmake.org/cmake-tutorial/)
- [24] Guía ROS CMakeLists.txt (http://wiki.ros.org/catkin/CMakeLists.txt)
- [25] Guía ROS MakeFile Python (http://wiki.ros.org/rospy_tutorials/Tutorials/Makefile)
- [26] Imágenes para Raspberry con ROS Preinstalado (https://downloads.ubiquityrobotics.com/)
- [27] Django Redis (http://niwinz.github.io/django-redis/latest/)
- [28] Bootstrap 4 (http://getbootstrap.com/docs/4.1/getting-started/introduction/)
- [29] Awesome Fonts (https://fontawesome.com/icons?d=gallery)
- [30] PyUSB (https://github.com/pyusb/pyusb)
- [31] Python (https://www.python.org/)

10. Índice de ilustraciones

Ilustración 2-1. Modelo conexión ideal.......... 13
Ilustración 2-2. Modelo conexión real. 14
Ilustración 2-3. Usando servidores STUN. 15
Ilustración 2-4. Usando servidores STUN/TURN.......... 16
Ilustración 3-1. Estructura nodo ROS Bridge. 19
Ilustración 4-1. Ejemplo JavaScript para definir qué medios se van a enviar y cuáles se van a recibir. 24
Ilustración 4-2. Ejemplo JavaScript para crear un peer y un dataChannel. 24
Ilustración 4-3. Ejemplo JavaScript para añadir los medios locales al peer.......... 24
Ilustración 4-4. Ejemplo JavaScript para crear una oferta y mandársela al peer remoto. 25
Ilustración 4-5. Ejemplo JavaScript para crear ICE Candidates cuando se encuentren disponibles.......... 25
Ilustración 4-6. Estructura de los nodos WebRTC.......... 26
Ilustración 4-7. Ejemplo JavaScript donde se capturan los medios locales (audio/vídeo)..... 27
Ilustración 5-1. Django FrontEnd DataFlow. 34
Ilustración 5-2. Cliente WebSocket JavaScript. 36
Ilustración 5-3. Parte de "devicesConsumer.py".......... 37
Ilustración 5-4. Diagrama de conexión remota. 38
Ilustración 5-5. Ejemplo de una de las clases para manejar excepciones. 39
Ilustración 5-6. Base de datos de la aplicación Django.......... 40
Ilustración 5-7. Mensajes JSON usados en la encriptación de los datos. 41
Ilustración 5-8. Ejemplo de uso de ficheros estáticos (extracto de template.html). 42
Ilustración 5-9. Inicialización de los sockets (extracto de channels.py). 45
Ilustración 5-10. Diferentes tipos de mensajes Json usados (extracto de jsonMessages.py). 45
Ilustración 5-11. Mostrando vídeo mandado por el peer remoto (extracto de videoHandler.py).......... 47
Ilustración 5-12. Captura y encapsulamiento del vídeo (extracto de videoHandler.py). 47
Ilustración 5-13. Reproduciendo audio remoto (extracto de audioHandler.py). 49
Ilustración 5-14. Capturando audio del micrófono y encapsulándolo (extracto de audioHandler.py).......... 49
Ilustración 5-15. Creación del peer, dataChannel y sus manejadores (extracto de webrtc.py). 50
Ilustración 5-16. Manejo de oferta remota y creación de respuesta (extracto de webrtc.py). 51
Ilustración 5-17. Gestor de mensajes entre channel interno y dataChannel (extracto de webrtc.py). 52
Ilustración 5-18. Constructor y método para firmar mensajes (extracto de asymmetric.py).. 52
Ilustración 5-19. Robot Lego MindStorms NXT.......... 53
Ilustración 5-20. Contenido de la carpeta "catkin_ws", donde se encuentran los nodos ROS. 55
Ilustración 5-21. Inicio del nodo 'publisher' para comunicarse con el robot (extracto de topicsHandler.py). 55
Ilustración 5-22. Clase auxiliar para manejar los comandos entrantes (extracto de topicsHandler.py). 56
Ilustración 5-23. Captura de uno de los dos subscribers (extracto de topicHandler.py). 57

Ilustración 5-24. Configuración del robot NXT (extracto de nxt_robot.yaml)........................... 58
Ilustración 5-25. Fichero de lanzamiento de los nodos ROS correspondientes (extracto de webrostc.launch).. 59
Ilustración 5-26. Captura del fichero package.xml con las dependencias del nodo webrostc. 60
Ilustración 5-27. Extracto del fichero CMakeLists.txt para construir el proyecto “webrostc”.. 61

11. Anexo

11.1 Manual de instalación

11.1.1 Aplicación Django

Si se desea implantar la solución Django, bien en un servidor dedicado, bien en un sistema Linux/Mac/Windows, basta con seguir estos sencillos pasos:

1. En primer lugar, dado que Django está basado en Python, es necesario instalar el compilador Python [31]. Actualmente se encuentra en la versión 3.6.5.
2. Posteriormente, se instalan las dependencias especificadas en el punto 7 con el gestor de paquetes de Python3, *pip3*. Ejemplo: *pip3 install django*
3. Una vez instaladas las dependencias, se puede ejecutar el servidor Django. Para ello, es tan sencillo como navegar hasta la carpeta raíz del servidor (donde se encuentra el fichero "*manage.py*" y ejecutar "*python3 manage.py runserver*". Por defecto, el servidor se iniciará en la dirección IP *127.0.0.1:8000*
4. Para que el servidor sea accesible fuera del propio equipo, es necesario especificarlo en el comando de ejecución. Concretamente, se especifica que la IP sea *0.0.0.0* para que el servidor adopte la IP del equipo. Por tanto, el comando resultante sería: "*python3 manage.py runserver 0.0.0.0*".

11.1.2 Aplicación WebRosTC

Para la ejecución del software del robot, es preciso utilizar un sistema Unix, ya sea bien Linux, o bien Mac, debido que algunas dependencias del paquete *aiortc* no se encuentran para los sistemas Windows (concretamente, los paquetes "*libvpx*" y "*libopus*").

Por otro lado, en el sistema robot, se ha contado con una imagen del sistema operativo *Lubuntu 16.04.3 LTS* [26] con el *framework* ROS preinstalado. Si se desea usar una imagen limpia de dicho S.O o cualquier otro compatible, es necesario instalar además el sistema ROS, como aparece indicando en su página web [2].

Para poder ejecutar correctamente el software del robot, se requieren los siguientes pasos:

1. Al igual que ocurre con Django, es preciso instalar los compiladores necesarios. Si se ha usado la imagen con ROS preinstalado, este paso no es necesario. En otro caso, se deben instalar los compiladores tanto de Python con de C++ y CMake. Para ello basta con instalar el paquete "*build-essentials*" en Linux. Dependiendo de la distribución de Linux usada, el comando varía. En el caso de Ubuntu, para instalarlo bastaría con ejecutar el comando "*sudo apt-get install build-essentials"*.
2. A continuación, de deberán instalar las dependencias especificadas en el apartado 7. Aquellos que se especifican junto a un "apt-get" se instalan de igual forma que el apartado anterior. Los especificados con "pip" se instalan mediante la herramienta *pip* de Python.
3. Los elementos marcados como "source" se deben agregar a la carpeta correspondiente al entorno ROS "catkin_ws". "NXT-Python", tal y como se indica en su página web [18] ha de compilarse (*make*) e instalarse (*make install*). Por otro lado, "NXT-Core ROS" se añade como un nodo más, el cual se compilará junto con el resto de los nodos.
4. Antes de poder ejecutar cualquier programa, es preciso compilarlo, por ello, el siguiente paso es navegar hasta la carpeta "catkin_ws" y ejecutar "catkin build". La razón por la que se usa este comando en vez "catkin_make" es porque, en primer lugar, permite compilar cada nodo de forma independiente, y aunque algún nodo falle en su compilación, el resto de los nodos se siguen compilando y, en segundo lugar, porque así lo especifica el manual de instalación de "NXT-Core ROS" [17].
5. Una vez compilado, es preciso que los nuevos archivos ejecutables sean vistos por la consola, por ello, se actualizan las variables de entorno con el comando *"source devel/setup.bash".*
6. Para ejecutar el nodo, se ejecuta el comando: *"roslaunch webrostc webrostc.launch".*
7. Además, es necesario ejecutar el nodo *bridge* desde otro terminal, el cual se puede ejecutar desde su carpeta raíz con el comando "*python3 robotWebRTC.py".*

11.2 Dependencias

Para recrear este proyecto en otros dispositivos, es necesario suplir las siguientes dependencias:

11.2.1 Dependencias del servidor Django

- PyCryptodome [19]
- Django [11]
- Django channels [12]
- Django redis [27]
- Bootstrap 4 [28]
- Awesome Fonts [29]

11.2.2 Dependencias WebRosTC

- libopus-dev (apt-get)
- libvpx-dev (apt-get)
- libsrtp2-1_2.1.0 (apt-get)
- libsrtp2-dev_2.1.0 (apt-get)
- aiortc (pip) [15]
- websockets (pip) [14]
- PyCryptodome (pip) [19]
- OpenCV-Python (pip) [8]
- libffi-dev (apt-get)
- portAudio19-dev (apt-get)
- pyAudio (pip) [10]
- pyUSB (pip) [30]
- NXT_Python (source) [18]
- NXT_ROS (source) [17]
- ROS Kinetic (apt-get) [2]
- Python 2.7 and 3.5 (apt-get)