Diseño de un servicio WEB y cliente Android para información de tráfico aéreo.

Manuel Marco Bartual

Director: Joan Vila Carbó

Universidad Politécnica de Valencia

DISCA

Máster de Ingeniería de Computadores

Valencia 2015

Índice de contenido

Índice de Figuras

1. INTRODUCCIÓN

1.1. Motivación

Desde hace más de tres décadas, el teléfono móvil se ha ido convirtiendo en uno de los elementos más utilizados y uno de los que más dependencia ha generado en la sociedad.

Un cambio progresivo, pero lento comparado con la revolución que ha tenido lugar desde que se ha dotado a estos dispositivos de tecnologías que les ofrecen características similares a las de los ordenadores tradicionales, como la posibilidad ejecutar aplicaciones, acceder a servicios en internet, etc. Con la aparición de los denominados "teléfonos inteligentes", se ha pasado de utilizar a los dispositivos móviles solamente como teléfonos portátiles, a utilizarlos para cualquier tarea antes inimaginable como la de pagar las compras en los establecimientos, o la de gestionar los aparatos en viviendas con instalaciones domóticas.

Pero todavía se espera que se aumente más su uso debido a que cada vez sirven para hacer más cosas, apoyándose en el hecho de que los dispositivos móviles en general (relojes, accesorios, etc) se van dotando cada vez más de conexión a internet, poniéndose en práctica la tendencia a lo que se denomina "El internet de las cosas".

Esta tendencia está ofreciendo multitud de nuevas oportunidades en el mundo de la programación por la gran demanda software que se genera para estos dispositivos, y personalmente, considero un opción acertada adquirir conocimientos de programación en ese campo para aprovechar esta tendencia y tener la posibilidad de acceder al mercado laboral desarrollando aplicaciones también para dispositivos móviles.

Por otro lado, y en lo que a tendencias respecta, tenemos el hecho destacable de que gestores de red ATM (*Air Traffic Managers*) como EUROCONTROL están empezando gestar mecanismos para ofertar servicios de tráfico aéreo (*Flight Services*, *Airspace Services*, *Flow Services*, *General Information Services*) como servicios web SOAP en una futura Internet de Tráfico Aéreo llamada SWIM que interconectará aeronaves, centros de operación de aerolíneas, gestores de red y centros de control de tráfico entre sí.

Los motivos por los que he decidido realizar este proyecto

son principalmente el de adentrarme en el mundo de la programación de aplicaciones móviles, así como también el de adquirir conocimientos en el ámbito de los servicios web, todo dentro del marco que es la tecnología que se utiliza para la localización de vuelos en la actualidad.

1.2.Definición del problema

En la Universidad Politécnica de Valencia actualmente existe una antena para el rastreo de vuelos en tiempo real, que conectada a una máquina servidor, ofrece los datos sobre la posición de los vuelos que se detectan a clientes, que son aplicaciones de escritorio desarrolladas en java, preparadas específicamente para conectarse a ese servidor.

Pero como tales aplicaciones cliente únicamente pueden ser utilizadas en ordenadores de sobremesa o portátiles, y dadas las necesidades de hoy en día, donde la tendencia es dejar de usarse estos ordenadores con tanta frecuencia para pasar a utilizarse más otro tipo de dispositivos, como los teléfonos móviles o las tabletas, es imprescindible cambiar las diferentes maneras de difundir la información.

Es necesario que los datos sean enviados de un modo más estandarizado, de manera que éstos puedan ser procesados por clientes de terceros que no necesariamente hayan tenido que conocer el código fuente del servidor para generar clientes compatibles, si no que simplemente obteniendo la información de las características del servicio web lo consigan, y también es necesario desarrollar nuevas aplicaciones cliente que puedan ser ejecutadas en dispositivos móviles.

1.3.Objetivos

Partiendo de la base de mi objetivo personal de fondo, que es aprender el desarrollo de aplicaciones para dispositivos móviles y la creación/consumo de servicios web, los objetivos del proyecto son los siguientes:

- Crear un servidor que obtenga los datos del tráfico aéreo del servidor que existe actualmente en la UPV, para que a su vez vuelva a enviarlos mediante servicios web SOAP.
- Desarrollar un cliente que pueda ejecutarse en

dispositivos con la plataforma Android que sea capaz de consumir el servicio web SOAP creado, y que además, éste sea compatible con el servidor que existe actualmente, para de ese modo poder seguir el tráfico aéreo en tiempo real desde los dispositivos móviles Android, aunque el servidor web SOAP no se encuentre operativo en esos momentos.

1.4.Estructura de la memoria

Para facilitar la lectura de la memoria del proyecto, a continuación se detallan los contenidos que albergan cada uno de los capítulos:

- Capítulo 2. Se hace una breve explicación de cómo se monitoriza el tráfico aéreo y de la importancia que tiene el transpondedor para llevar a cabo esa función.
- Capítulo 3. Es un estudio del estado del arte de las tecnologías empleadas donde se demuestra la gran relevancia que tienen dentro del mundo laboral.
- Capítulo 4. Se expone el diseño del proyecto diferenciando cada una de las partes que lo componen por su funcionalidad.
- Capítulo 5. Se exhibe parte del código junto con su correspondiente explicación de la implementación del proyecto.
- Capítulo 6. Se muestran las pruebas y evidencias de ejecución que demuestran que nuestro proyecto cumple con los objetivos requeridos.
- Capítulo 7. Se ponen en relieve las conclusiones a las que llegamos después del desarrollo de nuestro proyecto y además indicamos unos posible trabajos futuros que podrían mejorar en diferentes aspectos nuestro proyecto.
- Capítulo 8. Se compone de anexos que contienen el código completo de algunas de las clase más importantes.
- Capítulo 9. Es un acopio de fuentes de consulta de donde se ha obtenido la información para poder llevar a cabo el proyecto.

2.TRÁFICO AÉREO / TRANSPONDEDOR

Para la recepción de la información sobre el estado del tráfico aéreo en tiempo real, en un servicio de control del tráfico aéreo ATC (*Air Traffic Control*) se dispone de un radar primario PSR (*Primary Surveillance Radar*) que detecta las aeronaves mediante la reflexión de las ondas electromagnéticas y de un radar secundario SSR (*Secondary Surveillance Radar*) compuesto por un transpondedor para obtener más detalles de la información de cada aeronave.

Fig. 2.1: Esquema de la detección de aeronaves actual.

2.1.Radar primario PSR

El radar primario muestra al sistema de control de tráfico aéreo lo que sería una representación en forma de mapa, en el que se indican todos los ecos de radar de los aviones que se detectan dentro del alcance del radar. El sistema de control de tráfico aéreo puede determinar la posición de cada blanco detectado respecto al lugar de la instalación radar, de un blanco detectado con respecto a otro, así como del blanco con respecto a una estación de tierra definida en un mapa. Es decir, el equipo permite al controlador localizar blancos, pero no los puede identificar instantáneamente.

La información obtenida mediante un radar primario sería similar a la siguiente figura, un conjunto de circunferencias concéntricas dónde la última circunferencia externa delimitaría el alcance máximo del radar, el cual estaría situado justo en el centro común a todas en las circunferencias. En la representación también se pueden ver los blancos detectados que son los puntos verdes (serían las aeronaves detectadas).

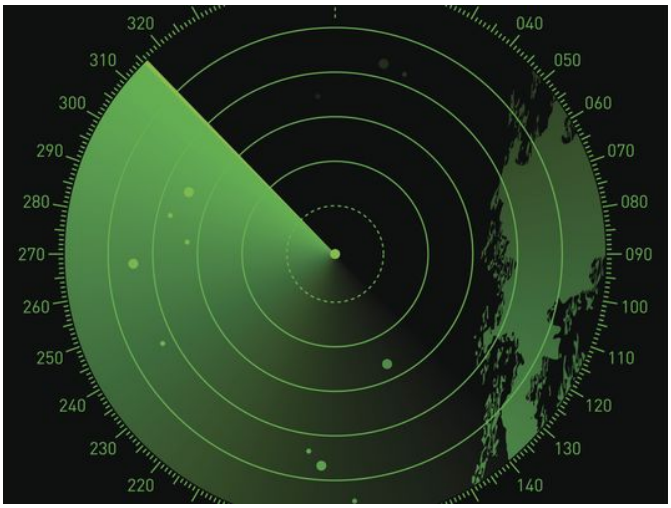

Fig. 2.2: Información obtenida del radar primario

Para controlar el tráfico aéreo resulta insuficiente porque no permite la identificación instantánea de las aeronaves detectadas, tampoco da a conocer la altitud a la que vuelan y además tiene un alcance mucho menor con respecto a un radar secundario SSR. Pero también es imprescindible debido a que al no tener implementado ningún elemento informático en su mecanismo, la probabilidad de que falle es menor, y por lo tanto, tiene una fiabilidad mucho mayor a la del radar secundario SSR que sí está completamente informatizado.

2.2.Radar Secundario SSR

Con el paso del tiempo, el servicio de control del tráfico aéreo ha ido requiriendo de más información (de la que haya podido proporcionar el radar primario) para poder gestionar las aeronaves en su espacio aéreo. Debido a que la cantidad de tráfico es cada vez mayor, es más probable que se produzcan colisiones entre los aviones, y es necesario conocer más datos y más precisos del comportamiento de éstos (como la altura, velocidad, etc.) para poder hacer un pronóstico más sofisticado de sus trayectorias, y poder evitar así este tipo de incidentes. Con lo que se impone la instalación de éste tipo de radares (radar secundario SSR) en todos los sistemas de control del tráfico aéreo comercial para tal fin.

El radar secundario funciona de una manera totalmente distinta a como lo hace el radar primario. En el radar primario, como hemos visto anteriormente, se detectan las aeronaves por reflexión (eco) de las ondas electromagnéticas emitidas, con lo que éstas no tienen que hacer nada, son pasivas. En cambio, en el funcionamiento del radar secundario éstas tienen un papel más activo. En este caso se intercambian mensajes entre las aeronaves y el radar secundario.

El radar secundario se define como estación "interrogadora" que emite unos mensajes que son recogidos por un aparato instalado a bordo de cada aeronave. Dicho equipo descifra cada mensaje o "interrogación" y retransmite otro mensaje o "respuesta" que es recibida por el radar de tierra. Dependiendo del tipo de mensaje de "interrogación" se generará un contenido de respuesta diferente en cada caso. Cada mensaje de respuesta contiene parámetros distintos del estado de vuelo de cada aeronave obtenidos de los sensores de abordo. Son parámetros de altitud, velocidad, posición, id, etc. que junto con la información obtenida del radar primario en pantalla, se genera un mapeado en la pantalla de los controladores muy parecido al de la realidad.

Este intercambio de mensajes entre la torre de vigilancia terrestre y las aeronaves se realiza gracias a unos equipos llamados transpondedores que lógicamente, tienen que estar instalados tanto en el radar secundario como abordo en cada aeronave. A continuación veremos con más detalle cuál es el funcionamiento de estos aparatos.

2.3.Transpondedor

El término transpondedor se origina de la fusión de las palabras transmisor y respondedor (receptor). Técnicamente un transpondedor es una combinación entre un radio transmisor y un receptor de radio que funciona retransmitiendo automáticamente datos entre las aeronave y los controladores de tráfico aéreo en tierra. Las señales enviadas proporcionan una identidad única para cada aeronave, especialmente para evitar colisiones en espacios aéreos muy masificados. Los transpondedores fueron creados inicialmente en la segunda guerra mundial para las aeronaves militares con el objetivo de identificar cual era su pertenencia, si al bando propio o al bando enemigo. Pero más tarde su objetivo evolucionó al campo comercial. La mayoría de los centros de control de tráfico aéreo dan preferencia a los datos de alcance y altitud recibidos de aeronaves comerciales por encima de los datos detectados en bruto por el radar primario.

Fig. 2.3: Transpondedor Bendix/King KT 73

Todas las consultas o "interrogaciones" se realizan en la frecuencia de 1030 MHz. Hay cuatro maneras de consulta llamados Modos. Cuando se realiza una consulta en Modo A a una aeronave se le está preguntando por su identidad, en cambio, si ésta se realiza en Modo C se estará preguntando por su altitud. Los Modos B y D no se utilizan. El Modo civil A coincide con el Modo militar 3, por lo que el Modo A es llamado también muy a menudo Modo 3/A. Las respuestas son denominadas *Codes*. La *Code* del Modo A son la retransmisión de cuatro dígitos que se han introducido previamente en cabina, y la *Code* del Modo C la retransmisión automática de la altitud (en escala 103.2 hectoPascales como en los niveles de vuelo). Combinando todos estos mensajes de respuesta se confecciona una información bastante certera y simple en la pantalla el controlador aéreo del radar para su gestión.

Sin embargo, debido al aumento del tráfico aéreo comercial, se ha ido generando una carencia de *Codes* posibles para el Modo A y además, la frecuencia de respuesta de 1090 MHz ha ido quedando saturada, generando el problema de que las respuestas desde más de una aeronave hacia una consulta pueden solaparse generando confusión, así como respuestas erróneas recibidas en tierra desde transpondedores que fueron interrogados por otros interrogadores también en tierra.

2.3.1.Modo S

El Modo *Mode Selection Beacon System* comúnmente denominado como "Modo S", fue desarrollado por Lincoln Labs en 1975 como una forma capaz de supervisar aeronaves dentro de un área de tráfico aéreo denso y solventar los problemas que tenían los Modos A/C.

A raíz de un accidente aéreo producido en 1986 se cambió la ley en EEUU. Esta nueva ley incluyó el requisito de que todas las compañías aéreas que operasen en EEUU llevando más de treinta pasajeros debían estar equipadas con instalaciones de sistemas de evitación de colisiones de tráfico aéreo TCAS (*Traffic colision avoidance system*) II en 1993 mientras que las compañías que operasen con de diez a treinta pasajeros sólo debían estar equipadas con TCAS I (ambas TCAS incluyen el Modo S). El Modo S lleva a cabo todas las funciones de los transpondedores del Modo A y del Modo C y tiene capacidad *Datalink* (capacidad de llevar a cabo funciones automatizadas en el intercambio de datos entre las aeronaves y el control terrestre). Los transpondedores de Modo S son un componente que se integra en todos los sistemas de evitación de colisiones de tráfico aéreo TCAS II y reemplaza a los transpondedores de Modo A y Modo C en las aeronaves equipadas con TCAS II. Un transpondedor de Modo S puede ser instalado para reemplazar a otro transpondedor de Modo A o C sin la necesidad de la instalación de TCAS.

Sin embargo, el Modo S es requerido dentro de TCAS para facilitar el ID, el *Flight status* (que indica si la aeronave está se encuentra actualmente en tierra o en vuelo) y la altitud. Diversas clases de transpondedores identificadas con los niveles de capacidad han sido desarrollados a lo largo del tiempo. Sin embargo, para reunir los requerimientos EUROCONTROL (La organización europea por la seguridad en navegación aérea) en la etapa de *Elementary Surveillance* (ELS, rastreo básico), es necesario como mínimo un transpondedor de *Level 2* capaz de soportar tanto los códigos *Interrogator Identifier* (II) como *Surveillance Identifier* (SI) de acuerdo con los requerimientos de la organización civil internacional de aviación (ICAO).

Una característica nueva del transpondedor de Modo S es que cada aeronave es definida con un único código de direccionamiento el cual es difundido en transmisiones automáticas (que no son solicitadas), llamadas "SQUITTER", cada intervalos de tiempo de un segundo aproximadamente. Un controlador de tráfico aéreo u otra aeronave equipada con un transpondedor de Modo S puede utilizar dicho código de direccionamiento con propósitos de consulta o comunicación. La capacidad *Flight ID* es una de las funciones también requerida en la etapa de *Elementary Surveillance*.

La segunda etapa, la de *Enhanced Surveillance* (EHS) es la más actual y permite una extensión del código SQUITTER de hasta 24 bits, transportando en su contenido el código de direccionamiento requerido por la ICAO y la información de reporte de la aeronave.

Los parámetros de información de las aeronaves requeridos en la etapa *Enhanced Surveillance* son los siguientes:

- *Magnetic heading* (Dirección de la aeronave relativa al Polo Norte magnético)
- *Indicated Airspeed/Match Number* (Velocidad)
- *Roll Angle* (ángulo de balanceo)
- *Track Angle Rate* (Velocidad de giro del avión)
- *Vertical Rate* (Velocidad de ascenso/descenso)
- *True Track Angle* (ángulo de derrota real)
- *Ground Speed* (Velocidad en tierra)
- *FMS Selected Altitude* (Altitud selecionada en el Sistema de Gestión de Vuelo FMS)

El siguiente esquema muestra los cambios en el aspecto de las tramas de SQUITTER normal y extendido.

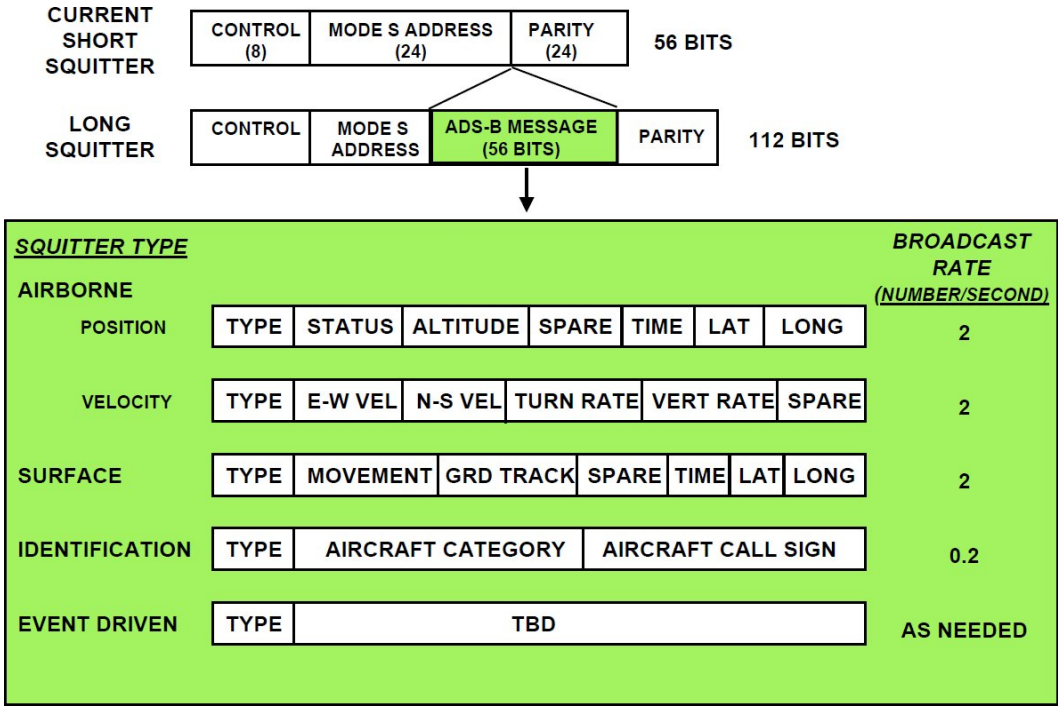

Fig. 2.4: Trama de SQUITTER extendido.

Los estándar para los servicios específicos del Modo S están descritos en ICAO (Organización de Aviación Civil Internacional).

Hay numerosos modelos de transpondedores de Modo S en servicio y no es fácil su identificación a simple vista. Existen alrededor de nueve fabricantes de sistemas de transpondedores de Modo S para aeronaves, algunos de los más conocidos son los siguientes: ACSS, *AlliedSignal*, *Bendix/King*, *Filser*, *Garmin*, *Honeywell*, *Narco*, *Rockwell Collins* y *Thompson-CSF*.

Dependiendo de su nivel de capacitación, los transpondedores Modo S son sometidos a inspecciones técnicas rutinarias para garantizar la seguridad requerida en cada nivel.

2.4.Radar UPV

En la UPV se dispone de un radar para el rastreo del tráfico aéreo. Es una antena conectada a una especie de unidad transpondedor virtual del fabricante *BaseStation* que es capaz de recibir mensajes que envían las aeronaves de manera pasiva, es decir, que puede recibir las respuestas a consultas que son hechas desde algún radar secundario, que tenga dentro de su cobertura las aeronaves que le responden.

Los usuarios pueden ver los datos en crudo enviado por la unidad SBS mediante el uso de la aplicación *Telnet* al puerto 30003.

El flujo de datos tiene el siguiente aspecto:

```
STA,,5,179,400AE7,10103,2008/11/28,14:58:51.153,2008/11/28,14:58:51.153,RM
MSG,4,5,211,4CA2D6,10057,2008/11/28,14:53:49.986,2008/11/28,14:58:51.153,,,408.3,146.4,,,64,,,,,
MSG,8,5,211,4CA2D6,10057,2008/11/28,14:53:50.391,2008/11/28,14:58:51.153,,,,,,,,,,,0
MSG,4,5,211,4CA2D6,10057,2008/11/28,14:53:50.391,2008/11/28,14:58:51.153,,,408.3,146.4,,,64,,,,,
MSG,3,5,211,4CA2D6,10057,2008/11/28,14:53:50.594,2008/11/28,14:58:51.153,,37000,,,51.45735,-
1.02826,,,0,0,0,0
MSG,8,5,812,ABBEE3,10095,2008/11/28,14:53:50.594,2008/11/28,14:58:51.153,,,,,,,,,,,0
MSG,3,5,276,4010E9,10088,2008/11/28,14:53:49.986,2008/11/28,14:58:51.153,,28000,,,53.02551,-
2.91389,,,0,0,0,0
MSG,4,5,276,4010E9,10088,2008/11/28,14:53:50.188,2008/11/28,14:58:51.153,,,459.4,20.2,,,64,,,,,
MSG,8,5,276,4010E9,10088,2008/11/28,14:53:50.594,2008/11/28,14:58:51.153,,,,,,,,,,,0
MSG,3,5,276,4010E9,10088,2008/11/28,14:53:50.594,2008/11/28,14:58:51.153,,28000,,,53.02677,-
2.91310,,,0,0,0,0
MSG,4,5,769,4CA2CB,10061,2008/11/28,14:53:50.188,2008/11/28,14:58:51.153,,,367.7,138.6,,,-
2432,,,,,
MSG,8,5,769,4CA2CB,10061,2008/11/28,14:53:50.391,2008/11/28,14:58:51.153,,,,,,,,,,,0
```

Fig. 2.5: Ejemplo de mensajes enviados por la unidad.

2.4.1.Tipos de mensajes:

Hay seis tipos de mensajes: MSG, SEL, ID AIR, STA, CLK.

La mayor parte de los datos que provienen de las aeronaves están contenidos en el tipo de mensajes MSG mientras que los otros tipos de mensajes son provocados mediante entrada del usuario o configuración del sistema.

En versiones *BaseStation* anteriores a la 1.2.3.145 los mensajes del tipo MSG tenían un retardo de cinco minutos pero de esta versión en adelante, estos mensajes son enviados en tiempo real.

Los tipos de mensajes son los siguientes:

ID	TIPO	DESCRIPCIÓN
SEL	SELECTION CHANGE MESSAGE	Generado cuando el usuario cambia la aeronave seleccionada en la BaseStation.
ED	NEW ID MESSAGE	Generado cuando una aeronave que está siendo rastreada cambia su “Callsign”.
AIR	NEW AIRCRAFT MESSAGE	Generado cuando unidad SBS recoge una señal de una aeronave que actualmente no estaba siendo rastreada.
STA	STATUS CHANGE MESSAGE	Generado cuando el estado de una aeronave cambia de acuerdo con los valores de “time-out” en el menú de configuración de datos.
CLK	CLICK MESSAGE	Generado cuando el usuario hace doble-click (o presiona intro) sobre una aeronave (i.e. Para abrir una ventana con los detalles de la aeronave)
MSG	TRANSMISSION MESSAGE	Generado por las aeronaves. Hay ocho clases diferentes de mensajes MSG

Los mensajes (MSG) desde las aeronaves pueden ser de una de las siguiente clases de MSG:

ID	TIPO	DESCRIPCIÓN
MSG,1	ES Identification and Category	Generado por el sensor del tren de aterrizaje delantero.
MSG,2	ES Surface Position Message	Generado por el sensor del tren de aterrizaje delantero.
MSG,3	ES Airborne Position Message	Generado por el sensor del tren de aterrizaje delantero.
MSG,4	ES Airborne Velocity Message	Generado por el sensor del tren de aterrizaje delantero.
MSG,5	Surveillance Alt Message	Generado por el radar de tierra. Sin seguridad de CRC. MSG,5 Sólo se enviará si éste es precedido por un mensaje MSG,1,2,3,4, ó ,8.
MSG,6	Surveillance ID Message	Generado por el radar de tierra. Sin seguridad de CRC. MSG,5 Sólo se enviará si éste es precedido por un mensaje MSG,1,2,3,4, ó ,8.
MSG,7	Air To Air Message	Generado por TCAS.
MSG,8	All Call Reply	Emitido sino también generado por el radar de tierra.

2.4.2.Campos de Datos:

Cada uno de los tipos de mensajes de arriba pueden contener hasta 22 campos de datos separados por comas.

Estos campos son:

1	Message Type	(MSG, STA, ID, AIR, SEL, ó CLK)
2	Transmission Type	Subclase de mensaje tipo MSG (1-8). No usado en otro tipo de mensajes.
3	Sesion ID	Número de Registro de Sesión en la Base de Datos.
4	AircraftID	Número de Registro de la Aeronave en la Base de Datos.
5	Hexident	Código hexadecimal Modo S de la Aeronave.
6	FlightID	Número de Registro de Vuelo en la Base de Datos.
7	Date message generated	Fecha de creación del mensaje.
8	Time message generated	Hora de creación del mensaje.
9	Date message logged	Fecha de registro del mensaje.
10	Time message logged	Hora de registro del mensaje.

Los campos de datos básicos de arriba son comunes a todos los mensajes (el campo 2 se utiliza sólo para mensajes del tipo MSG). Los campos de abajo contienen información específica de cada aeronave.

11	Callsign	Identificador de vuelo de ocho dígitos, puede número de Registro o de vuelo (o puede ir vacío).
12	Altitude	Altitud en Modo C. Altura relativa a 1013.2mb (Flight Level). No se refiere a la altura AMSL (sobre n. del mar).
13	GroundSpeed	Velocidad en tierra (no indica la velocidad en vuelo).
14	Track	Dirección de la aeronave. Calculada a partir de la velocidad Este/Oeste y de la velocidad Norte/Sur
15	Latitude	Latitud. Norte y Este positivo. Sur y Oeste negativo.
16	Longitude	Longitud. Norte y Este positivo. Sur y Oeste negativo.
17	VerticalRate	Resolución de 64ft.
18	Squawk	Código “Squawk” asignado por el Modo A.
19	Alert (Squawk change)	Flag para indicar que el “Squawk” ha cambiado.
20	Emergency	Flag para indicar que un código de emergencia ha sido introducido.
21	SPI (Ident)	Flag para indicar que ha sido activado el identificador del *BaseStation* transpondedor.
22	IsOnGround	Flag para indicar si la aeronave está en tierra.

Observaciones:

El campo 11 (*Callsign*) es de ocho caracteres (6 bits ASCII). En la *BaseStation*, un NULL se muestra como el carácter '@' que en ASCII se utiliza para representar el NULL. Los caracteres introducidos en cabina como espacios se envían como caracteres NULL por el transpondedor. Por lo tanto, si se introducen ocho espacios en cabina, se mostrarán como "@@@@@@@@" en el campo.

- El campo 12 (*Altitude*) puede tener una resolución de 25 o de 100 ft. El Modo C tiene una resolución de 100ft, pero la mayoría de las aeronaves actualmente lo envían con una resolución de 25ft, que es un requisito para poder volar en el espacio aéreo europeo. La unidad *BaseStation* solamente muestra la altitud barométrica, pero los datos que envía son HAE (*height above ellip-soid*, altura sobre la elipsoide), la cual es la dife-rencia entre la altitud GPS y la altitud barométrica.

2.4.3.Campos de los Mensajes:

Cada tipo de mensaje contiene diferentes campos en su contenido. En la tabla siguiente, el verde representa los campos que son enviados y los blancos representan los datos que son NULL. Como puede observarse, los mensajes del tipo MSG contienen hasta 22 campos, mientras que los otros tipos hasta 10 campos.

	1	2	3	4	5	6	7	8	9	10		11	12	13	14	15	16	17	18	19	20	21	22
MSG 1	MT	TT	SID	AID	Hex	FID	DMG	TMG	DML	TML		CS											
MSG 2													Alt	GS	Trk	Lat	Lng						Gnd
MSG 3													Alt			Lat	LNG			Alrt	Emer	SPI	Gnd
MSG 4														GS	Trk			VR					
MSG 5													Alt							Alrt		SPI	Gnd
MSG 6													Alt						Sq	Alrt	Emer	SPI	Gnd
MSG 7													Alt										Gnd
MSG 8																							Gnd
SEL												CS											
ID												CS											
AIR																							
STA																							
CLK				-1		-1																	

Fig. 2.6: Campos de los mensajes enviados por la BS.

Observaciones:

- Los mensajes STA utilizan el campo *Callsign* para registrar *flags* de estado basados en valores *time-out* del usuario. Los valores son PL (*Position Lost*), SL (*Signal Lost*), RM (*Remove*), AD (*Delete*), y OK (utilizada para resetear los *time-out* si la aeronave vuelve a estar dentro de la cobertura).

- Los mensajes CLK devuelven un valor -1 en los campos 4 y 6. El campo 5 será NULL.

- Los mensajes del tipo MSG,7 (mensaje aire-aire) han sido añadidos recientemente.

- Aunque actualmente las aeronaves retransmiten el "*Heading*" y la "*True Airspeed*", la unidad no envía esta clase de información.

- Puede observarse que no existe algún tipo de mensaje MSG que ofrezca todos los datos que se utilizan en al unidad *BaseStation* y que algunos campos de datos son únicos para un tipo determinado de mensaje. El campo *Callsign* sólo puede encontrarse en mensajes del tipo MSG,1, el *VertRate* solamente en mensajes del tipo MSG,4 y el *Squawk* en MSG,6.

- Para completar los once campos de datos de una aeronave determinada sería necesaria la recepción de al menos cuatro tipos diferentes de mensajes MSG (MSG,1, MSG,3, MSG,4, y MSG,6) pero, nótese que los mensaje del tipo MSG,6 únicamente son provocados por la consulta del radar de tierra. Si la aeronave no está dentro de la cobertura de algún radar de tierra, ningún mensaje del tipo MSG,6 será enviado. Dado que el tipo de mensaje MSG,6 es el único por el que se envía el código *Squawk*, dicho código sólo estará disponible para los usuarios de *BaseStation* que reciban datos de aeronaves que se estén comunicando en Modo S con un radar de tierra dentro de su cobertura.

- Los mensajes del tipo MSG,5 y MSG,8 se obtienen de misma manera de consulta que los del tipo MSG,6, pero los datos que éstos contienen pueden ser obtenido mediante otro tipo de mensajes.

- En la tabla anterior se puede ver que en los mensajes del tipo MSG,1, únicamente se envían datos en los once primeros campos y el resto van vacíos. Los campos vacíos

contiguos se representan como una serie de comas juntas (,,,,).

- Si el campo 22 (*IsOnGround*, aeronave en tierra) es difundido, esto provocará cambios en los valores de los campos 12 (Altitud), 15 (Latitud) y 16 (Longitud). En ese caso el campo 12 será cero y mientras la aeronave permanezca en tierra no habrá cambios en ese campo.

A continuación puede verse un ejemplo de cada tipo de mensaje que podría ser enviado:

```
SEL,,496,2286,4CA4E5,27215,2010/02/19,18:06:07.710,2010/02/19,18:06:07.710,RYR1427
ID,,496,7162,405637,27928,2010/02/19,18:06:07.115,2010/02/19,18:06:07.115,EZY691A
AIR,,496,5906,400F01,27931,2010/02/19,18:06:07.128,2010/02/19,18:06:07.128
STA,,5,179,400AE7,10103,2008/11/28,14:58:51.153,2008/11/28,14:58:51.153,RM
CLK,,496,-1,,-1,2010/02/19,18:18:19.036,2010/02/19,18:18:19.036
MSG,1,145,256,7404F2,11267,2008/11/28,23:48:18.611,2008/11/28,23:53:19.161,RJA1118,,,,,,,,,,
MSG,2,496,603,400CB6,13168,2008/10/13,12:24:32.414,2008
/10/13,12:28:52.074,,,0,76.4,258.3,54.05735,-4.38826,,,,,0
MSG,3,496,211,4CA2D6,10057,2008/11/28,14:53:50.594,2008/11/28,14:58:51.153,,37000,,,51.45735,-
1.02826,,,0,0,0,0
MSG,4,496,469,4CA767,27854,2010/02/19,17:58:13.039,2010/02/19,17:58:13.368,,,288.6,103.2,,,-
832,,,,,
MSG,5,496,329,394A65,27868,2010/02/19,17:58:12.644,2010/02/19,17:58:13.368,,10000,,,,,,,0,,0,0
MSG,6,496,237,4CA215,27864,2010/02/19,17:58:12.846,2010
/02/19,17:58:13.368,,33325,,,,,,0271,0,0,0,0
MSG,7,496,742,51106E,27929,2011/03/06,07:57:36.523,2011/03/06,07:57:37.054,,3775,,,,,,,,,0
MSG,8,496,194,405F4E,27884,2010/02/19,17:58:13.244,2010/02/19,17:58:13.368,,,,,,,,,,,0
```

Fig. 2.7: Ejemplo de cada uno de los tipos de mensajes.

2.4.4. Precisión posicional

Para lograr la máxima precisión posicional, los datos de latitud y longitud son enviados en 17 bits. Los 17 bits ofrecen una precisión de cuatro decimales para cada valor, que se traducen en un margen de error de 5.1 metros. Para el posicionamiento en tierra se requiere de una precisión mayor, de diez decimales para cada dato, que equivalen a un margen de error de 1.25 metros.

3.Estado del Arte en Tecnologías empleadas.

3.1.Servicios Web Java

Los Servicios Web son aplicaciones cliente/servidor sobre WWW (*World Wide Web's*) y HTTP (*Hyper Text Transfer Protocol*). Como se indica en el W3C (*World Wide Web Consortium*), los Servicios Web proveen medios estándar de interoperabilidad entre aplicaciones software que son ejecutadas en diferentes plataformas o *frameworks*. Se caracterizan por su gran interoperabilidad, capacidad de extenderse, y también por sus descripciones procesables por cualquier ordenador gracias a al uso de XML. Los Servicios Web pueden estar combinados de una manera imprecisa para lograr cálculos con operaciones complejas. Cualquier tipo de programas que proporcionen servicios simples pueden relacionarse entre ellos para ofrecer servicios de valor añadido sofisticados.

3.1.1.Tipos de Servicios Web

A nivel conceptual, un servicio es un componente software proporcionado mediante un punto o extremo accesible a través de una red. Tanto los usuarios como los proveedores del servicio, utilizan mensajes para intercambiar información con la finalidad de realizar invocaciones de consulta y respuesta, en forma de documentos auto-contenido que hacen muy pocas conjeturas sobre las características tecnológicas de la máquina receptora.

Mientras que a nivel técnico, los Servicios Web pueden implementarse de varias formas. Estas dos maneras de implementación se denominan Servicios Web de tipo “Big” (SOAP) y Servicios Web de tipo “RESTful”.

3.1.1.1.Servicios Web "Big" (SOAP)

Los objetivos de diseño de este tipo de servicios web son proporcionar una interacción transparente entre pilas de tecnología *Middleware* heterogéneas y el fomento del acoplamiento flexible entre el consumidor y el proveedor de los servicios.

Los Servicios Web "Big" utilizan mensajes XML que cumplen con el estándar SOAP (*Simple Object Access Protocol*), que es un lenguaje XML para la definición de una arquitectura y un formato de mensajería. Tales sistemas, a menudo contienen una descripción interpretable por máquina de las operaciones que se ofrecen por el servicio, escritas en lenguaje WDSL (*Web Services Description Language*) que es un lenguaje XML que se utiliza para la definición sintáctica de interfaces. En Java EE6, JAX-WS (API de Java para los Servicios Web XML) proporciona la funcionalidad para este tipo de Servicios Web "Big".

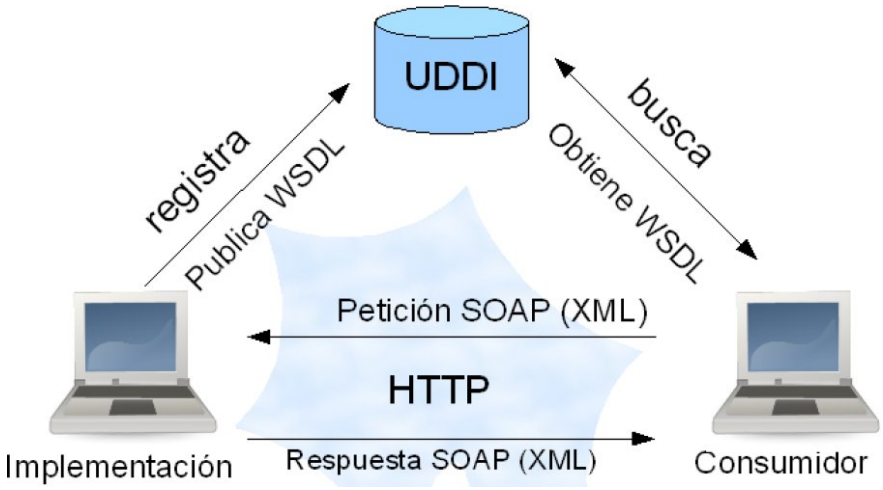

Fig. 3.1: Esquema de las interacciones con WS

La documentación SOAP define un elemento XML de nivel superior llamado "*envelope*", que contiene una cabecera y un cuerpo. La cabecera SOAP es un contenedor extensible para la información de la infraestructura de capa de mensaje que puede ser utilizada para la configuración del encaminamiento y la calidad del servicio. El cuerpo contiene el contenido del mensaje. El esquema XML se utiliza para describir la estructura del mensaje SOAP, de manera que un motor SOAP en cada uno de los extremos de la conexión realizan la función del codificado y decodificado del contenido del mensaje y también su posterior entrega de su contenido a su destino para su procesamiento.

Como hemos dicho anteriormente, mediante WSDL se pueden describir las interfaces sintácticamente. El *Port type* en WSDL contiene múltiples operaciones abstractas que son asociadas con mensajes entrantes y salientes. El *Binding* en WSDL enlaza el conjunto de operaciones abstractas con protocolos de transporte concretos y con formatos de serialización. Los servicios son direccionados, o bien a nivel de transporte, es decir, con URIs (*Universal Resource Identifiers*) con destino SOAP/HTTP, o bien, a nivel de la capa de mensaje con el direccionamiento del estándar WSDL.

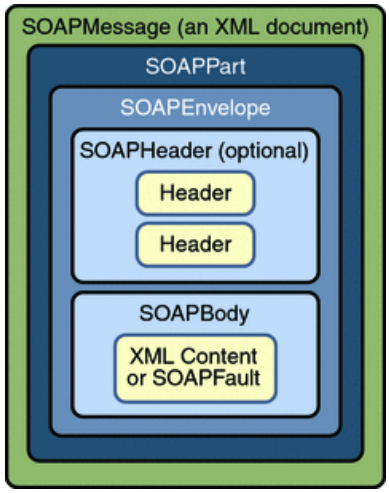

Fig. 3.2: Estructura del mensaje SOAP

Por defecto, no existe la noción de estado, es decir, no se mantiene ninguna información entre sucesivas invocaciones. La interacción con los servicios web sin estado son proporcionados gracias a al WSRF (*Web Services Resource Framework*), que opera la gestión de recursos tras una interfaz de servicios web. La tecnología de la pila WSDL ofrece también muchas otras características QoS (*Quality of Service*) requeridas para garantizar la interacción entre sistemas *middleware* avanzados. Existe un gran conjunto de especificaciones WSDL dada la modularidad y la complejidad del enfoque.

El formato de mensajes SOAP y la interfaz de definición de lenguajes WSDL han tenido una aceptación bastante generalizada. Muchas herramientas de desarrollo, como por ejemplo la IDE (*Integrated development environment*) de *NetBeans*, ofrecen la ventaja de reducir la complejidad del desarrollo de aplicaciones con servicios web.

Un diseño basado en SOAP debe incluir los siguientes elementos:

- Se debe establecer un convenio formal para describir la interfaz que ofrece el servicio. WSDL podría ser utilizado para la descripción de los detalles del convenio que incluye mensajes, operaciones, *Bindings* y la ubicación del servicio web. También es posible el uso de mensajes SOAP en un servicio JAX-WS sin la necesidad de haberlo publicado previamente en un WSDL.
- La arquitectura debe abordar los requisitos complejos no funcionales. Numerosas especificaciones de servicios web cumplen con tales requerimientos y establecen un vocabulario común para ellos. Estas especificaciones pueden incluir seguridad, direccionamiento, confianza, coordinación, etc.
- Es necesario que la arquitectura gestione la invocación y el procesamiento asíncrono. En tales casos, la infraestructura proporcionada por estándares como WSRM (*Web Services Reliable Messaging*) y APIs (*Application Programming Interface*) como JAX-WS, con su soporte de invocación asíncrona en el lado del cliente, pueden ser aprovechadas.

3.1.1.2.Servicios Web RESTful

REST (*Representational State Transfer*) fue originalmente presentado como un estilo arquitectural para la construcción de sistemas hipermedia distribuidos de gran envergadura. Este estilo es más bien una entidad abstracta, cuyos principios han sido utilizados para expresar la excelente escalabilidad del protocolo HTTP 1.0 y también ha limitado el diseño de su versión siguiente, HTTP 1.1. De ese modo, el término REST es muy a menudo utilizado junto con HTTP.

El estilo arquitectural REST está basado en cuatro principios:

- Identificación de recurso mediante URI. Un servicio web RESTful expone un conjunto de recursos que identifican los objetivos de las interacciones con sus clientes. Los recursos son identificados mediante URIs, que proporcionan un espacio de direccionamiento para la detección de recursos y servicios.

- Interfaz uniforme. Los recursos son manipulados con el uso de un conjunto definido de cuatro operaciones, crear, leer, actualizar y borrar: PUT, GET, POST y DELETE. PUT crea un recurso nuevo que puede ser eliminado con la operación DELETE. GET obtiene el estado actual de un recurso representado en cualquier formato y POST transfiere un estado nuevo a un recurso.

- Mensajes auto-descriptivos. Los recursos son separados de su representación de manera que su contenido pueda ser accedido en una diversidad de formatos como HTML, XML, texto plano, PDF, JPEG, etc. Los metadatos del recurso están disponibles y son utilizados por ejemplo para controlar el cacheo, detectar errores de transmisión, negociar el formato adecuado de la representación y también llevar a cabo el control de autenticación o acceso.

- Operaciones con estado mediante hiperenlaces. Toda interacción con un recurso es "sin estado", es decir, los mensajes de petición son autosuficientes. Las interacciones "con estado" se basan en el concepto de transferencia de estado explícito. Existen varias técnicas de intercambio de estado como reescritura URI, las *Cookies*, y campos de formulario ocultos. El estado puede ser incrustado en mensajes de respuesta para indicar estados futuros en la interacción.

En Java EE6, JAX-RS proporciona la funcionalidad para los servicios web RESTful. REST es muy adecuado para escenarios básicos de integración ad-hoc. Los servicios web RESTful son más a menudo mejor integrados con HTTP que con SOAP, no requieren de mensajes XML o de definiciones API de servicios WSDL.

El proyecto Jersey es la implementación de referencia de producción preparada para la especificación JAX-RS. Jersey implementa soporte para las notas definidas en la especificación JAX-RS, facilitando a desarrolladores la construcción de servicios web con Java y la maquina virtual Java (JVM).

Debido a que los servicios RESTful utilizan estándares ampliamente conocidos como W3C y IETF (*Internet Engineering Task Force*: HTTP, XML, URI, MIME), y que poseen una infraestructura ligera que permite que los servicios se construyan con un uso mínimo de herramientas, el desarrollo de este tipo de servicios web es barato, y por lo tanto existe una barrera muy pequeña para su adopción. Se pueden utilizar herramientas de desarrollo tales como el IDE de *NetBeans* para reducir aún más la complejidad del desarrollo de servicios web RESTful.

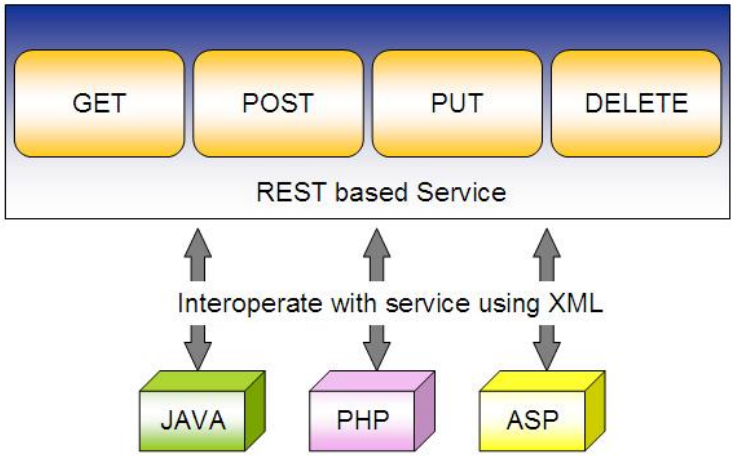

Fig. 3.3: Esquema de operaciones en RESTful

Un diseño RESTful puede ser apropiado cuando las siguientes condiciones se reúnan:

- Los servicios web sean completamente sin estado. Un buena prueba sería considerar si una interacción pudiera seguir sin errores después de un reinicio del servidor.
- Una infraestructura caché pueda ser aprovechada para el funcionamiento. Si los datos que son devueltos por el servicio web no son dinámicamente generados y pueden ser cacheados, la infraestructura caché que los servidores web y otros intermediarios inherentemente proporcionan, pueden ser aprovechadas para la mejora de su funcionamiento. Sin embargo, el desarrollador debe tener cuidado porque tales caché suelen están limitadas al método HTTP GET en la mayoría de los servidores.

- Que tanto el productor como el consumidor del servicio tengan un conocimiento mutuo del contexto y contenido que se estén intercambiando. Debido a que no hay manera alguna de describir la interfaz de los servicios web, ambas partes deben de antemano cumplir con unos esquemas en los que se describen la manera en la que los datos se intercambian y también la forma en la que éstos tienen que ser procesados con sentido. Normalmente, muchas aplicaciones comercializadas que publican sus servicios web implementados en RESTful, también publican un conjunto de herramientas (de valor añadido) que describen las interfaces a los desarrolladores en los lenguajes de programación mayormente utilizados.

- En ocasiones en las que el ancho de banda sea limitado. REST es especialmente útil para dispositivos que habitualmente sufren este tipo de problemas, como PDAs y teléfonos móviles para los que la sobrecarga que producen los elementos XML (de cabeceras y capas adicionales de SOAP) deben ser limitadas.

- En el caso de sitios Web que existan previamente, añadir la prestación o agregación de sus servicios Web, puede hacerse fácilmente mediante el uso de REST. Los desarrolladores pueden utilizar tecnologías tales como JAX-RS y AJAX (*Asynchronous JavaScript with XML*) y también herramientas como DWR (*Direct Web Remoting*) para consumir los servicios en sus aplicaciones web. Los servicios pueden ser expuestos en XML y consumidos mediante HTML sin apenas modificar la arquitectura del sitio web existente. Los desarrolladores son más productivos porque en realidad lo que hacen, es añadir algo de lo que ya les resulta familiar en lugar de tener que partir de cero con otra tecnología nueva.

3.2.1.Uso de servicios web Java

Para hacer una composición de lugar de las características de los servicios web Java que son los más utilizados en la actualidad, he realizado varias búsquedas en la web donde he encontrado un estudio realizado por la compañía *Plumbr*, que es una empresa que ofrece servicios de testeo y monitorización para la mejora del rendimiento de los sistemas a sus clientes. En base al estudio, y a otros datos encontrados, he confeccionado una pequeña lista de las versiones java en uso, de los servidores web java más utilizados habitualmente y también de sitios web donde se ofrece la posibilidad de publicar esta clase de servicios web.

3.2.1.1.Uso de versiones Java

El estudio realizado por la compañía citada anteriormente está basado en la monitorización de sus propios clientes, de los que se ha recogido un muestreo de 758 JVMs (*Java Virtual Machine*) distintas durante los meses de febrero y marzo de 2015. Para poder recopilar los datos han sido utilizadas llamadas al método *System.getProperty()* con el objetivo de obtener los parámetros *os.arch*, *os.version*, *java.version*, etc.

Según los resultados, 2015 ha sido el primer año en el que no se detecta ninguna instancia de la versión 5 de Java. El siguiente gráfico muestra las proporciones en el uso de las versiones 6, 7 y 8 en la actualidad.

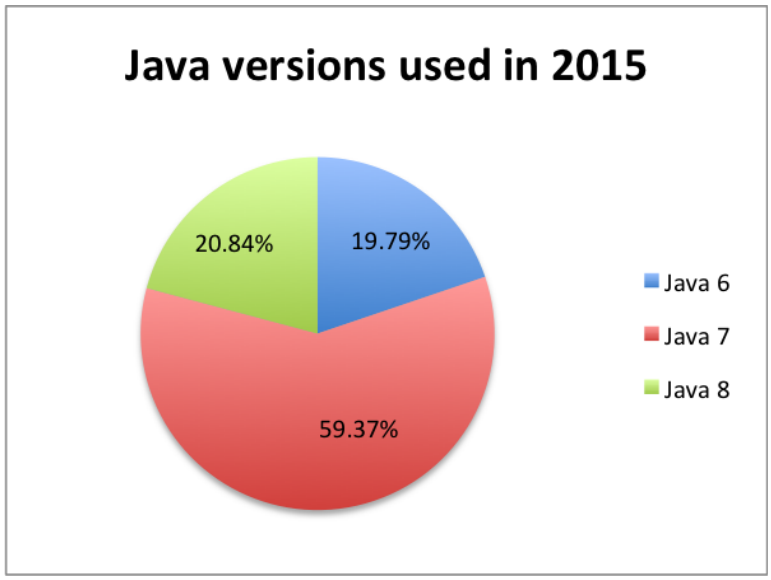

Fig. 3.4: Versiones Java en uso en 2015

Del gráfico inferimos que la mayor parte (59,37%) funcionan actualmente con la versión Java 7 y el resto de la tarta se la reparten entre Java 6 (19,79%) y Java 8 (20,84%) que ya supera la quinta parte del total y sigue en aumento.

En el estudio también se muestra la tendencia del uso de las versiones Java a lo largo del tiempo, se han ido recogiendo datos desde los últimos tres años. La gráfica siguiente muestra estas tendencias de uso:

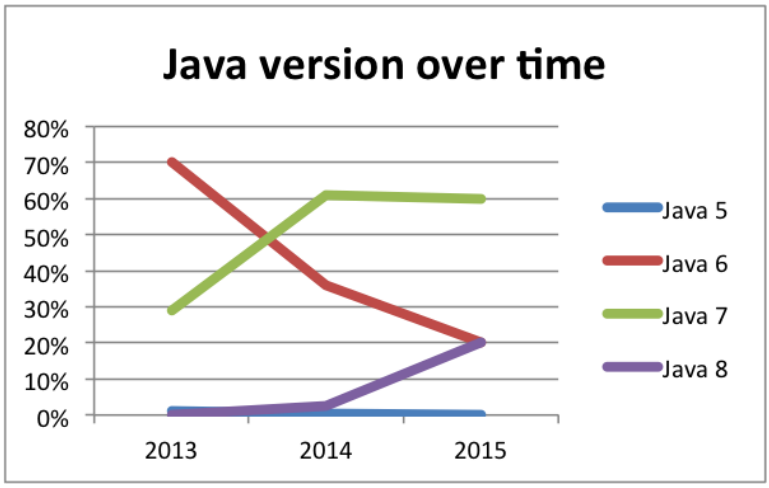

Fig. 3.5: Tendencias de uso en los últimos 3 años

En la gráfica podemos ver cómo este año la versión Java 5 ya queda prácticamente en desuso. Que la versión Java 6 tiende también a caerse, se estima que en el próximo año 2016 baje del 10%, aunque como hemos visto en ésta y en la gráfica anterior casi un 20% la siguen utilizando. Se considera que la versión Java 7 ya ha alcanzado su máximo porcentaje de uso y se estima para el próximo año 2016 que la versión Java 8 ya sea la versión más utilizada.

En el estudio también se detallan datos curiosos como el de que el 20% de las JVMs utilizan versiones "vulnerables", como la *PRE 1.7.0_45*, de la que se conocen más de 100 fallos de seguridad. Son versiones para las que ya existen actualizaciones con parches de seguridad que solucionan las posibles vulnerabilidades pero que, como puede verse, aún no se han aplicado.

3.2.1.2.Uso de servidores web Java

Con respecto a qué servidores web Java son los más utilizados en la actualidad, según el estudio de la compañía anteriormente citada, basado en el mismo muestreo de máquinas (758 JVMs), durante ese periodo de tiempo (marzo – abril 2015), el gráfico de resultados es el siguiente:

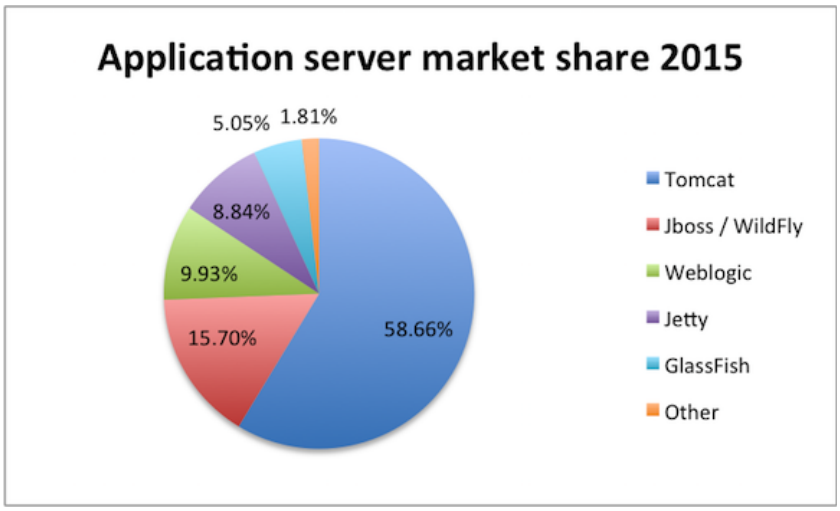

Fig. 3.6: Distribución de servidores en uso 2015

Cabe destacar dos importantes aspectos que han influido muy negativamente en los resultados mostrados:

- De las 758 máquinas del muestreo, solamente ha sido posible identificar con veracidad el modelo de la máquina en 554 casos (73%), ya que en los otros casos no se trataba de servidores de servicios web propiamente dichos como *Scala*, *Groovy*, *Elasticsearch*, *TIBCO*, *Maven*, *IDEA*, *Eclipse*, etc.

- El servidor web de IBM **WebSphere** no aparece en el estudio debido a que los agentes de la compañía que testean a las máquinas (JVM) no son compatibles con las máquinas de IBM, por lo tanto no podemos decir que el estudio sea representativo del mercado actual pero sí aproximado con la ausencia, claro está, del servidor WebSphere de IBM. Aunque, de todos modos, en otros estudios se le asigna una parte del pastel no superior al 4%, que no llega a ser demasiado relevante.

Del gráfico podemos sacar varias conclusiones:

- **Tomcat** es el servidor (de Servlets, pero con motores de funcionalidad añadidos) que supera la mitad del mercado de servidores en uso.

- **WildFly** (o Jboss) ostenta el 15,7% del mercado.

- **WebLogic** (de Oracle) se hace casi con el 10%.

- **Jetty** el 8,8%.

- **GlassFish** el 5%.

- El resto (2%) de los servidores se compone de una amplia variedad de la que caben destacar **Resin** y **OC4J**.

En el estudio también se muestra la tendencia del uso de los servidores Java a lo largo del tiempo, se han ido recogiendo datos desde los últimos tres años. La gráfica siguiente muestra estas tendencias de uso:

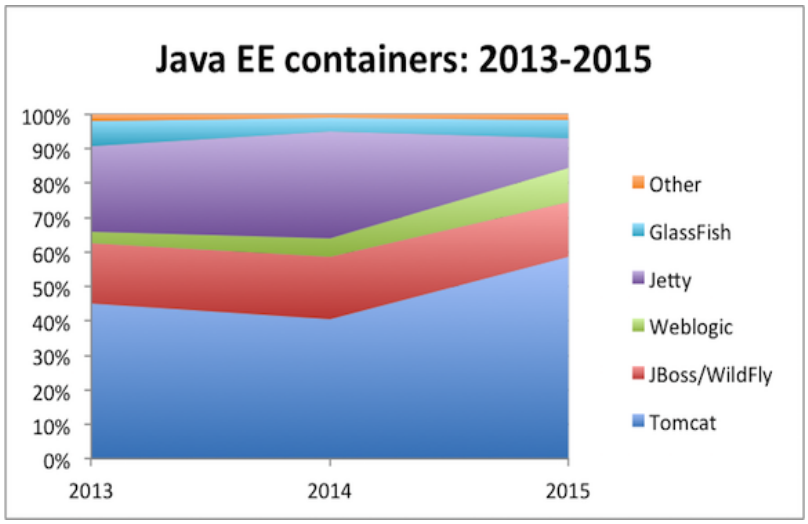

Fig. 3.7: Tendencias de uso en los últimos 3 años

En estos últimos tres años ha pasado lo siguiente, se han producido tres cambios importantes:

- **WebLogic** de Oracle ha duplicado su presencia en el mercado, pero queda en un cuarto lugar.
- **Jetty** ha caído de ser el segundo a ser tercero en benficio de **WildFly** (Jboss).
- **Tomcat**, además de seguir siendo el más utilizado, ha aumentado su ventaja con respecto al resto.

3.2.2.Apache Tomcat

Dado que Tomcat es el servidor de servicios web más utilizado en el mercado, y que además es el utilizado para la infraestructura de mi servicio web, voy a incluir una breve información del servidor para tratar de hacer más comprensible el porqué de su éxito en la actualidad.

A diferencia de lo que muchos creen, Tomcat no es un servidor de aplicaciones propiamente dicho. Tomcat es realmente un contenedor de *servlets* (y servidor web) de código abierto desarrollado por la fundación de software de Apache (ASF) que implementa varias especificaciones de Java EE como *Java Servlet*, *Java Server Pages* (JSP), Java EL, etc. Y

proporciona un entorno servidor web HTTP en "Java puro" para la ejecución de código Java. Tomcat está siendo desarrollado y mantenido por una comunidad de desarrolladores de código libre bajo el auspicio de la fundación de software Apache.

En sus inicios, Tomcat empezó siendo una aplicación de referencia *Servlet* implementada por un desarrollador de arquitecturas en *Sun Microsystems* llamado James Duncan Davidson. El mismo Davidson, quien ayudó realizando un importante papel durante el proceso de donación de Tomcat por parte de *Sun Microsystems* a la fundación Apache, quiso bautizar el proyecto con el nombre de un animal porque todos los libros de O'Reilly relacionados con proyectos de código abierto tenían un animal en la portada. Y la razón por la que escogió como animal a un gato fue debido a que éstos representan a una especie de animales que pueden valerse por sí mismos. En 2003, Davidson ve sus sueños cumplidos con el lanzamiento del libro "*Tomcat: The Definitive Guide*" de O'Reilly en el tiene en su portada a un leopardo de las nieves.

La tabla que se muestra a continuación resume un informe con las características que el servidor ha ido ofreciendo a lo largo del tiempo:

Series	Declaradas estables	Descripción	Última versión	Fecha
3.0	1999	Versión inicial. Fusión de código donado: Servidor Web de Sun Java y ASF Especificaciones implementadas: Servlet 2.2 y JSP 1.1	3.3.2	2004-03-09
4.1	2002-09-06	Especificaciones implementadas: Servlet 2.3 y JSP 1.2	4.1.40	2009-06-25
5.0	2003-12-03	Especificaciones implementadas: Servlet 2.4, JSP 2.0 y EL 1.1	5.0.30	2004-08-30
5.5	2004-11-10		5.5.36	2012-10-10
6.0	2007-02-28	Especificaciones implementadas: Servlet 2.5, JSP 2.1, and EL 2.1	6.0.43	2014-11-22
7.0	2011-01-14	Especificaciones implementadas: Servlet 3.0, JSP 2.2 y EL 2.2 specifications.	7.0.61	2015-04-07
8.0	2014-06-25	Especificaciones implementadas: Servlet 3.1, JSP 2.3, EL 2.3 y WebSocket.	8.0.21	2015-03-26

En la tabla anterior se aprecia que en la primera columna el fondo de las casillas está coloreado. Los colores tienen el siguiente significado:

- Las filas cuya primera casilla tienen el fondo de color naranja se refieren a series de versiones de Tomcat obsoletas.
- Las que tienen el fondo de color amarillo son series de versiones también obsoletas pero que tienen aún soporte.
- La última, la de color verde, es la serie más actual.

Las características de Tomcat que se destacan como ventaja para ser elegido como el servidor utilizado son las siguientes:

- Es bastante simple y de fácil configuración.
- Es de código abierto.
- No hace un consumo excesivo de los recursos del sistema, es bastante ligero.
- Ofrece una gran modularidad. Al servidor se le pueden añadir diferentes módulos de configuración para poder agregarle más funcionalidad. En este aspecto es muy flexible.
- Existen muchos proveedores de complementos que trabajan para Tomcat, lo que facilita la programación en entornos con éste servidor.

Tal y como he indicado al principio de este apartado (apartado dedicado a Tomcat), el servidor Tomcat no es un servidor de aplicaciones, es básicamente un contenedor de *servlets* que puede adquirir nuevos roles gracias a la posibilidad que ofrece de poder ser configurado agregándole módulos o motores. Gracias a esta posibilidad, para poder ofrecer servicios web SOAP mediante un servidor Tomcat basta con agregarle el módulo correspondiente. Para la construcción del servicio web de rastreo de vuelos que he creado y poder ofrecerlo mediante SOAP, he añadido un módulo llamado Axis2.

A continuación se describe en qué consiste tal módulo.

3.2.3.Apache Axis2

Apache Axis2 es un motor nuclear para servicios web que se presentó durante una cumbre realizada en Colombo (Sri Lanka) en 2004. Es un rediseño total y una evolución completa de la anterior arquitectura "Apache Axis" que añade más flexibilidad, eficiencia y capacidad de configuración. Existen versiones de Axis2 tanto en Java como en C.

Axis2 no sólo provee la capacidad de agregar servicios web a las aplicaciones web, sino que además puede funcionar como servidor autónomo. Además de ofrecer soporte para SOAP 1.1 y SOAP 1.2, también integra soporte para el popular estilo REST para servicios web ampliamente utilizado en la actualidad con el auge del uso de clientes en dispositivos móviles.

Apache Axis2 también incluye soporte para los siguientes estándares:

- WS - *ReliableMessaging* - mediante Apache Sandesha2.
- WS - *Coordination* - mediante Apache Kandula2.
- WS - *AtomicTransaction* - mediante Apache Kandula2.
- WS - *SecurityPolicy* - mediante Apache Rampart.
- WS - *Security* - mediante Apache Rampart.
- WS - *Trust* - mediante Apache Rampart.
- WS - *SecureConversation* - mediante Apache Rampart.
- SAML 1.1 - mediante Apache Rampart.
- SAML 2.0 - mediante Apache Rampart.
- WS - *Addressing* - módulo incluido como parte de Axis2 core.

Este motor incluye las características suficientes junto con el servidor Tomcat, para poder llevar a cabo el objetivo de ofrecer el servicio de rastreo de vuelos mediante SOAP.

3.3.Aplicaciones móviles

Una aplicación móvil o app (término adquirido del inglés) es una aplicación informática diseñada e implementada para ser ejecutada en teléfonos inteligentes, tabletas y otros dispositivos móviles. Para comprender un poco mejor el concepto, podemos decir que las aplicaciones son a los móviles lo que los programas son a los ordenadores de sobremesa. Generalmente se encuentran disponibles para los clientes a través de plataformas de distribución, gestionadas por las compañías de los sistemas operativos móviles, como Android, iOS, Windows Phone, BlackBerry OS, entre otros. Existen aplicaciones móviles gratuitas (la mayoría con ventanas de publicidad incrustadas) o de pago. Normalmente, el 20-30% (en promedio) del costo de cada aplicación se destina al distribuidor (la plataforma de distribución) y el resto es para el personal desarrollador.

3.3.1.Tipos de aplicaciones móviles

A nivel de desarrollo, existen varias formas de programar las aplicaciones. Cada una de la formas, tiene diferentes características y limitaciones, sobretodo desde el punto de vista técnico. Aunque a primera vista esto no parezca incumbencia del diseñador, la realidad es que el tipo de aplicación que se elija, condicionará el diseño visual y la interacción.

Existen básicamente tres tipos de aplicaciones móviles según la manera de programarlas:

- Aplicaciones móviles nativas.
- Aplicaciones móviles web.
- Aplicaciones móviles híbridas.

3.3.1.1.Aplicaciones móviles nativas

Las aplicaciones móviles nativas son aquellas que han sido desarrolladas específicamente con un software que se ofrece en cada sistema operativo para poder ser utilizado por los programadores que se lo descarguen, llamado genéricamente *Software Development Kit* o SDK. De ese modo, Android programado en Java, iOS programado en *Objective C* y Windows Phone programado en *.Net*, tienen un entorno de desarrollo software diferente (SDK) donde las aplicaciones nativas se diseñan y programan específicamente para cada plataforma.

A continuación se indica un resumen de las ventajas e inconvenientes que ofrecen ésta clase de aplicaciones móviles:

Ventajas:

- Acceso completo al dispositivo. Las aplicaciones están realmente integradas al teléfono, lo cual les permite utilizar todas las características de hardware del terminal, como la cámara de fotos y vídeo, los sensores (GPS, acelerómetro, giróscopo, entre otros) y también el servicio de notificaciones.

- Mejor experiencia del usuario. No requieren Internet para funcionar, y al estar integradas en el dispositivo, ofrecen una experiencia de uso más fluida.

- Fácil visibilidad y actualización. Dado que las aplicaciones pueden estar disponibles en las plataformas de distribución, estas aplicaciones y sus actualizaciones pueden ser fácilmente accesibles para los usuarios de la plataforma.

Inconvenientes:

- Se requieren diferentes herramientas de desarrollo. Para cada plataforma de desarrollo se utilizan diferentes herramientas de programación o SDK.

- El código cliente no es reutilizable para cada plataforma. Como hemos indicado anteriormente, Android se desarrolla en Java, IOS en *Objective* C, etc. Con lo que el código fuente no es reutilizable entre diferentes plataformas.

3.3.1.2.Aplicaciones móviles web

Las aplicaciones móviles web o también llamadas "*webapps*" tienen HTML como lenguaje base de programación conjuntamente con JavaScript y CSS, que son herramientas ya mayormente conocidas por los programadores web.

En este caso se programa de manera independiente al sistema operativo (en el cual se utilizará la aplicación) y no se emplea un SDK en concreto. Las aplicaciones web no necesitan instalarse, ya que se visualizan usando el navegador del teléfono como un sitio web normal, por eso estas aplicaciones pueden ser fácilmente utilizadas en diferentes plataformas sin mayores inconvenientes y sin necesidad de desarrollar un código diferente para cada caso particular.

Estas características ofrecen las siguientes ventajas e inconvenientes:

Ventajas:

- El mismo código base sirve para múltiples plataformas.
- Proceso de desarrollo más sencillo.
- No es necesario ninguna aprobación externa para poder publicarse.
- El usuario siempre dispone de la última versión de la aplicación.
- Pueden reutilizarse sitios "*responsive*" ya diseñados.

Inconvenientes:

- Se requiere de conexión a Internet.
- Acceso muy limitado a los elementos y características del hardware del dispositivo.
- La experiencia del usuario (navegación, interacción...) y el tiempo de respuesta es menor que en una aplicación nativa.
- Se requiere de mayor esfuerzo en promoción y visibilidad, ya que las aplicaciones en este caso no están publicadas en una plataforma de distribución.

3.3.1.3.Aplicaciones móviles híbridas

Este tipo de aplicaciones es una especie de combinación entre las dos anteriores. La forma de desarrollarlas es similar a la de una aplicación web (usando HTML, CSS y JavaScript), aunque a diferencia de las aplicaciones web, éstas permiten acceder (usando librerías) a las capacidades del teléfono, tal como lo haría una aplicación móvil nativa. Y una vez que la aplicación está terminada, se compila o empaqueta de forma tal, que el resultado final es como si se tratara de una aplicación nativa.

Esto permite casi con un mismo código obtener diferentes aplicaciones, por ejemplo, para Android y iOS, y distribuirlas en cada una de sus tiendas.

A continuación se indica un resumen de las ventajas e inconvenientes que ofrecen ésta clase de aplicaciones móviles:

Ventajas:

- Es posible distribuirlas en las tiendas de IOS y Android.
- Se trata de una instalación nativa pero desarrollada con lenguajes más populares como JavaScript, HTML o CSS. Con lo que el desarrollo de estas aplicaciones está al alcance de más programadores.
- El mismo código base sirve para diferentes plataformas. Esto supone un gran ahorro del tiempo empleado para su desarrollo.
- Acceso a parte (no es total) del hardware del dispositivo. A diferencia de las aplicaciones móviles web, en estas aplicaciones sí es posible.

Inconvenientes:

- Experiencia del usuario más propia de la aplicación web que de la aplicación móvil nativa. Dado que las aplicaciones son ejecutadas en los navegadores web de los dispositivos.
- Diseño visual no siempre relacionado con el sistema operativo en el que se muestre. Ya que no se tratan de aplicaciones móviles específicas para cada sistema, existen ciertas incompatibilidades para cada caso.

3.3.2.Uso de Aplicaciones móviles

Para hacernos una idea de las características de las aplicaciones móviles que son las más utilizados en la actualidad y de las tendencias entre los desarrolladores, he realizado varias búsquedas en la web donde he encontrado un artículo realizado por la compañía *VisionMobile*, que es la empresa líder de investigación en la economía de las aplicaciones móviles. El artículo llamado "*Developer Economics Q1 2015 report*" trata las tendencias en desarrollo, de las plataformas utilizadas, herramientas de desarrollo, etc. y es la octava edición de una serie de artículos que muestran el estado de la comunidad de los desarrolladores a nivel global mediante un sondeo. En esta ocasión, el sondeo se ha realizado a más de 8.000 encuestados de 143 países diferentes.

Del artículo, he estimado oportuno destacar los datos referentes a porcentajes sobre la dedicación (para la plataforma que programan) de los desarrolladores con respecto al tiempo (el último año y lo que llevamos de éste) y también con respecto a su ubicación geográfica.

Siguiendo la evolución temporal tenemos el siguiente gráfico:

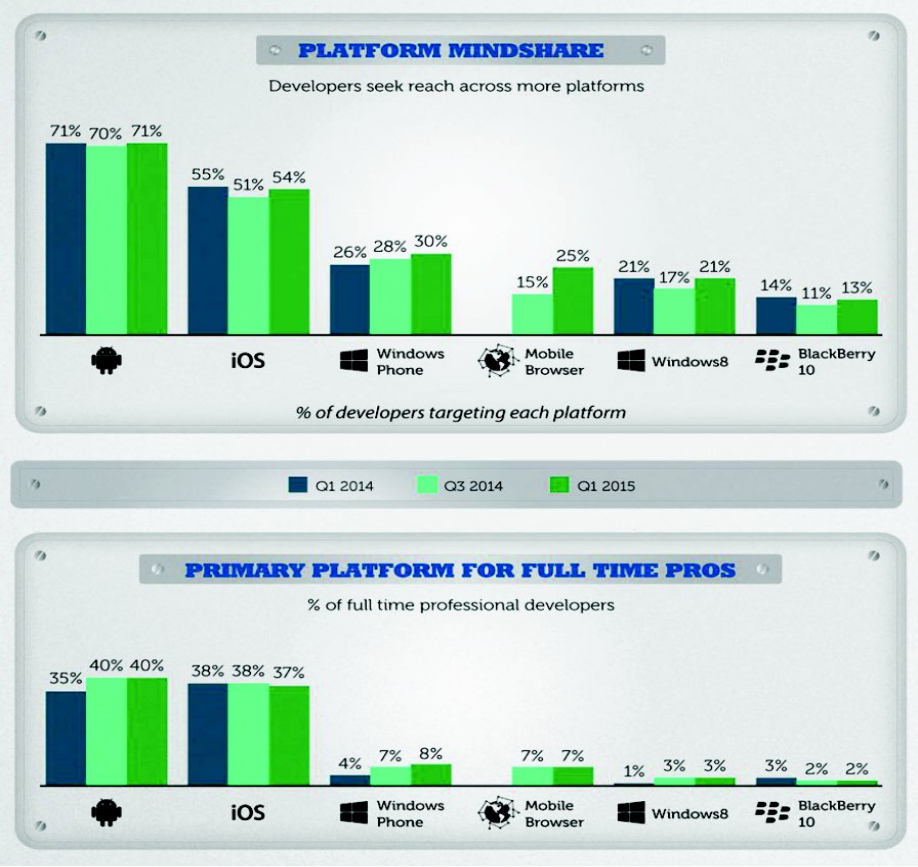

Fig. 3.8: % de S.O. elegidos por los programadores desde 2014.

En la primera gráfica se indican los porcentajes de los programadores que desarrollan para cada plataforma y en la segunda los porcentajes de los programadores, que son profesionales a tiempo completo, y que programan para una plataforma en concreto. Existen tres barras, cada una con un color distinto que representan a un cuatrimestre diferente, el primero del 2014 azul, el tercero del 2014 verde claro y el primer cuatrimestre del 2015 en verde oscuro.

Según indican las gráficas podemos afirmar que la mayoría de los programadores lo hace para la plataforma Android con ligeras diferencias con respecto al tiempo, parece que la batalla entre las plataformas Android e IOS se ha ido estabilizando en el último año. Teniendo cada plataforma su parte del mercado asegurado.

Siguiendo la evolución geográfica tenemos el siguiente gráfico:

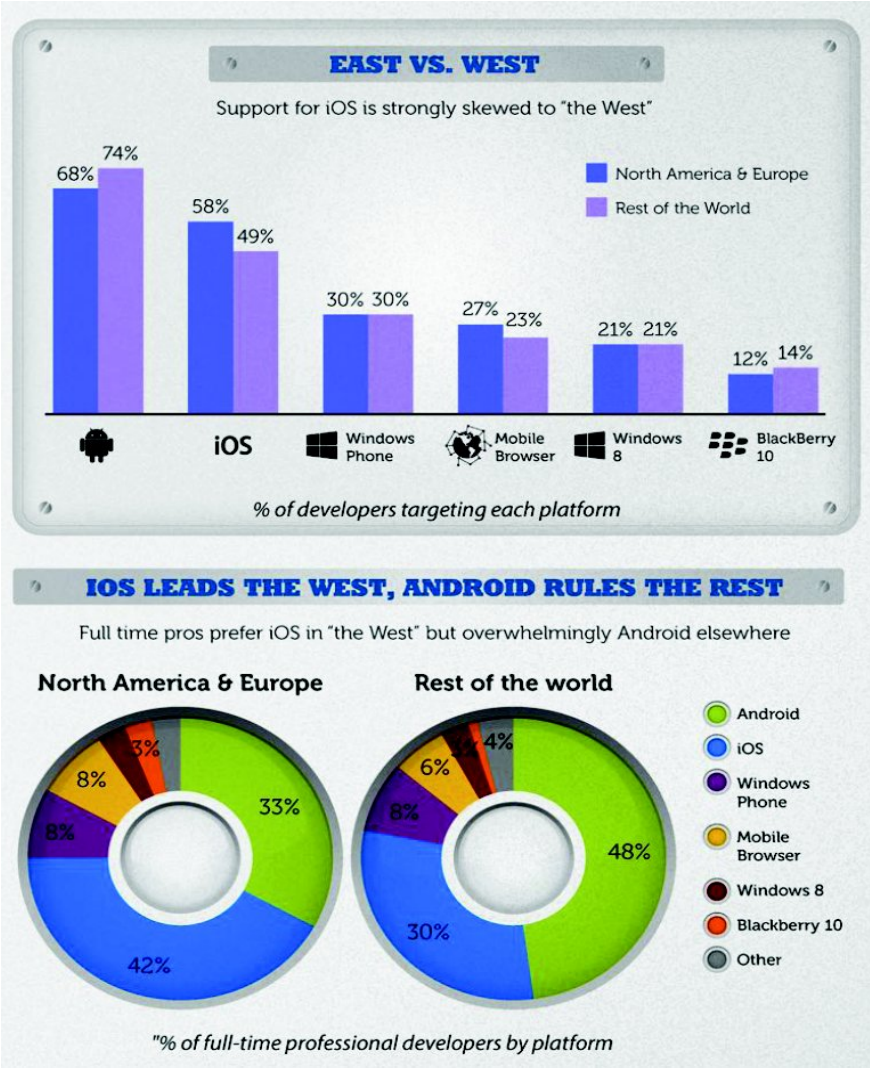

Fig. 3.9: % de S.O. elegidos por programadores en el mundo.

En la primera gráfica se indican los porcentajes de los programadores que desarrollan para cada plataforma con respecto a su ubicación geográfica. Se diferencian dos grandes grupos, por un lado el grupo (en azul) de los desarrolladores para las zonas de América del Norte y Europa y por el otro (en morado), el grupo de desarrolladores que lo hacen para el resto del mundo. Mientras que en la segunda gráfica se indica el porcentaje de los desarrolladores trabajando a tiempo completo específicamente para cada plataforma, todos ellos clasificados dentro de cada uno de los dos grupos definidos en la gráfica anterior, de nuevo América del Norte y Europa comparado con el resto del mundo.

Según ambas gráficas, podemos afirmar que la mayoría de los desarrolladores, tanto en América del Norte como en el resto del mundo, lo hacen para la plataforma Android. Pero la segunda gráfica nos muestra un matiz diferente, la mayoría de los programadores que trabajan a tiempo completo para un determinada plataforma en la zona de América del Norte y Europa los hacen para la plataforma IOS. En el informe justifica este hecho porque se obtienen más beneficios en la zona de América del Norte y Europa programando únicamente para IOS. Mientras que en resto del mundo, la mayoría de los profesionales especializados en una plataforma lo hacen para Android.

Como conclusión a los datos aportados, podemos afirmar que la mayoría del mercado, es decir, la mayor demanda de aplicaciones existente se produce en la plataforma Android, ya que la mayoría de los desarrolladores obtienen sus beneficios programando para esta plataforma y no existen ninguna tendencia (hasta la fecha) que indique lo contrario.

3.3.3.Uso de sistemas operativos móviles

Así como lo he hecho con el uso de las aplicaciones móviles, también he estimado oportuno hacer un rápido sondeo por internet para tratar de averiguar en este caso, cuál es la tendencia de uso en el mercado global a nivel de plataforma o de sistema operativo, es decir, intentar saber cual es el sistema operativo más comúnmente utilizado a nivel mundial en los dispositivos móviles.

De los sitios web que ofrecen esa información encontrados, he elegido *NetMarketShare* por la metodología con la que dicen que obtienen los datos.

Publican en su web que recopilan datos de los navegadores clientes (dato con el que se puede averiguar el sistema operativo del dispositivo también) que visitan más de 40.000 sitios web distribuidos por todo el planeta. Su metodología es la de contar únicamente a los visitantes de las páginas web, es decir, si el mismo cliente visita un sitio web dos veces en el mismo día contaría a efectos estadísticos como una sola visita. Tratando de evitar de ese modo un posible fraude y recopilar unos datos lo más aproximados a la realidad posible.

Los datos obtenidos para un período de tiempo reciente como pueda ser el durante el pasado mes de abril son los siguientes:

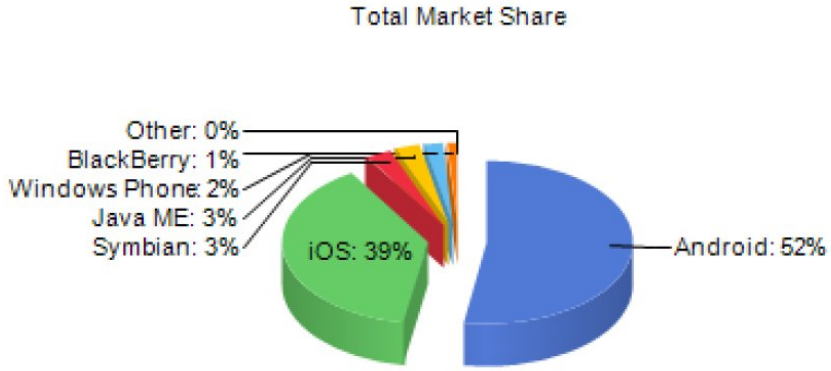

Fig. 3.10: Distribución del uso de SO en el mes de abril 2015.

La siguiente tabla muestra los datos con más detalle:

Sistema Operativo	Distribución en abril de 2015
Android	52.47%
iOS	38.81%
Symbian	2.79%
Java ME	2.78%
Windows Phone	2.07%
BlackBerry	0.96%
Kindle	0.06%
Samsung	0.03%
Bada	0.01%
Windows Mobile	0.00%
LG	0.00%

Los resultados obtenidos a partir de datos recopilados durante el mes de abril de 2015 indican que la plataforma más utilizada en dispositivos móviles es sin lugar a dudas Android, siendo más de la mitad (52.47%) de los dispositivos utilizados actualmente.

Pero si hacemos un estudio con respecto al tiempo durante los últimos dos años podemos observar lo siguiente:

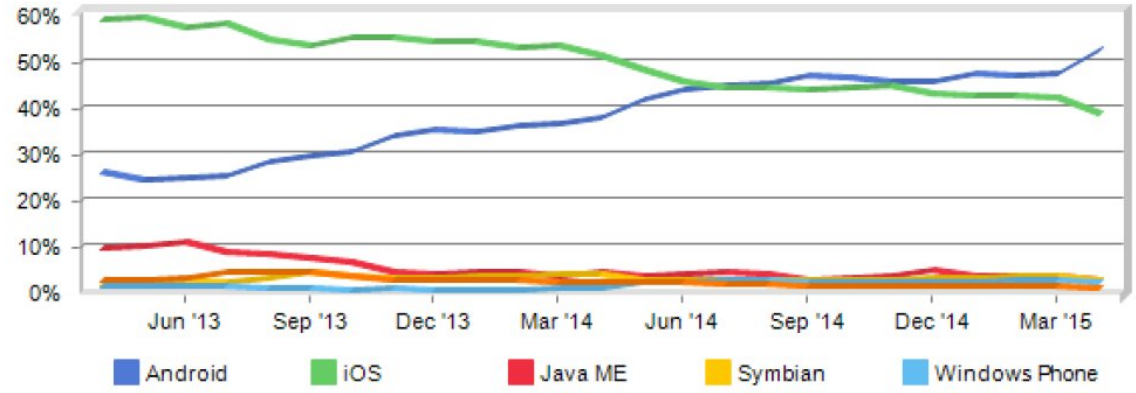

Fig. 3.11: Distribución del uso de SO desde abril 2013.

El sistema operativo más utilizado anteriormente era IOS, y lo venia siendo desde que Apple lanzó su iPhone alrededor del 2008. Pero como puede verse en la gráfica, a partir del verano de 2014 IOS cede el primer puesto a Android. Pero este cambio de tendencia no se debe a que los usuarios cambien sus dispositivos iOS por otros con Android (generalmente), si no que se debe al aumento del uso de los dispositivos móviles. Ya que la mayor parte de los dispositivos que se adquieren son de gama media o baja y vienen equipados con Android, que es de código abierto.

3.3.4.El sistema operativo Android

Dado que Android es el sistema operativo para dispositivos móviles más utilizado en el mercado, y que además es el que he utilizado para crear el cliente de mi servicio web, voy a incluir una breve información de la plataforma para tratar de hacer más comprensible el porqué de su éxito en la actualidad.

Android es un sistema operativo que inicialmente fue diseñado con el objetivo de gestionar recursos hardware de dispositivos móviles con pantalla táctil, como teléfonos móviles o tabletas, pero con el paso del tiempo se ha ido adaptando a las nuevas necesidades del mercado y también se ofrece para ser ejecutado en dispositivos novedosos como relojes inteligentes, o televisores y automóviles.

Android fue desarrollado por la compañía Android Inc. que fue adquirida por Google en 2005. El sistema operativo fue presentado por vez primera en 2007, junto la fundación del *Open Handset Alliance* (un consorcio de compañías de hardware, software y telecomunicaciones) para avanzar en los estándares abiertos de los dispositivos móviles donde Google liberó la mayor parte del código bajo la licencia Apache, que es una licencia libre y de código abierto.

La arquitectura está clasificada en cuatro capas según sus funcionalidades, todas ellas basadas en software libre.

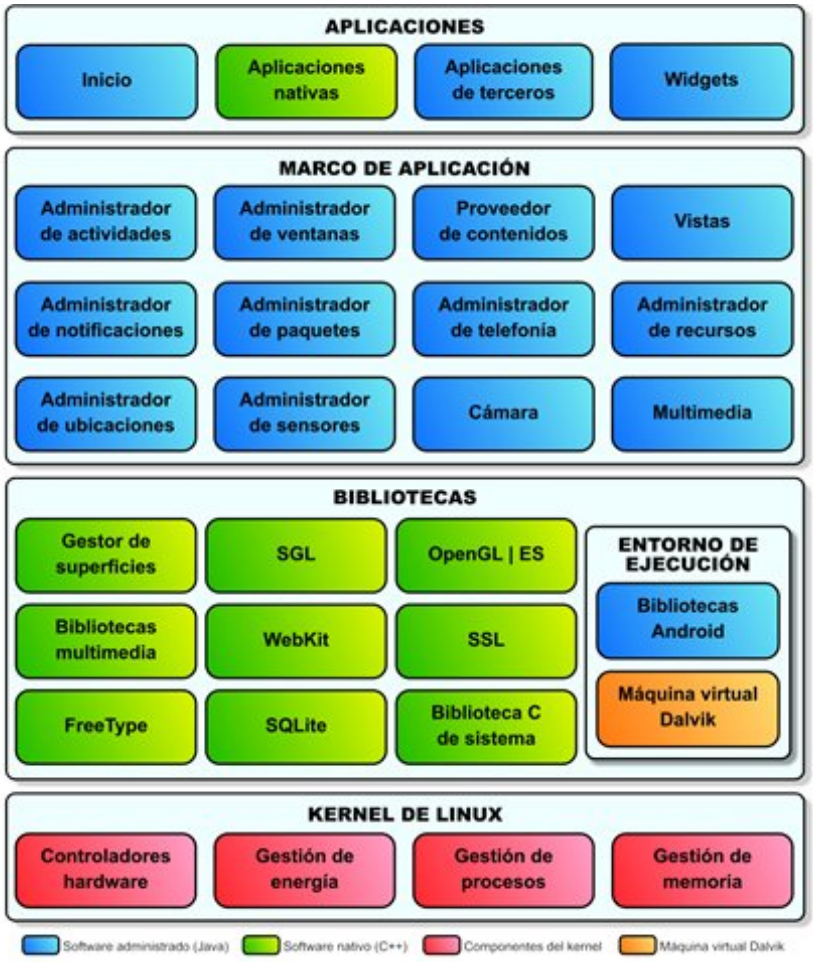

Fig. 3.12: Pila de capas en la arquitectura de Android.

- El kernel de Linux. Esta capa del modelo actúa como capa de abstracción entre el hardware y el resto de la pila, es dependiente del hardware. Esta capa proporciona servicios como la seguridad, el manejo de la memoria, el multiproceso, la pila de protocolos y es soporte de drivers para dispositivos.

- El entorno de ejecución. Está basado en el concepto de máquina virtual utilizado en java, aunque Google tomó la

decisión de crear una nueva que respondiera mejor a las limitaciones de los dispositivos en los que había de ejecutarse Android, con poca memoria y procesador limitado, la máquina virtual *Dalvik*.

- Bibliotecas o librerías nativas. Son un conjunto de librerías desarrolladas en C/C++ usadas en varios componentes de Android compiladas en código nativo del procesador. Muchas de las librerías utilizan proyectos de código abierto. Algunas de las librerías son *WebKit*, *FreeType*, *SQLite*, *SSL*, etc.

- Marco o entorno de aplicación. Proporciona una plataforma de desarrollo libre para aplicaciones con una gran riqueza e innovaciones (sensores, localización, servicios, etc.). Esta capa ha sido diseñada para simplificar la reutilización de componentes. Las aplicaciones pueden publicar sus capacidades y otras pueden hacer uso de ellas (sujetas a las restricciones de seguridad). Los servicios más importantes que incluye son *Views*, *Resource Manager*, *Activity Manager*, *Notification Manager* y *Content Provider*.

- Aplicaciones. Este nivel está formado por el conjunto de aplicaciones instaladas en una máquina Android. Todas las aplicaciones han de ejecutarse en la máquina virtual Dalvik para garantizar la seguridad del sistema. Normalmente, las aplicaciones Android están escritas en Java.

Las versiones de Android que podemos encontrar funcionando en dispositivos en la actualidad son las siguientes:

Versión	Nombre en código	Fecha de distribución	API level
5.1	*Lollipop*	06 de abril de 2015	22
5.0	*Lollipop*	3 de noviembre de 2014	21
4.4	*Kit Kat*	31 de octubre de 2013	19
4.3	*Jelly Bean*	24 de julio de 2013	18
4.2	*Jelly Bean*	13 de noviembre de 2012	17
4.1	*Jelly Bean*	9 de julio de 2012	16
4.0	*Ice Cream Sandwich*	16 de diciembre de 2011	14
2.3	*Gingerbread*	9 de febrero de 2011	10
2.2	*Froyo*	20 de mayo de 2010	8

Como podemos ver en la tabla, la versión más reciente del sistema operativo es la 5.1

Android, al contrario que otros sistemas operativos para dispositivos móviles como iOS o Windows Phone, se desarrolla de forma abierta y se puede acceder tanto al código fuente como a la lista de incidencias donde se pueden ver problemas aún no resueltos y reportar problemas nuevos.

El desarrollo de aplicaciones para Android no requiere aprender lenguajes complejos de programación. Todo lo que se necesita es un conocimiento aceptable de Java y estar en posesión del kit de desarrollo de software o «SDK» provisto por Google el cual se puede descargar gratuitamente.

Los usuarios de Android disponen de una plataforma en línea de software desarrollado por Google para dispositivos Android, a la que pueden acceder mediante una aplicación. Dicha plataforma, conocida con el nombre de Google Play, es la que alberga mayor cantidad de aplicaciones comparada con el resto, y además, la mayoría de éstas son gratuitas aunque contienen publicidad de terceros.

Estas características, además de otras menos relevantes, hacen que Android se haya convertido en un modelo a seguir por desarrolladores de tendencias o negocios importantes y por lo tanto, sea la plataforma que se ha elegido para el desarrollo de la aplicación para el rastreo de vuelos de la tesina.

Para la construcción del servicio web de rastreo de vuelos que he creado y poder consumirlo mediante SOAP, he tenido que añadir una librería llamada KSoap2 a la aplicación cliente. A continuación explicamos brevemente la librería.

3.3.5.KSoap2-Android

Ksoap y *Ksoap2* eran proyectos que implementaban la pila de servicios web SOAP hasta que quedaron abandonados desde aproximadamente 2006. Posteriormente, los desarrolladores de Android optaron por adaptar este proyecto (ya que era bastante ligero) a su sistema, y también a realizar nuevas versiones de librerías para poder comunicarse con servidores mediante servicios SOAP.

Actualmente el proyecto es llamado *ksoap2-Android* y proporciona un librería cliente SOAP de ligera y eficiente para la plataforma Android, aunque también funciona en otras plataformas que utilicen librerías Java.

Su versión más reciente es la 3.4.0 y es la librería más comúnmente utilizada entre los desarrolladores de Android ya que es de código abierto y, por lo tanto, puede también obtenerse gratuitamente desde el sitio web de Google.

4.Arquitectura del proyecto:

En este capítulo se explica el proyecto a grandes rasgos y de una manera lógica y funcional, sin entrar en los detalles del código que lo implementa. Se realizará un despiece de cada uno de los módulos que componen el proyecto indicando sus características y su funcionalidades.

La siguiente figura nos muestra un esquema general y básico de los componentes del proyecto dentro del escenario que es la red interna de la UPV.

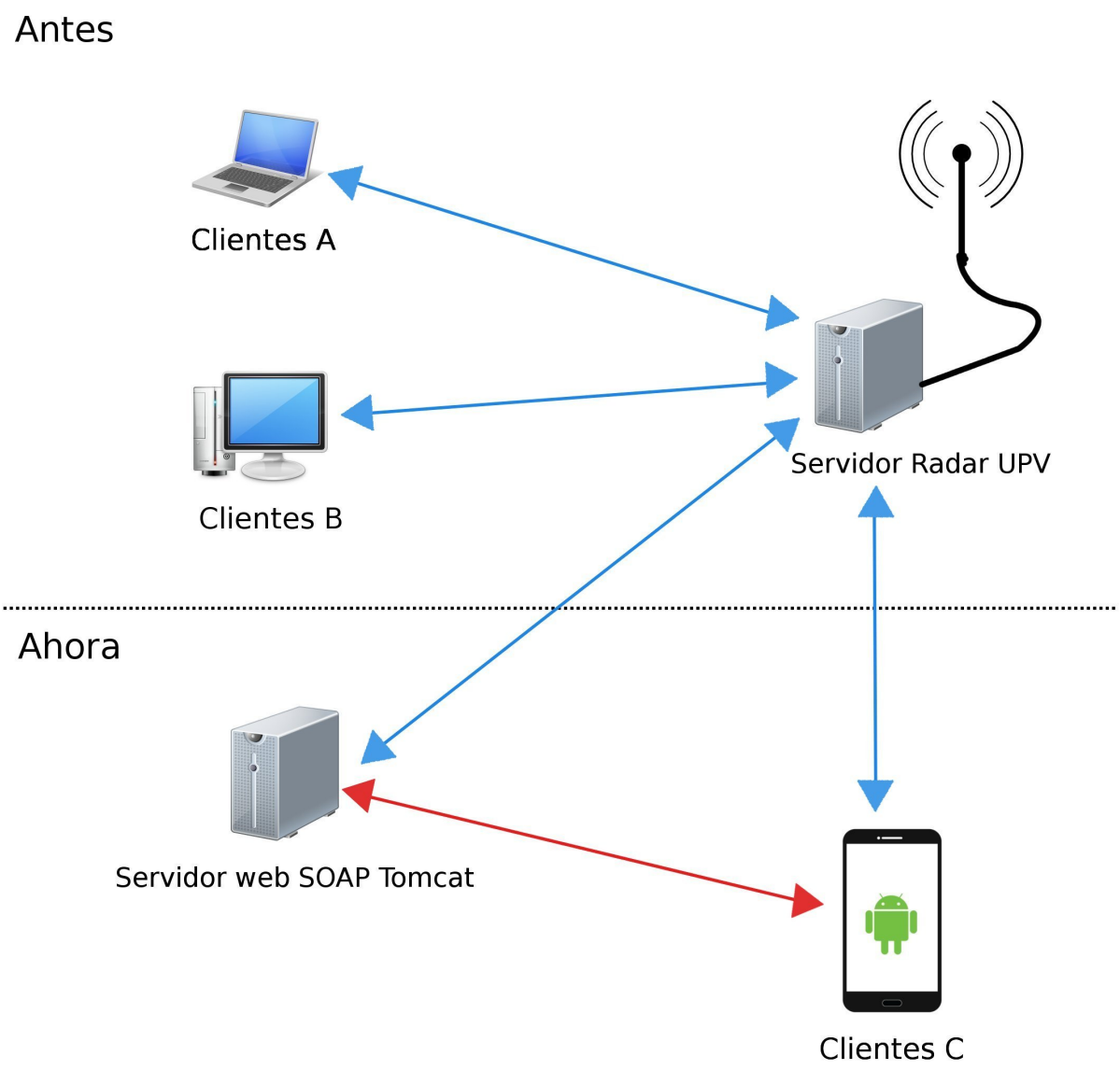

Fig. 4.1: Esquema general de los componentes del proyecto.

El esquema, como puede apreciarse, está dividido en dos partes separadas por una línea discontinua negra. La parte superior (etiquetada como "Antes") representa al estado del contexto antes de la realización del proyecto, en la que únicamente existía una antena para el rastreo de vuelos en tiempo real, que conectada a una máquina servidor, ofrecía los datos (trazo azul en el esquema) sobre la posición de los vuelos que se detectaban a clientes que eran aplicaciones de escritorio desarrolladas en java, preparadas específicamente para conectarse a ese servidor. Tales clientes, únicamente podían ser ejecutados en ordenadores portátiles o de sobremesa, ya que las máquinas virtuales Java sólo funcionan mayormente en esta clase de equipos.

Con la finalidad de cumplir con los objetivos que se plantean en el proyecto de estandarizar el envío de los datos mediante el uso de servicios web SOAP, y de que éstos sean accesibles a los dispositivos más comúnmente utilizados en la actualidad (los teléfonos móviles), se añaden dos componentes al escenario.

La parte inferior del esquema (etiquetada como "ahora") representa el estado posterior a la realización del proyecto, en la que se puede ver un nuevo servidor que obtiene los datos de los vuelos de la misma forma que los clientes anteriores (trazo azul), y que a su vez los envía mediante el uso de SOAP (trazo rojo).

El otro componente añadido, al que nos hemos referido anteriormente, representa a los clientes Android que pueden acceder con la aplicación del proyecto instalada su sistema. Esta aplicación cliente Android desarrollada en el proyecto, ofrece la posibilidad de poder obtener los datos del nuevo servidor web SOAP (trazo rojo) Y también obtenerlos directamente del servidor de la antena que ya funcionaba anteriormente (trazo azul).

4.1.Especificaciones funcionales del proyecto

Tal y como se indica en el apartado 1.3 del primer capítulo que trata de los objetivos, las especificaciones del proyecto son la de crear un servicio web SOAP que ofrezca los datos del tráfico aéreo del servidor/antena que existe actualmente en la UPV, y también la de desarrollar un cliente que pueda ejecutarse en dispositivos con la plataforma Android que sea capaz de consumir el servicio web SOAP creado, y que además, éste sea compatible con el servidor/antena, para de ese modo poder seguir el tráfico aéreo en tiempo real desde los dispositivos móviles Android, aunque el servidor web SOAP no se encuentre operativo en esos momentos.

En los siguientes apartados se explica con más detalle las especificaciones de cada uno de los componentes que componen el proyecto.

4.1.1.Esp. funcionales del servicio web SOAP

Las especificaciones funcionales del servicio web son como podría esperarse, diferentes a las típicas de un "*front-end*", es decir, el usuario no interactúa directamente con el servidor ya que éste no genera la interfaz funcional de la aplicación, sino que sería más bien un eslabón de la cadena que forma parte de la infraestructura "*back-end*" encargada de hacer peticiones (haciendo el rol de cliente) al servidor/antena para recibir los datos de los aviones detectados y volver a enviarlos en formato XML y siguiendo los patrones de la comunicación SOAP.

El servicio funciona bajo demanda (por motivos de eficiencia), es decir, el servidor web SOAP no está constantemente obteniendo los datos del otro servidor/antena para así tener una especie de base de datos con la información actualizada.

Su funcionamiento real es el siguiente:

- El servidor web SOAP recibe un petición de algún cliente y la procesa.
- El servidor web SOAP lanza una petición al servidor/antena.
- El servidor/antena recibe la petición, la procesa y envía la respuesta.

- El servidor web SOAP recibe la respuesta y la procesa.
- El servidor web SOAP responde al cliente con los datos en formato XML SOAP.

Los paso anteriores para servir la información únicamente serán realizados en caso de recibirse la petición desde algún cliente, evitando de ese modo un procesamiento redundante en el servidor.

Con todo esto tendríamos el siguiente esquema WSDL generado en Eclipse sobre el funcionamiento del servidor web SOAP.

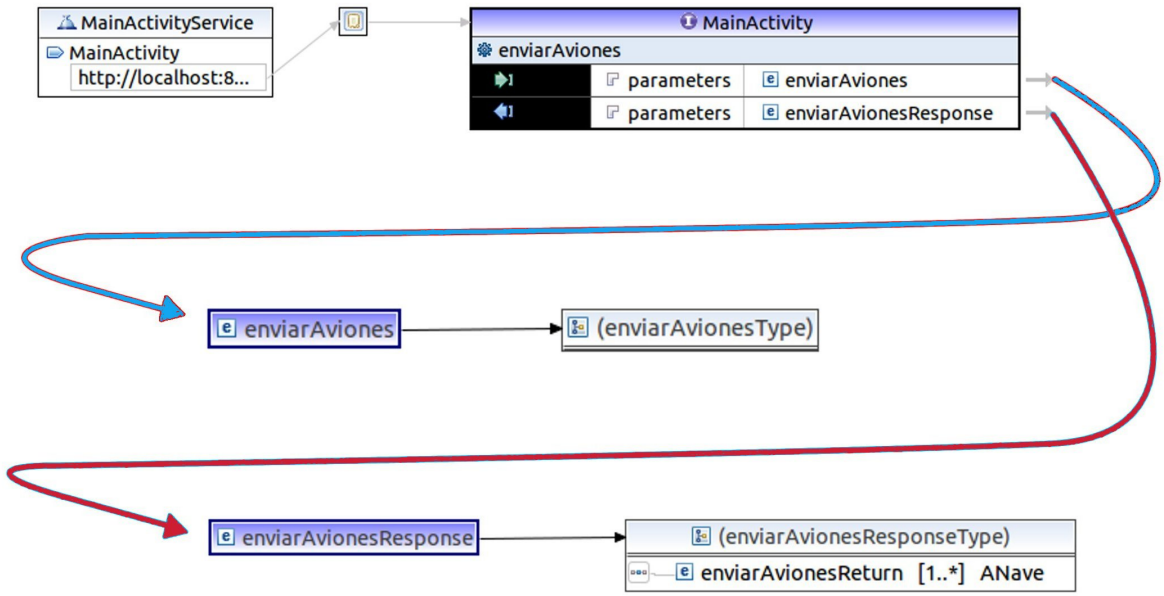

Fig. 4.2: Esquema WSDL del servicio SOAP en Eclipse.

En el esquema puede verse como el servicio se compone por un módulo principal *MainActivity* con dos procedimientos fundamentales *enviarAviones* y *enviarAvionesResponse* que gestionan los parámetros de entrada y los de salida respectivamente.

4.1.2.Esp. funcionales de la aplicación móvil

La función principal de la aplicación móvil, tal y como se indica en anteriores apartados donde se tratan los objetivos, es la de interpretar el rol de cliente y mostrar sobre un mapa la posición de cada uno de los vuelos que son detectados por la antena de la UPV.

La aplicación no está concebida para ser publicada en ningún sitio web ajeno a la UPV, ya que para poder obtener los datos del servidor/antena de la UPV tienen que cumplirse alguno de los siguientes requisitos:

- Conectarse al servidor/antena desde la red local de la UPV, o bien hacerlo desde afuera mediante el acceso VPN.
- Obtener los datos del servidor web SOAP estando éste conectado al servidor/antena desde la red local de la UPV, o bien conectado desde afuera mediante el acceso VPN.

En resumen, para cumplir con los requisitos anteriores es necesaria la conexión a la red de la UPV. Y esto no es posible si no se dispone de los credenciales de alguna cuenta de alumno o de personal de la UPV. Algo que no está al alcance del público en general de manera permanente. Por este motivo, se opta por un enfoque de testeo, ofreciendo la posibilidad de configurarse para recibir los datos del tráfico aéreo de manera más técnica. La aplicación nos permite introducir parámetros como la IP, puerto, etc. para la su configuración.

La aplicación está compuesta de varias vistas o ventanas que se detallan a continuación para poder comprender de una manera más precisa la funcionalidad.

4.1.2.1.Vista del mapa de vuelos

Desde el primer instante en el que se carga la aplicación aparece la vista del mapa de vuelos. A primera vista se observa un mapa en modo de satélite que está enfocado en la Comunidad Valenciana y parte de los alrededores. También se dispone de un botón semi-transparente con el texto “Actualizar” que sirve para hacer una petición al servidor de los vuelos que se detectan en el instante en que el botón es presionado.

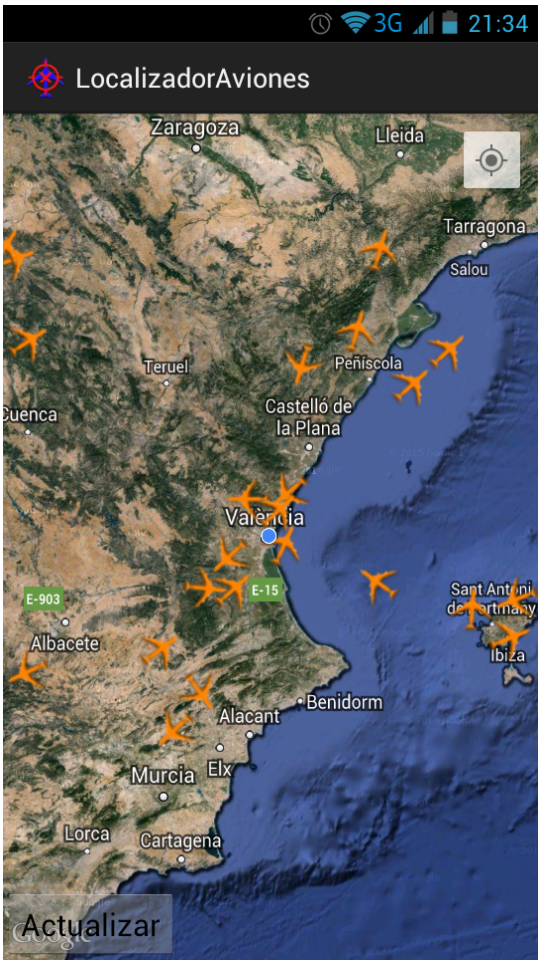

Fig. 4.3: Representación vuelos y Fig. 4.4: Datos del vuelo.

La figura de la izquierda es el resultado de una petición (haber pulsado el botón Actualizar) al servidor con resultado satisfactorio, el mapa permite hacer zoom, cambiar de área, etc.

Cada uno de los aviones que se aprecian se corresponde con lo que en realidad es un avión que sobrevuela el punto exacto en el que está situado en el mapa.

Si pulsamos sobre el icono de cualquiera de los aviones aparece una leyenda con información relevante como el nombre del vuelo, la altura a la que vuela y la velocidad con la que se desplaza. Esto puede verse en la figura de la derecha, donde el vuelo en cuestión es el RYR53J que vuela a la altura de 9.449 metros y se desplaza a una velocidad de 630 km/hora.

4.1.2.2.Vistas de menú y preferencias

Si durante la ejecución de la aplicación pulsamos sobre el icono de menú del terminal se abrirá un menú con dos opciones. En la instantánea de la izquierda se muestra el menú que se muestra al presionar sobre el icono anteriormente citado. Se

trata de un menú de dos ítems, *Configuración* y *Acerca de ...*, el segundo ítem muestra una vista con la información de desarrollo de la aplicación, mientras que el primero nos lleva a una vista de menú con cuatro ítems que nos permitirá configurar la conexión para poder intercambiar datos con el servidor. Cabría destacar que la instantánea también nos ofrece una información implícita, como es la del alcance de la antena de la UPV. Como puede apreciarse, se detectan aviones hasta a una distancia de unos 500-600 km. Aproximadamente.

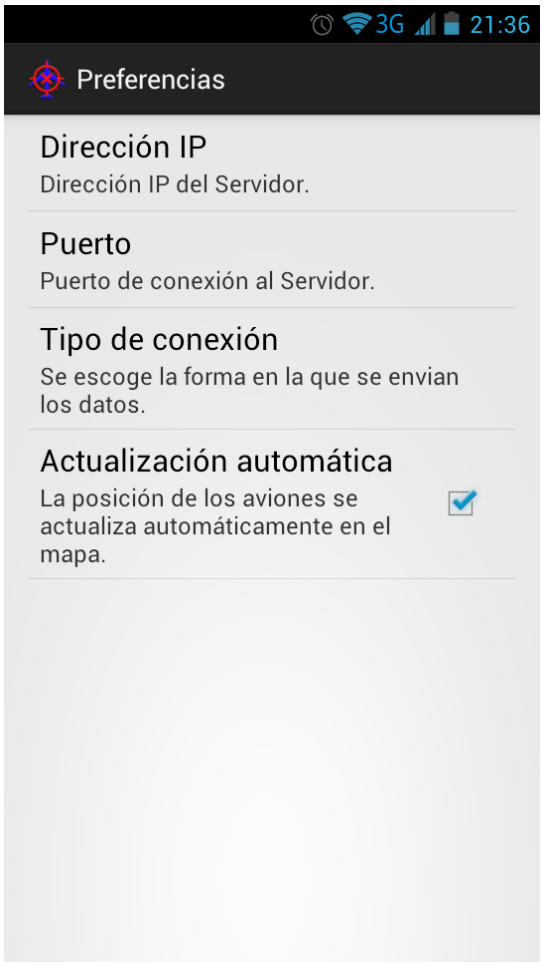

Fig. 4.5: Menú de navegación y Fig. 4.6: Vista de preferencias.

En la figura de la derecha se muestra el menú citado anteriormente, donde los tres primeros ítems sirven para configurar parámetros de la conexión con el servidor como son la dirección IP del servidor, el puerto en el que escucha y el tipo de conexión con el que se quiere intercambiar los datos. Mientras que la última opción del menú sirve para activar/desactivar la actualización automática de la posición de los aviones cada cinco segundos aproximadamente.

4.1.2.3.Vistas de configuración de la conexión

Los tres primeros ítems del menú nos llevan a las siguientes vistas:

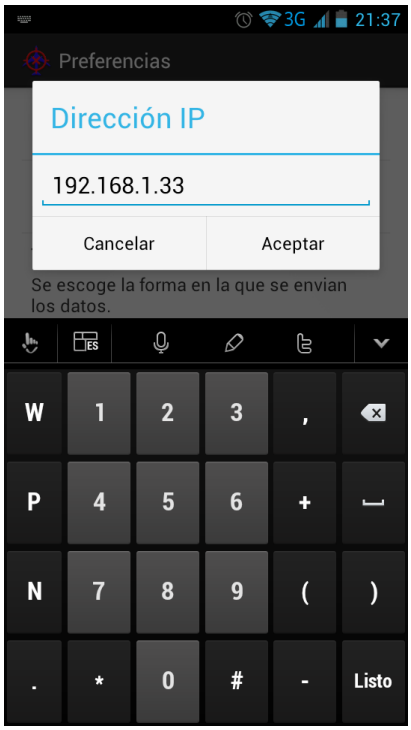

Fig. 4.7: Configurar IP y Fig. 4.8: Configurar Puerto.

Fig. 4.9: Configurar conexión.

- Las dos instantáneas de la parte superior muestran las vistas con las que se introducen los parámetros de la dirección IP y del puerto del servidor con el que queremos conectar (ítems primero y segundo). Se trata de vistas compuestas por un cuadro de texto y un teclado numérico.
- El tercer ítem no lleva a la vista de la parte inferior que es una lista de posibles maneras de poder conectar, de las que solamente elegiremos una.

4.2.Diseño del proyecto

Una vez ya descritas las especificaciones funcionales, en esta sección vamos a describir de una manera abstracta cada uno de los módulos que forman los componentes de este proyecto.

Para tener una visión general del mecanismo principal del proyecto, vamos ha hacer uso de un diagrama de secuencia para explicar como se interacciona durante el funcionamiento de la aplicación.

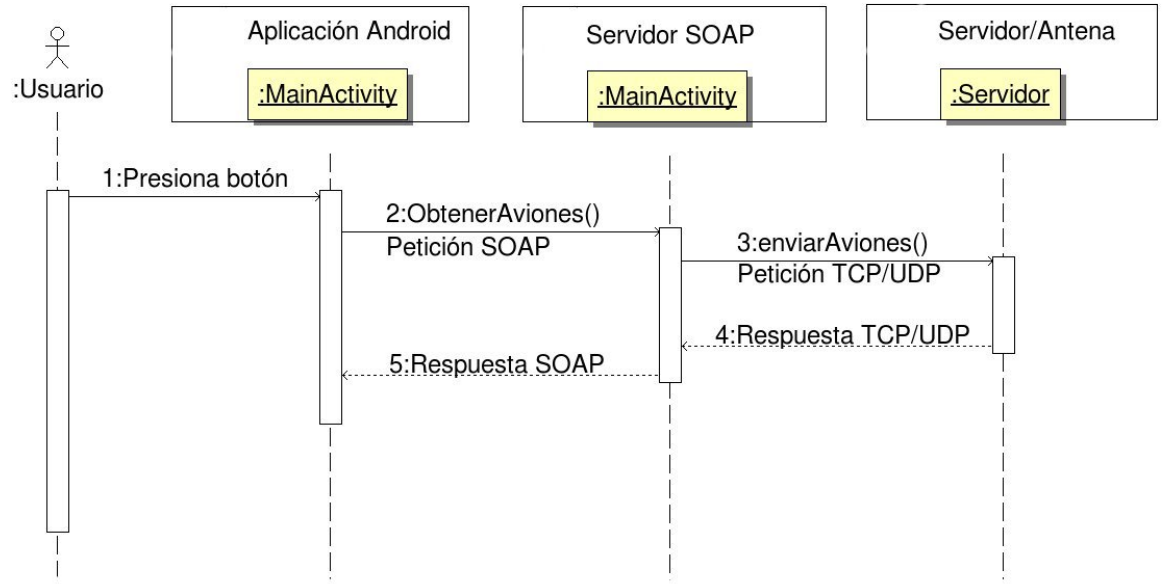

Fig. 4.10: Diagrama de secuencia del proyecto.

En el diagrama de secuencia se describe el proceso que se genera cuando el usuario presiona el botón *Actualizar* de la vista principal del mapa. Como puede observarse, la aplicación recoge el evento y realiza una petición que se propaga a través del servidor SOAP hasta que alcanza al servidor/antena, que responde de nuevo a través del servidor SOAP hasta que se reciben los datos en la aplicación móvil y se muestran en pantalla.

En el proyecto también podría darse el caso en el que la aplicación se comunicara directamente (enviando peticiones TCP/UDP) con el servidor/antena. Consideramos que el anterior diagrama de secuencia también nos sirve para entender el caso anterior, ya que solamente habría que sustituir al servidor SOAP por la aplicación Android, quedando sólo dos componentes y el usuario.

4.2.1.Diseño del Servicio Web SOAP

Con el objetivo de cumplir con las especificaciones del servicio web SOAP, se construyen un conjunto de clases que a continuación se muestran representadas en un diagrama de clases para una comprensión más sencilla de sus funciones y de la relación que tienen entre ellas.

El diagrama es el siguiente:

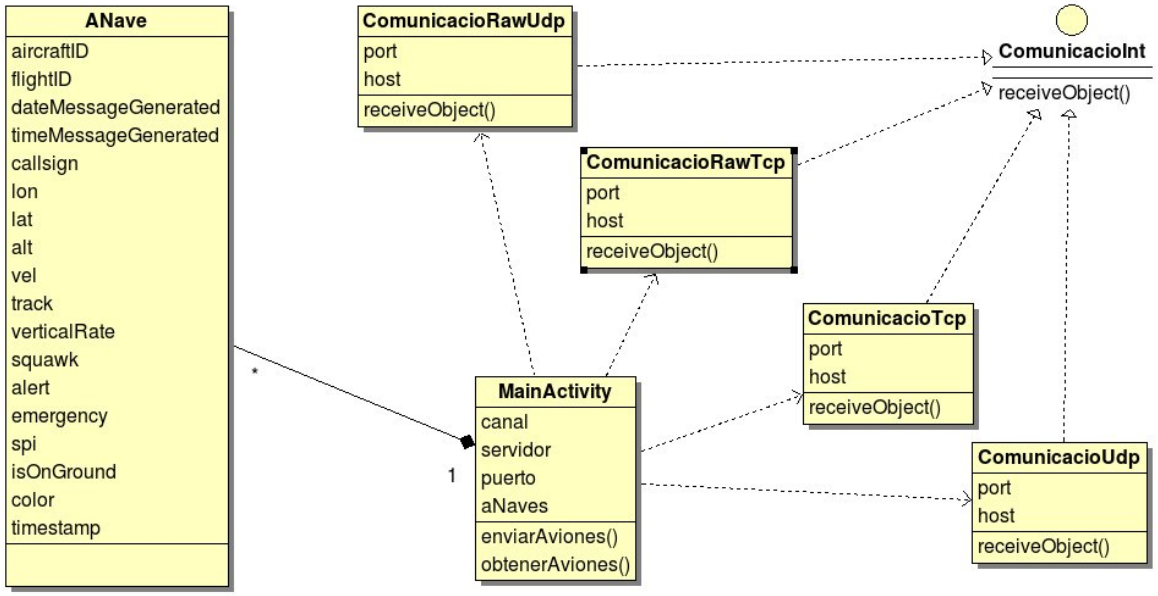

Fig. 4.11: Diagrama de clases del servidor web SOAP.

La posición de cada una de las clases en el diagrama está determinada por la función que despeña. Diferenciamos tres partes o módulos de funcionalidad. Por un lado tenemos los datos propiamente dichos, que están situados al margen izquierdo del diagrama (la clase *ANave*), en el centro situamos la clase que contiene los datos y la lógica del programa (la clase *MainActivity*) y en el extremo derecho tenemos un conjunto de clases cuyo rol principal es el las comunicaciones con el servidor/antena de la UPV (la interfaz *ComunicacioInt* y las demás clases que la implementan). A continuación se detalla cada uno de estos módulos.

- **Los datos.** Cada una de las instancias de la clase *Anave* representa a cada uno de los aviones detectados por la antena de la UPV en tiempo real. La clase únicamente alberga un conjunto de atributos de tipo simple como enteros y cadenas de caracteres para definir características como la posición, el nombre del vuelo, la altura, la velocidad, etc.

- **La lógica y almacén.** La clase *MainActivity* Como su propio nombre indica es la clase principal, esta clase contiene la lógica del programa que está desarrollada con un par de métodos principales. Contiene cuatro atributos:

    - La tabla *ANaves*. Que contiene el conjunto de instancias de la clase *ANave* (una por cada vuelo detectado) que se reciben del servidor/antena de la UPV.

    - *servidor* y *puerto*. Que son una cadena de caracteres un entero respectivamente para definir los parámetros de la conexión.

    - El objeto *canal*. Mediante el cual se obtienen los datos.

Es bastante simple ya que su función no es más que la de recibir/enviar peticiones y la de recibir/enviar datos (vuelos). Cuando la clase está instanciada y se recibe una petición invocando su método *enviarAviones()*, se ejecuta el código que llama al otro método *obtenerAviones()* (privado) pasándole los atributos de la clase como parámetros para generar la conexión con el servidor/antena.

- **El módulo de las comunicaciones.** La interfaz *ComunicacioInt* obliga a que las clases que la implementan dispongan del método *receiveObject()* que devuelve un objeto de tipo array de bytes que contiene el conjunto de vuelos detectados.

Las clases que implementan la interfaz *ComunicacioInt* (*ComunicacioTcp*, *ComunicacioUdp*, *ComunicacioRawTcp* y *ComuncacioRawUdp*), son cada uno de los diferentes tipos de conexión que pueden (o que han podido a lo largo de los últimos años) establecerse con el servidor/antena para obtener los datos.

4.2.2.Diseño de la aplicación móvil

De igual modo que con el servidor web SOAP, hemos considerado oportuno explicar el diseño de las clases de la aplicación móvil con la ayuda de un diagrama de clases.

La siguiente figura nos muestra el diagrama:

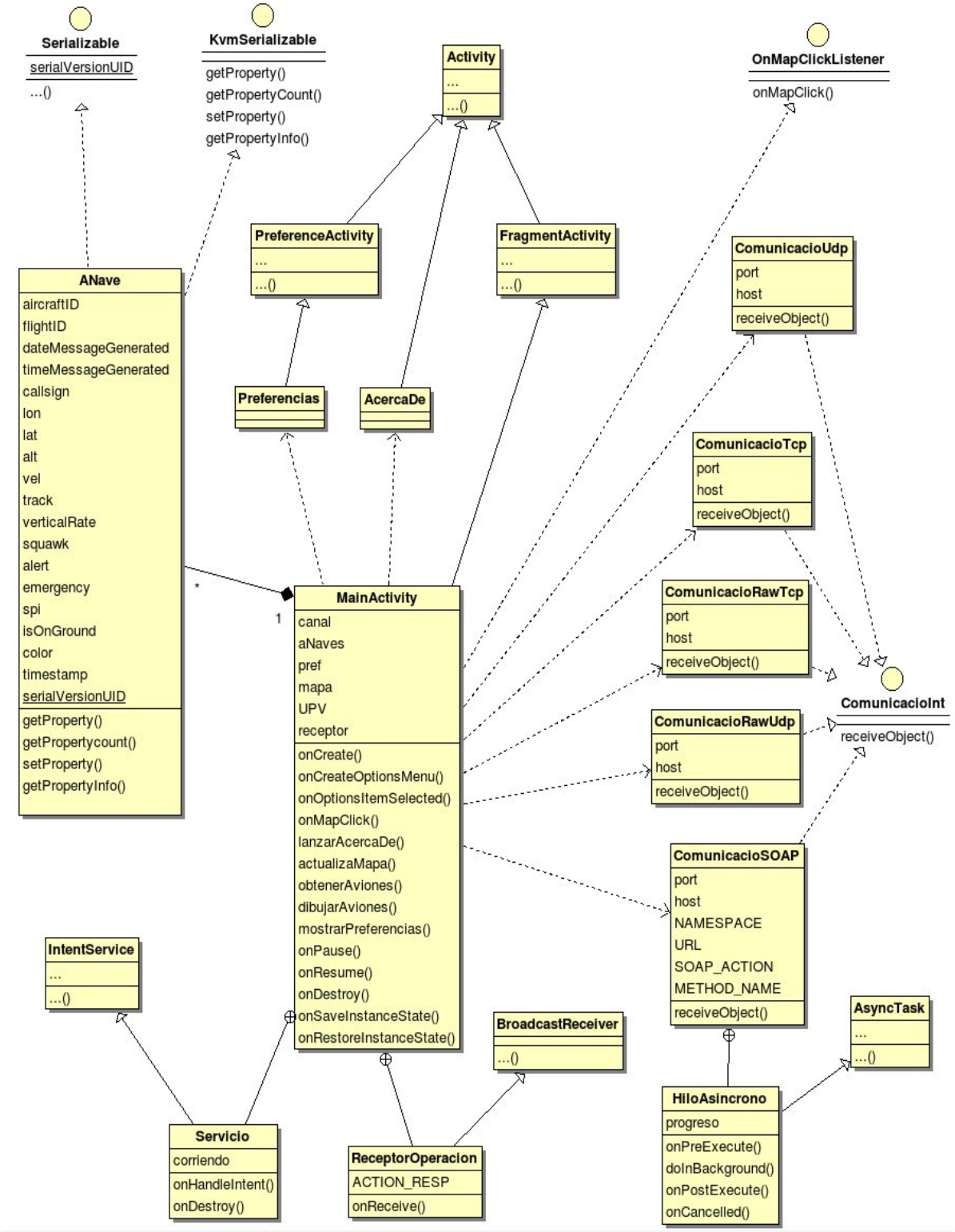

Fig. 4.12: Diagrama de clases de la aplicación móvil.

Al igual que en el apartado anterior con el diagrama de clases del servidor web SOAP, en este diagrama de clases de la aplicación móvil, la posición de cada una de las clases en el diagrama está determinada por la función que despeña. En este caso diferenciamos cinco partes o módulos de funcionalidad. Por un lado tenemos los datos, que están situados al margen izquierdo del diagrama (la clase *ANave*), en el centro situamos la clase que contiene los datos y la lógica del programa (la clase *MainActivity*), en la parte superior ubicamos al módulo de las vistas, en el extremo derecho tenemos un conjunto de clases cuyo rol principal es el las comunicaciones con el servidor/antena de la UPV y el servidor web SOAP (la interfaz *ComunicacioInt* y las demás clases que la implementan), y en la parte inferior un módulo de clases anidadas cuyas instancias son procesos auxiliares para realizar operaciones (como la obtención de datos por red, un temporizador, etc.) que no se pueden llevar a cabo en el proceso principal de la aplicación debido a que son bloqueantes.

A continuación se detalla cada uno de estos módulos:

- **Los datos.** Cada una de las instancias de la clase *Anave* representa a cada uno de los aviones detectados por la antena de la UPV en tiempo real.

La clase alberga un conjunto de atributos de tipo simple como enteros y cadenas de caracteres para definir características como la posición, el nombre del vuelo, la altura, la velocidad, etc. Pero en este caso se implementan las interfaces *Serializable*, que permite que el objeto sea serializado para poder ser guardado dentro de un contexto de la aplicación principal, y *KvmSerializable* que le permite al objeto ser enviado y recibido mediante el uso de la librería KSoap2-Android (en comunicaciones tipo SOAP).

- **La lógica y almacén.** La clase *MainActivity*, al igual que en el servidor web SOAP, es la clase principal.

Esta clase contiene la lógica del programa que está desarrollada con un par de métodos principales. Contiene los siguientes atributos a destacar:

- La tabla *ANaves*. Que contiene el conjunto de instancias de la clase *ANave* (una por cada vuelo detectado) que se reciben del servidor/antena de la UPV.
- Las preferencias *pref*. Que es una cadena de caracteres con las opciones introducidas para definir los parámetros de la conexión.

- El objeto *canal*. Mediante el cual se obtienen los datos en las comunicaciones.

- El mapa. Es la clase de la vista (por así decirlo) que se obtiene de la clase heredada *FragmentActivity*. Es el mapa principal de nuestra aplicación móvil.

- La ubicación *UPV*. Son las coordenadas de la posición exacta de la UPV que es donde está situada el radar.

- El objeto *receptor*. Que es una clase que hereda de *BroadcastReceiver* y cuya instancia tiene la función de recibir peticiones desde otros procesos que se ejecutan fuera del proceso principal.

A diferencia de la clase *MainActivity* del servidor web SOAP, ésta es más compleja ya que su función no es solamente la de recibir/enviar peticiones y la de recibir/enviar datos (vuelos). En este caso se definen también funciones propias de una aplicación Android, como las de escucha de eventos, las de gestión de los procesos y vistas, etc. que serán explicados con más detalle en próximo capítulo.

- **El módulo de las comunicaciones.** La interfaz *ComunicacioInt* (como hemos indicado en apartado anterior) obliga a que las clases que la implementan dispongan del método *receiveObject()* que devuelve un objeto de tipo array de bytes que contiene el conjunto de vuelos detectados.

Las clases que implementan la interfaz *ComunicacioInt* (*ComunicacioTcp*, *ComunicacioUdp*, *ComunicacioRawTcp* y *ComuncacioRawUdp*), son cada uno de los diferentes tipos de conexión que pueden (o que han podido a lo largo de los últimos años) establecerse con el servidor/antena para obtener los datos. Cabe destacar que en este caso también contamos con la clase *ComunicacioSOAP* que nos va a servir para comunicarnos con el servidor web SOAP con la misma metodología que con las anteriores.

- **El módulo de las vistas.** A parte de la vista principal que es el mapa, existen más vistas que componen a la aplicación como la vista *AcercaDe* o la vista *Preferencias* que es la vista que se muestra con las opciones de configuración (todas ellas heredan de *Activity*).

- **Clases auxiliares de la aplicación.** Como hemos indicado anteriormente, en la parte inferior de nuestro diagrama de clases ubicamos un tipo de clases anidadas cuyas instancias son procesos auxiliares que nos sirven, entre otras cosas, para realizar operaciones que pueden bloquear al proceso que las ejecuta durante un tiempo.

El sistema por defecto no crea un hilo independiente cada vez que se crea un nuevo componente. Es decir, todas las actividades y servicios de una aplicación son ejecutados por el hilo principal. Si le asignamos a este hilo el desempeño de una actividad que tarde bastante, desde el punto de vista del usuario se tendrá la impresión de que la aplicación se ha colgado. Mas todavía, si el hilo principal es bloqueado más de 5 segundos, el sistema mostrará un cuadro de dialogo al usuario "La aplicación no responde" para que el usuario decida si quieres esperar o detener la aplicación. Las clases que llevan a cabo tales operaciones bloqueantes son:

- La clase *HiloAsincrono*. Esta clase se encuentra anidada dentro de cada una de las clases que implementan la interfaz de comunicaciones y su función es la de permanecer en ejecución durante el tiempo que tarde en completarse cada operación de recepción/envío de datos, hereda de *AsyncTask*.

- La clase *Servicio*. Es la clase que permite que la pantalla se actualice cada cierto intervalo de tiempo, hereda de *IntentService*. El usuario decide su tiempo de ejecución mediante el menú de configuración.

Otra clase que realiza un tarea auxiliar es *ReceptorOperacion*, pero a diferencia de las dos anteriores, su función es la de recibir peticiones desde otros procesos que se ejecutan fuera del proceso principal, funciona de alguna manera como un disparador ante eventos externos al proceso principal, la clase hereda de *BroadcastReceiver*.

4.3.Tecnologías empleadas:

Para describir con más detalle el proyecto, haremos un breve resumen esquemático (dado que éstas son explicadas con más detalle en el capítulo 3) de las tecnologías empleadas:

- **Entorno** de desarrollo:
  - Eclipse 4.3 Kepler.
  - Plugins de Eclipse utilizados:
    - ADT Android (entre otros).
- **Lenguajes** de programación utilizados:
  - Java, Version 6 de JDK.
  - XML (en la implementación de mensajes SOAP).
- **Servidor** web con soporte para SOAP:
  - Apache Tomcat 7.
  - Complementos del servidor:
    - Apache Axis2.
  - Plataformas soportadas por la aplicación:
    - Versión SDK mínima: 8, Android 2.2 Froyo.
  - Versión SDK objetivo: 18, Android 4.3 Jelly Bean.
- **Librerías** complementarias utilizadas:
  - KSoap2-Android (para consumir servicios SOAP mediante sistemas Android).
  - GoogleMaps v2 (para implementar el mapa de la aplicación).

5. Implementación

Una vez ya se conocen las especificaciones y el diseño del proyecto, en este capítulo lo que se detalla es el cómo se traduce a código lo descrito anteriormente, haciendo especial hincapié en las partes de mayor relevancia o que han ofrecido una mayor dificultad para ser implementadas. Así como también de la metodología (o pasos a seguir) puesta en práctica para el alcance de alguno de los objetivos.

5.1. Puesta a punto

Para poder llevar a cabo la realización del proyecto se necesita disponer de un entorno de desarrollo para poder crear las aplicaciones necesarias. En nuestro caso ya se disponía con anterioridad de una versión actualizada del Eclipse.

En el caso de la parte del proyecto del servidor web SOAP es necesario tener instalado un servidor con el que lanzar peticiones y probar nuestro código. En nuestro caso se elige el servidor Apache Tomcat junto con el motor de servicios web Apache Axis2 para poder ofrecer servicios web SOAP mediante el servidor. Una vez descargados ambos, se configura el Eclipse para que sea capaz de encontrar sus ejecutables y poder de ese modo crearse perfiles de ejecución de nuestro código con el usos de estos nuevos componentes.

Por otro lado, en el caso de la parte del proyecto de la aplicación móvil será necesaria la instalación del plugin de Eclipse ADT (Android Development Tools) para poder desarrollar el código de la aplicación siguiendo los patrones que ha de seguir toda aplicación nativa Android. Además, es recomendable disponer de un terminal con la Plataforma Android para poder testear el código, ya que la máquina virtual (simulador) que viene por defecto en la herramienta no permite testear correctamente (o bien se hace de un modo bastante precario) algunas funcionalidades de los terminales como el GPS, acelerómetro, etc. y además es bastante lenta y problemática (se cuelga con bastante facilidad). En nuestro caso disponemos de un terminal propio de la marca ZTE modelo "Grand(xm)" con la versión 4.0 "Ice Cream Sandwich" del sistema operativo Android para hacer pruebas.

5.2.Implementación del Servidor web SOAP

Como hemos indicado anteriormente en el apartado del diseño de las clases, diferenciamos tres partes o módulos de funcionalidad. Por un lado tenemos los datos que se intercambian, que están codificados por la clase *ANave*, por otro, la clase que contiene los datos y la lógica del programa es la clase *MainActivity*, y además tenemos un conjunto de clases cuyo rol principal es el de realizar las comunicaciones con el servidor/antena de la UPV que son la interfaz *ComunicacioInt* y las demás clases que la implementan. A continuación se muestra cómo están implementadas y también cómo se ha generado el servicio.

5.2.1.La clase ANave

El contenido de la clase *ANave* es el siguiente:

```
package localizadoraviones;
public class ANave {
        public String AircraftID = null;
        public String HexIdent = null;
        public String FlightID = null;
        public String Callsign = null;
        public String lon = null;
        public String lat = null;
        public String alt = null;
        public String vel = null;
        public String Track = null;
        ...
        public ANave(){
          } // ctor
        public ANave (String _AircraftID, String _HexIdent, String _FlightID,...)
          {
            AircraftID = _AircraftID;
            HexIdent = _HexIdent;
            FlightID = _FlightID;
            ...
          } // ctor
}
```

Es la clase que se utiliza para representar a los vuelos, que son los datos en nuestro proyecto. Cada una de las instancias de esta clase se corresponde con uno de las aeronaves que son detectadas por el servidor/antena de la UPV. La clase se compone de diecinueve atributos y de dos métodos constructores. En la página anterior se muestran algunos de los atributos que son relevantes para nuestro proyecto, el resto de los atributos de la clase no se utilizan pero se mantienen por si en un futuro se decide ampliar las especificaciones funcionales de la aplicación. Los atributos a destacar son:

- La cadena *Callsign*. Es el nombre del vuelo.
- La cadena *lon*. Longitud del vuelo.
- La cadena *lat*. Latitud del vuelo. El anterior atributo y éste componen las coordenadas geográficas de la posición del vuelo.

- La cadena *alt*. Es la altura a la que vuela la aeronave.
- La cadena *vel*. La velocidad a la que se desplaza.
- La cadena *Track*. Es el ángulo que forman el vector desplaza-miento de la aeronave con respecto a Norte. Este dato se utiliza para representar los gráficos de las aeronaves con una inclinación acorde con su desplazamiento.

5.2.2.La clase MainActivity

Es la clase principal y se compone de los siguientes atributos:

```
private ComunicacioInt canal = null;
private String servidor;
private int puerto;
private static java.util.Hashtable<String, ANave> aNaves = new java.util.Hashtable<String, ANave>();
```

- *ComunicacioInt canal*. Se utiliza para establecer comunicaciones en red y necesita de los atributos *servidor* (cadena) y *puerto* (entero) para su construcción. Mediante una instancia de este objeto y a través de su método *receiveObject()* se obtienen los datos objetos ANave del servidor sin la necesidad de programar a nivel de Socket.
- *TablaHash aNaves*. Es el objeto que almacena las instancias de cada vuelo hasta que son enviadas al cliente.

Los métodos de la clase principal son los siguientes:

```
public ANave[] enviarAviones(){

	ANave[] r = null;
		ANave auxA;
		if(obtenerAviones(servidor, puerto)) {
			Set<String> llaves = MainActivity.aNaves.keySet();
			int i = 0;
			if (llaves.size() > 0){
				r = new ANave[llaves.size()];
				for(String llave: llaves){
					auxA = aNaves.get(llave);
					r[i] = auxA;
					i++;
					}
				}
			}else
				{
				System.out.println("No se han recibido los aviones correctamente.");
				}
		return r;
	}
```

La función *enviarAviones()* es el primero que se ejecuta al recibirse una petición SOAP y devuelve un array de instancias *ANave*. Primero se crean los objetos auxiliares, se llama al método que obtiene los datos del servidor/antena (en la primera sentencia *if*) y si el resultado es favorable se extraen los datos de la tabla y se preparan en el array para ser devueltos, en caso contrario se muestra un mensaje de error.

```
private boolean obtenerAviones(String servidor, int puerto){
  canal = new ComunicacioRawTcp(servidor, puerto);
  byte[] rebut_byte = null;
  if(canal != null){
    Object aux = canal.receiveObject();
    if(aux instanceof byte[]){
         rebut_byte = (byte[])aux;
         if(rebut_byte != null){
                  aNaves.clear();
                  String linia;
                  java.io.ByteArrayInputStream bais = new java.io.ByteArrayInputStream(rebut_byte);
                  java.io.BufferedReader br = new java.io.BufferedReader(new java.io.InputStreamReader(bais));
                  boolean terminado = false;
                  int i = 0;
                  while(!terminado){
                           ...
                           //Tratamiento de datos en el bucle
                           ...
                  }//while terminado
            }

         }else{
            System.out.println("Error de entrada/salida: No hemos recibido un byte[]. ");
            return false;
     }//si el objeto no es de tipo byte[]
    }else{
         System.out.println("Error de entrada/salida: canal es null. ");
         return false;
  }//si canal es diferente de null...
  return true;
}
```

La otra función, *obtenerAviones()* devuelve verdadero si la operación se ha completado satisfactoriamente y falso en caso contrario. En primer lugar se crea un objeto de la clase de comunicaciones con los parámetros de direccionamiento de red, sin todo va bien se lanza la función *receiveObject()* que almacenamos en la variable *aux* (de tipo objeto) y se comprueba que sea de tipo array de bytes. Si es de ese tipo se procede a inicializar las variables para operar con los bytes y se realiza su tratamiento dentro del bucle que itera hasta que éstos se terminan.

5.2.3.Las clases de intercambio de datos

Las clases de este tipo implementan la siguiente interfaz:

```
package localizadoraviones;
public interface ComunicacioInt{
         public Object receiveObject();

}
```

Existen cuatro clases que implementan la interfaz y son cada uno de los diferentes tipos de conexión que pueden (o que han podido a lo largo de los últimos años) establecerse con el servidor/antena para obtener los datos:

- Para crear conexiones de tipo TCP se dispone de las clases *ComunicacioTcp* y *ComunicacioRawTcp*.
- *ComunicacioUdp* y *ComunicacioRawUdp* se utilizan, como era de esperar, para construir conexiones de tipo UDP.

En los inicios del proyecto únicamente se disponía de las clases *ComunicacioTcp* y *ComunicacioUdp*, pero debido a una actualización del servidor y por motivos de compatibilidad se crearon también el resto. Se ha decidido no descartar las clases antiguas por si se diera el caso de algún nuevo cambio que revierta la configuración. A continuación se muestra el código de una de estas clases:

```
package localizadoraviones;
public class ComunicacioRawTcp implements ComunicacioInt{
        int port;
        String host = null;

        public ComunicacioRawTcp(String host, int port){
                super();
                this.port = port;
                this.host = host;
        }
        public Object receiveObject(){
          java.lang.String retString = new java.lang.String();
          java.lang.String aux = null;
          Object ret  = null;
          java.net.Socket so = null;
          if(this.host != null){//Estem ben configurats
            try {
             java.net.InetAddress ia = java.net.InetAddress.getByName(host);
             try{
               so = new java.net.Socket(ia, this.port);
               try{
                  java.io.InputStream is = so.getInputStream();
                  java.io.BufferedReader br = new java.io.BufferedReader(new java.io.InputStreamReader(is));
                  aux = br.readLine();
                  while(aux != null){
                   retString = retString.concat(aux + "\n");
                   aux = br.readLine();
                  }
                  is.close();
                }catch(java.io.IOException e){
                  System.err.println("Error d'entrada/eixida: " + e.getLocalizedMessage());
                  }
                so.close();
               } catch(java.io.IOException e){
                 System.err.println("Error en la creació del socket: " + e.getLocalizedMessage());
                 }
            } catch (java.net.UnknownHostException e) {
                 System.err.println("Nom del host desconegut: " + e.getLocalizedMessage());
                 }
         }
         ret = retString.getBytes();
         return(ret);
       }

}
```

Como puede observarse, son clases que operan con objetos a nivel de red como los de tipo *Socket*, etc.

5.2.4.Generación del servicio

Para la creación del servicio web SOAP se ha elegido la metodología *botom up* que consiste en generar el servicio partiendo del código ya que es una manera bastante simple y no requiere de unos conocimientos muy avanzados para la creación de esta clase de servicios. Los pasos seguidos han sido los siguientes:

- Creación de un método *main* auxiliar. En primer lugar, para probar que mediante el uso de las clases de comunicaciones, creamos un método main que al ejecutarse crea una instancia de un tipo de comunicaciones y se llama a su función para que se reciban los datos. Se comprueba que se reciben los datos mostrando por pantalla lo que se recibe.

- Creación del servicio. Con la ayuda de las herramientas que ofrece Eclipse creamos el servicio seleccionando la clase y también el método que se publica en el servicio. Este paso se explicará más adelante con más detalle debido a su relevancia.

- Despliegue en el servidor. Partiendo de la base que se tiene instalado y configurado el servidor (Apache Tomcat) para ejecutarse a través del Eclipse. Se añade nuestro proyecto a la lista de proyectos publicados por el servidor.

- Testeo del servicio. Haciendo uso la herramienta de Eclipse *Web Services Explorer,* se prueba nuestro servicio hasta que funcione correctamente.

A continuación se indica paso a paso cómo se ha creado nuestro servicio web SOAP en Eclipse.

5.2.4.1.Creación del servicio en Eclipse

Para crear un servicio web SOAP en Eclipse a partir del código, debemos seleccionar (desplegando nuestro proyecto en el *Package Explorer* o similar) la clase que contiene la función o las funciones que queremos que se publiquen como servicio web. A continuación con el puntero del ratón sobre la clase abrimos el menú contextual presionando el botón derecho y elegimos las siguientes opciones: *Web Services → Create Web Service.*

La siguiente figura nos muestra el formulario que se muestra:

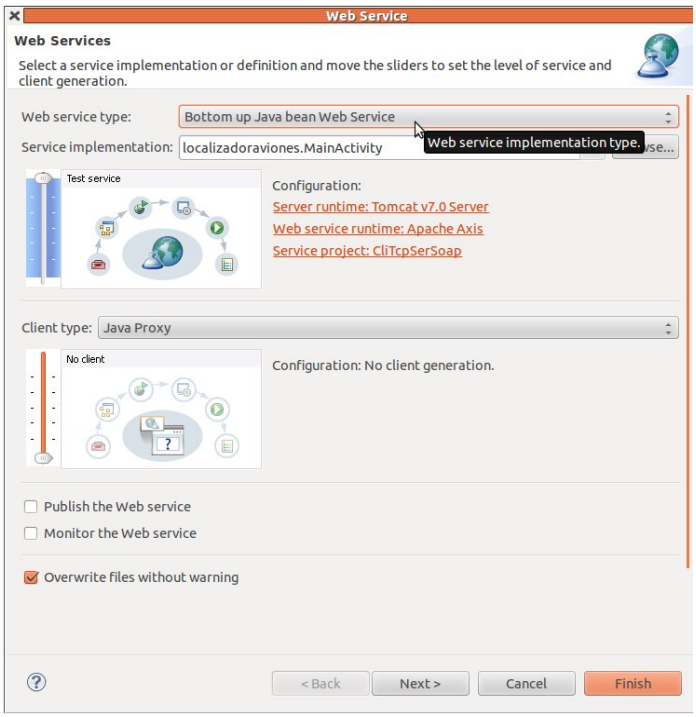

Fig. 5.1: Formulario de creación de servicios web.

En este formulario se configura el tipo (*Bottom up*), la clase de nuestro proyecto y el nivel hasta el que se desea llegar en su creación (*Test service*, hasta el nivel de testeo). Pulsamos en *Next* y en el siguiente formulario editamos el nombre del archivo de publicación de servicios y se seleccionan las funciones de la clase a publicar. Ya podremos pulsar en *Finish*.

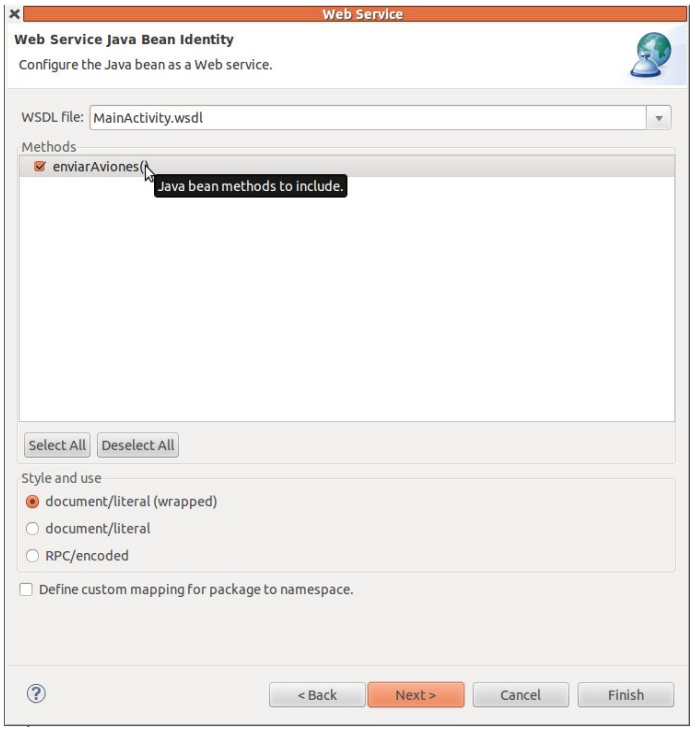

Fig. 5.2: Formulario de creación de servicios web.

Una vez se han completado los pasos anteriores, si todo ha ido correctamente, se genera un conjunto de ficheros distribuidos en distintas carpetas de nuestro proyecto de los que destacamos un fichero XML con extensión (*MainActivity.wsdl* en nuestro proyecto) *.wsdl* que contiene las características de nuestro servicio y que es el fichero que se publica para que sea descargado por los clientes que deseen consumir nuestro servicio.

5.3.Implementación de la aplicación Android

Refiriéndonos en este caso a los aspectos de la implementación de la aplicación móvil, cabría el destacar que se ha tratado de seguir el mismo patrón en la nomenclatura de las clases con respecto a sus roles, al igual que con el servicio web. En este caso también tenemos una clase para datos con el mismo nombre, ídem para las clases de comunicaciones, etc. En los siguientes apartados se indican los cambios realizados en cada clase para poder realizar sus funciones en el nuevo contexto.

5.3.1.La clase ANave

La clase de datos se llama de la misma manera y conserva los mismos atributos que la clase del servicio web. Pero existen diferencias notables, en este caso se implementan las interfaces *Serializable* y *KvmSerializable*.

```
…
//Atributo para implementar Serializable
private static final long serialVersionUID = 77777L;

//Funciones para implementar KvmSerializable:
public Object getProperty(int arg0) {…}
public int getPropertyCount() {…}
public void getPropertyInfo(int arg0, Hashtable arg1, PropertyInfo arg2) {…}
public void setProperty(int arg0, Object arg1) {…}
…
```

La aplicación móvil contiene más de una vista (*Activity*). Con el salto entre diferentes vistas (cambio de la pantalla que se muestra en primer plano) se pierden los datos en contexto. Para que esto no suceda, es necesario guardar los datos antes del cambio en un objeto para posteriormente recuperar los mismos datos de este objeto desde la otra vista. En problema es que este objeto almacén solamente acepta datos serializados. Se implementa la clase *Serializable* con el objetivo de poder guardar nuestra clase de datos serializada en ese objeto. Y para esto es necesario añadir el atributo indicado anteriormente.

El motivo por el cual nuestra clase de datos también implementa la interfaz *KvmSerializable* es el de agregarle la característica de poder ser enviado o recibido mediante la librería SOAP para Android *Ksoap2-Android.* Para poder implementar esta interfaz, nuestra clase tiene que disponer de las funciones indicadas anteriormente.

5.3.2.La clase MainActivity

Al igual que en el servicio web, esta es la clase más importante de nuestra aplicación ya que contiene los métodos y un conjunto de clases anidadas que definen el comportamiento de nuestro programa. En este caso las funciones que se realizan van más allá que las de enviar y recibir datos, se representan en un mapa y además se gestionan los procedimientos que se llevan a cabo durante el cambio de vistas, los datos que se reciben por parte del usuario, los eventos que se producen, etc.

Vamos a continuación a proceder a la explicación del código de nuestra clase. Debido a la extensión de nuestra clase (tiene casi trescientas cincuenta líneas de código), consideramos que no es viable la exposición de la totalidad del código en este apartado. En su lugar se indicarán solamente las cabeceras de los métodos y funciones, aunque en algún caso en particular sí se muestren completos. Para poder ver el código completo de los mismos se remite al lector al capítulo de anexos donde se exponen algunas de las clases más importantes y representativas.

Nuestra clase empieza con las siguientes líneas:

```
public class MainActivity extends FragmentActivity implements OnMapClickListener {

        private ComunicacioInt canal = null;
        private SharedPreferences pref;
        private static GoogleMap mapa;
        private static LatLng UPV = new LatLng(39.481106, -0.340987);
        private java.util.Hashtable<String, ANave> aNaves = new java.util.Hashtable<String, ANave>();
        private ReceptorOperacion receptor;
…
```

La clase hereda de *FragmentActivity* porque es la que muestra la vista principal que es el mapa de Google (GoogleMapsV2 que más adelante será tratado), e implementa la clase *OnMapClickListener* que ofrece la posibilidad de interactuar con el mapa. Los atributos tienen las siguientes funciones:

- El objeto *canal* y la tabla *aNaves* se repiten también en esta clase. Al igual que en el programa del servicio web, el primero sirve para recibir datos mientras que el segundo los almacena.

- El objeto *pref* tiene la utilidad de contener las opciones de menú que han sido previamente seleccionadas por el usuario.

- El atributo *mapa* es la clase *GoogleMaps* que muestra el mapa de la vista principal y que ofrece los métodos de la clase para su configuración (aspectos de zoom, aspecto, ubicación, etc.).

- *UPV* es el punto del mapa con el que se se abre la vista del mapa. Está compuesto de dos objetos, uno para definir la latitud y el otro para definir la longitud del punto que se refiere a la ubicación de la Universidad Politécnica de Valencia.

- El objeto *receptor* ofrece la posibilidad a la aplicación de responder frente a unos eventos determinados. En nuestra aplicación tiene el rol de actualizar el mapa cada vez que se genera el evento desde el proceso temporizador.

Los métodos para la gestión de datos (instancias ANave) en nuestra clase son los siguientes:

```
…
        public void actualizaMapa(View view){ … }
        private void obtenerAviones(){ … }
        private void dibujarAviones(){ … }
…
```

De los tres métodos anteriores, los más importantes son *obtenerAviones()* y *dibujarAviones()*, ya que el primero lo que hace, es en primer lugar hacer un borrado de los elementos del mapa y a continuación la llamada de los otros dos métodos.

- Dentro del método *obtenerAviones()* lo primero que se hace es leer la configuración que se ha obtenido del usuario, a continuación se crea el objeto de comunicaciones para obtener los datos, posteriormente se invoca (se obtienen los datos), y finalmente se realiza un tratamiento de lo recibido en función del tipo de conexión que se ha seleccionado. Si se ha seleccionado un tipo de conexión SOAP se reciben los datos directamente en un array de objetos ANave, mientras que si se ha seleccionado cualquier otro tipo de conexión, los datos se reciben en un array de bytes. Finalmente los datos se almacenan en la tabla de los vuelos (instancias ANave) detectados.

- El método *dibujarAviones()* itera la tabla de los vuelos creando para cada vuelo detectado un marcador en una posición del mapa determinada.

5.3.2.1.Las vistas

En nuestra aplicación disponemos de varias vistas cuyo código es externo a esta clase principal, pero se ha decidido incluir dentro de su apartado porque contiene a los métodos que las gestionan.

En las aplicaciones Android suele ser habitual seguir la filosofía de crear cada vista con el uso de una actividad que carga su aspecto desde un fichero XML dentro de la carpeta de recursos /res. En nuestra aplicación también se sigue el mismo patrón para la implementación de las vistas. Se dispone de varias pero a continuación solamente vamos a ver unos de los casos más representativos, la vista del menú de configuración. Para acceder a la vista del menú de configuración hay que previamente seleccionar el ítem *Configuración* dentro del menú inicial (el menú al que se accede presionando el botón de menú de nuestro terminal). La siguiente función es la que gestiona la elección de los ítems de ese menú:

```
…
@Override public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case R.id.acercaDe:
              lanzarAcercaDe(null);
              break;
        case R.id.config:
             Intent i = new Intent(this, Preferencias.class);
             startActivity(i);
             break;
         }
         return true;
}
…
```

La función recibe como parámetro el ítem de la vista que se desea abrir, en este caso para acceder a la vista de configuración el ítem recibido tendría el valor *R.id.config* que obliga a ejecutarse la parte del *switch* en la que se crea un objeto *Intent* al que se le pasa por parámetro la clase Preferencias (que es nuestra actividad de configuración) y posteriormente la lanza.

La clase Preferencias tiene el siguiente aspecto:

```
package com.example.localizadoraviones;

import android.os.Bundle;
import android.preference.PreferenceActivity;


public class Preferencias extends PreferenceActivity {
   @Override protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      addPreferencesFromResource(R.xml.preferencias);
   }
     }
```

La clase solamente contiene un método que es el que sirve para crear la vista y en método se carga por referencia (*R.xml.preferencias*) el archivo XML que suministra el aspecto de la vista que físicamente esta ubicado en */res/xml/*. Para ver el código del archivo XML consultar el capítulo de anexos.

5.3.2.2.Google MapsV2

La vista más importante de nuestra aplicación es la que se muestra al iniciarse, se trata de la vista del mapa Google MapsV2. Para que funcione es requisito indispensable que el dispositivo donde instalemos nuestra aplicación tenga *Google Play* instalado. A diferencia de Android, Google MapsV2 no es un software libre, por lo que está limitado a una serie de condiciones de servicio. Los pasos a seguir para poder mostrar el mapa en nuestra aplicación son los siguientes:

- Obtener una clave Google Maps.
- Importar la librería *Google Play Services* del SDK de Android a nuestro proyecto.
- Crear el archivo XML de interfaz que albergue el mapa.
- Modificar el archivo *AndroidManifest.xml* para añadirle la clave y los permisos de aplicación necesarios.

Al iniciarse la vista del mapa se ejecutan las siguientes líneas en el método *onCreate()* de la actividad principal:

```
…
setContentView(R.layout.activity_main);
…
mapa = ((SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map)).getMap();
mapa.setMapType(GoogleMap.MAP_TYPE_HYBRID);
mapa.setMyLocationEnabled(true);
mapa.getUiSettings().setZoomControlsEnabled(false);
mapa.getUiSettings().setCompassEnabled(true);
mapa.moveCamera(CameraUpdateFactory.newLatLngZoom(UPV, 7));
mapa.setOnMapClickListener(this);
...
```

Primeramente se indica el soporte sobre el que se va a situar el mapa que es *R.layout.activity_main*, posteriormente se obtiene el mapa, y a continuación las líneas de código siguientes definen el aspecto del mapa, el punto central, el zoom, etc.

Con esto es suficiente para que el mapa se muestre con las características que son necesarias para nuestra aplicación.

5.3.2.3.Servicio y receptor de eventos

Para suministrar la funcionalidad de la actualización automática de la posición de los vuelos detectados sobre el mapa, se hace uso de un par de clases que se implementan anidadas en la clase principal. Se trata de un servicio temporizador que lanza cada cierto intervalo de tiempo un evento, y de un receptor de eventos que cada vez que recibe un evento de este tipo ejecuta los métodos que hacen que se actualice la vista del mapa, con el cambio de posición de los vuelos detectados.

- El servicio hereda de la clase *IntentService* y el código que ejecuta es el siguiente:

```
@Override
protected void onHandleIntent(Intent intencion) {
        while(corriendo){
                SystemClock.sleep(7000);
                Intent i = new Intent();
                i.setAction(ReceptorOperacion.ACTION_RESP);
                i.addCategory(Intent.CATEGORY_DEFAULT);
                sendBroadcast(i);
                }
}
```

El código que ejecuta se trata de un bucle que el hilo itera mientras se cumplen las condiciones. Se mantiene dormido durante siete segundos y cuando se despierta crea un objeto Intent que posteriormente se configura con el evento definido y a continuación lo lanza.

- El receptor de eventos hereda de la clase *Broadcast Receiver* y tiene el siguiente aspecto:

```
public class ReceptorOperacion extends BroadcastReceiver {
 public static final String ACTION_RESP = "com.example.intentservice.intent.action.RESPUESTA_OPERACION";
 @Override
 public void onReceive(Context context, Intent intent) {
        mapa.clear();
        obtenerAviones();
        dibujarAviones();
}
```

Como era de esperar, cuando se recibe el evento se ejecutan los métodos que actualizan el mapa explicados anteriormente.

Para que el receptor de eventos esté activo es necesario que se instancie al crear la clase principal dentro del método *onCreate()* de la actividad y se registre el receptor:

```
//Receptor de anuncios:
IntentFilter filtro = new IntentFilter(ReceptorOperacion.ACTION_RESP);
filtro.addCategory(Intent.CATEGORY_DEFAULT);
receptor = new ReceptorOperacion();
registerReceiver(receptor, filtro);
…
```

5.3.2.4.Ciclo de vida de la aplicación

En el desarrollo de nuestra aplicación hemos tratado de ser cuidadosos en pequeños detalles que mejoran la experiencia del usuario como la capacidad de la conservación del contexto en el caso en que nuestra aplicación deje de estar en primer plano o que se gire el terminal, también de tratar de que sea una aplicación eficiente y que consuma los mínimos recursos de batería o procesamiento posibles, etc.

Para la implementación de esas pequeñas mejoras hemos implementado código en los métodos del ciclo de vida de nuestra aplicación. Hemos realizado los siguientes cambios:

```
//MÉTODOS DE CICLO DE VIDA DE LA APLICACIÓN:
@Override
protected void onPause() {
        super.onPause();
        SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(this);
        boolean servicio = pref.getBoolean("servicio", false);
        if (servicio){//Desactivar servicio }
}
@Override
protected void onResume() {
        super.onResume();
        SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(this);
        boolean servicio = pref.getBoolean("servicio", false);
        if (servicio){//Activar servicio}
        }
@Override
protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(receptor);
}
```

Con el objetivo de evitar la sobrecarga del terminal mientras nuestra aplicación no está en primer plano, modificamos los métodos que se ejecutan siempre que se produce este evento (del cambio de la aplicación en primer plano) para activar/desactivar el servicio que actualiza el mapa según convenga.

Siempre que nuestra actividad deje de estar en primer plano se ejecutará el método *onPause()*. Dentro del método, n primer lugar se lee la configuración del usuario y si el servicio está activado se matará su proceso. En cambio, si es nuestra actividad la que pasa a estar en primer plano se ejecutará el método *onResume()* que es implementado para realizar la función inversa. Si el sistema requiere de

recursos y mata el proceso de nuestra aplicación, en ese caso se ejecutará el método *onDestroy()*. Lo aprovechamos implementando el código que desactiva al receptor de eventos para liberar recursos.

Para la conservación del contexto de nuestra actividad realizamos cambios en los siguientes métodos:

```
@Override
protected void onSaveInstanceState(Bundle estadoGuardado){
   super.onSaveInstanceState(estadoGuardado);
   LatLng cameraLatLng = mapa.getCameraPosition().target;
   float cameraZoom = mapa.getCameraPosition().zoom;
   estadoGuardado.putInt("map_type", mapa.getMapType());
   ...
   }
@Override
protected void onRestoreInstanceState(Bundle estadoGuardado){
    super.onRestoreInstanceState(estadoGuardado);
    mapa.setMapType(estadoGuardado.getInt("map_type", GoogleMap.MAP_TYPE_NORMAL));
    double savedLat = estadoGuardado.getDouble("lat");
    ...
    dibujarAviones();
   }
```

En el primer método se guarda el estado de las variables en uso dentro del objeto *estadoGuardado* antes de que nuestra actividad abandone el primer plano, mientras que en el segundo se sigue el proceso inverso y posteriormente se actualiza el mapa.

5.3.3.Las clases de intercambio de datos

Las clases encargadas de realizar la función del intercambio de datos son prácticamente las mismas que en el caso del servicio web. Los cambios más notables son que cada una de las instancias de estas clases se ejecutan dentro de un hilo asíncrono auxiliar para no bloquear el hilo principal de la aplicación. Y también que hemos creado una nueva clase para poder comunicarnos mediante el servicio web SOAP llamada *ComunicacioSOAP*. En los siguientes apartados se muestra con más detalle los cambios que hemos expuesto anteriormente.

5.3.3.1.Uso del hilo auxiliar

Tal y como se ha indicado en apartados anteriores, para resolver el problema que surge al realizar peticiones por red, hemos escogido la solución de delegar este tipo de operaciones a procesos auxiliares e impedir de este modo que no se lleven a cabo en el proceso principal de la aplicación y que éste pueda quedar bloqueado. La clase que nos ha solucionado el problema recibe el nombre de *HiloAsincrono* y hereda de *AsyncTask*. Se implementa dentro de cada una de las clases de comunicaciones y tiene el siguiente aspecto:

```
class HiloAsincrono extends AsyncTask<String, Void, Object> {

   private ProgressDialog progreso;

   @Override protected void onPreExecute() {...}
   @Override protected Object doInBackground(String... cantidad) {
         //Operación de red.
         }
   @Override protected void onPostExecute(Object res) {...}

   @Override protected void onCancelled() {...}
    }
```

La clase consta de un atributo que es una vista de progreso (del proceso) que se muestra en pantalla mientras dura la operación y de cuatro métodos:

- **onPreExecute()**: Es el método que se ejecuta antes de la creación del hilo, es donde se suelen inicializar las vistas de progreso.
- **DoInBackground(...)**: El cuerpo de este método (al que se le pueden pasar parámetros) es la operación que realiza el proceso, en nuestro caso las peticiones en red.
- **OnPostExecute(...)**: Se ejecuta al finalizar el proceso.
- **OnCancelled()**: Se ejecuta si el proceso finaliza de manera inesperada.

Y para crear la instancia y que se realice la operación:

```
HiloAsincrono hilo = new HiloAsincrono();
hilo.execute(entrada);
try {
    res = hilo.get(5, TimeUnit.SECONDS);
     }catch...
         ...
```

5.3.3.2.K2soap-Android

Uno de los requisitos que tenía que cumplir nuestra aplicación era la de ser capaz de obtener los datos de los vuelos de un servidor que los ofreciera mediante un servicio web SOAP. Dado que la mayor parte de los servicios web que se utilizan para clientes en terminales móviles son de tipo REST, la información que se ha podido obtener para poder cumplir con el requerimiento no ha sido muy abundante, pero suficiente. La información encontrada apuntaba a la necesidad de importar una librería llamada K2soap-Android a nuestro proyecto. En capítulos anteriores que trataban sobre las tecnologías utilizadas hemos hecho referencia de la librería explicando su proyecto y de dónde descargarla. Pero en este apartado de implementación lo que vamos a hacer es explicar brevemente cómo la hemos utilizado en nuestra aplicación mostrando las líneas de código pertinentes.

Para consumir servicios web SOAP es necesario obtener primeramente el archivo con extensión WSDL, ya que en él están descritas las características del servicio y los parámetros necesarios para configurar nuestro cliente para poder obtener los datos. Los del parámetros del servidor necesarios son los siguientes:

```
...
this.NAMESPACE = "http://localizadoraviones";
this.URL = "http://"+host+":"+port+"/CliTcpSerSoap/services/MainActivity";
this.SOAP_ACTION = "http://"+host+":"+port+"/CliTcpSerSoap/services/MainActivity";
this.METHOD_NAME = "enviarAviones";
...
```

El fragmento de código es parte del método constructor de la clase de comunicaciones para las conexiones del tipo web SOAP. Tales parámetros se pasan al método constructor del objeto de la librería para crear la instancia con la que enviar/recibir datos:

```
...
        // Creamos el objeto soap request
        SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME);

        // Creamos el sobre (envelop), será utlizado para enviar las peticiones.
        SoapSerializationEnvelope envelope = new     SoapSerializationEnvelope(SoapEnvelope.VER11);
        envelope.setOutputSoapObject(request);
        envelope.addMapping(NAMESPACE, "SimpleObject", new ANave().getClass());
        HttpTransportSE androidHttpTransport = new HttpTransportSE(URL);
           try {
                 androidHttpTransport.debug = true;
                 androidHttpTransport.call(SOAP_ACTION, envelope);
                 vector = (Vector) envelope.getResponse();
                 arrayAvionAux = new ANave[vector.size()];
                  for (int i = 0; i < vector.size(); i++) {
                      SoapObject response = (SoapObject)vector.get(i);
                      if (response != null) {
                           ANave ret = new ANave();
                           ret.setAircraftID(response.getProperty("AircraftID").toString());
                          ...
                          ...
```

En primer lugar se crea el objeto, a continuación el *envelope* y ambos se configuran junto con los parámetros WSDL definidos, y posteriormente se realiza la petición que vuelca los datos dentro de un vector. Acto seguido se van extrayendo los datos uno a uno en formato de cadena de caracteres y se van construyendo los objetos de los vuelos pasando tales cadenas como parámetros.

6. PRUEBAS DE EJECUCIÓN

Para testear nuestra aplicación y comprobar que se cumplen los requisitos que ésta debe cumplir, se realizan una serie de pruebas donde se demuestra que se obtienen datos del servidor mediante servicios web SOAP y que además éstos se corresponden con los vuelos que existen en la realidad y en tiempo real. Dentro del contenido de este capítulo, de lo que se trata es de aportar estas pruebas realizadas ofreciendo capturas de pantalla, donde se evidencia el correcto funcionamiento de nuestra aplicación.

Existen aplicaciones gratuitas en el mercado *Play Store* de Google que sirven para mostrar el tráfico aéreo en tiempo real. *Flightradar24* es un ejemplo de aplicación bastante conocida dentro de este conjunto de aplicaciones que ofrecen esa clase de servicio. Para realizar nuestras pruebas nos hemos descargado la aplicación y hemos comprobado que el aspecto del tráfico aéreo detectado que muestra nuestra aplicación es prácticamente el mismo, salvo en algunos casos aislados (en los que se detectan vuelos que por su ámbito no es posible que sean detectados por la antena de la UPV), si las ejecutamos simultáneamente y para la misma área geográfica dentro del mapa. También hemos detectado que la aplicación descargada nos muestra el tráfico con un retardo mayor con respecto al de nuestra aplicación, aunque la diferencia no es relevante ya que se trata de menos de un minuto y nuestro rol no es precisamente el de controlador aéreo profesional.

En la página siguiente mostramos un par de capturas de pantalla del tráfico aéreo detectado en cada una de las aplicaciones. La figura del margen izquierdo se corresponde con una captura realizada desde nuestra aplicación, mientras que la otra se corresponde con una captura de la aplicación descargada del mercado de Google, *Flightradar24.*

Como puede observarse, ambas capturas de pantalla abarcan aproximadamente el mismo área geográfica y los vuelos que se detectan son prácticamente los mismos. En ambas capturas hemos seleccionado el mismo vuelo (en nuestra aplicación esto se percibe con una etiqueta sobre el vuelo seleccionado, mientras que en la descargada con el vuelo de color rojo) y el nombre de los vuelos coinciden. Por lo que se demuestra que el tráfico que se muestra se corresponde con la realidad.

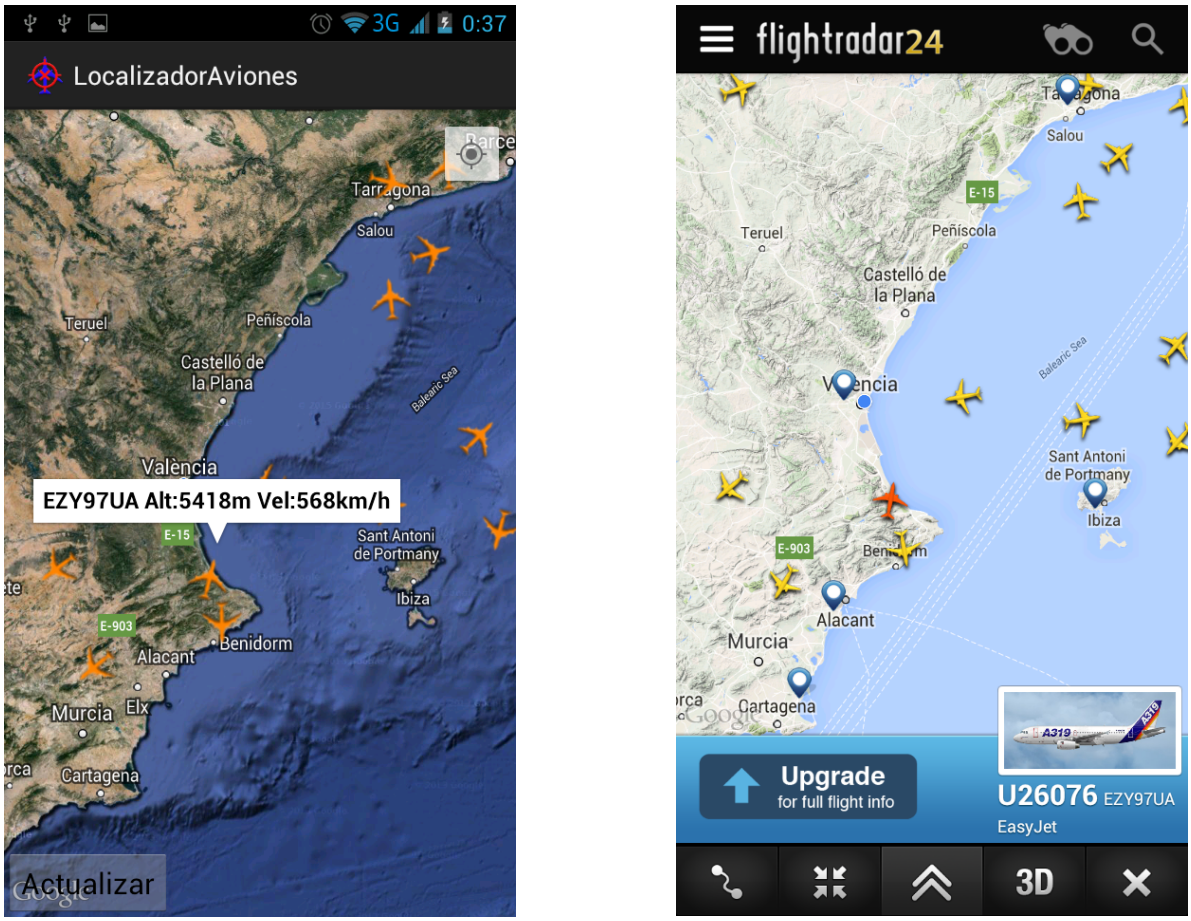

Fig. 6.1: Capturas de pantalla de aplicaciones diferentes.

Otra de las evidencias que queremos mostrar en este capítulo es la que demuestra que los datos se transfieren en el formato requerido de servicio web SOAP, cumpliendo de ese modo con otro de los requisitos.

En la página que se presenta a continuación se expone una captura de pantalla (se muestra en vertical para que los datos que aparecen sean legibles) que contiene una ventana del programa *Wireshark* (programa que monitoriza el tráfico en red) con el contenido de los paquetes que se han adquirido de la conversación en red de nuestro cliente Android con el servidor web SOAP, donde se pone de manifiesto que los datos se envían en formato SOAP y que la conversación se corresponde con los datos de tráfico aéreo representados en la anterior captura de la aplicación Android. Esto es así y puede comprobarse fijándonos en el texto que se ha seleccionado, que se refiere al vuelo **EZY97UA** (campo *Callsign*) en ambas capturas de pantalla.

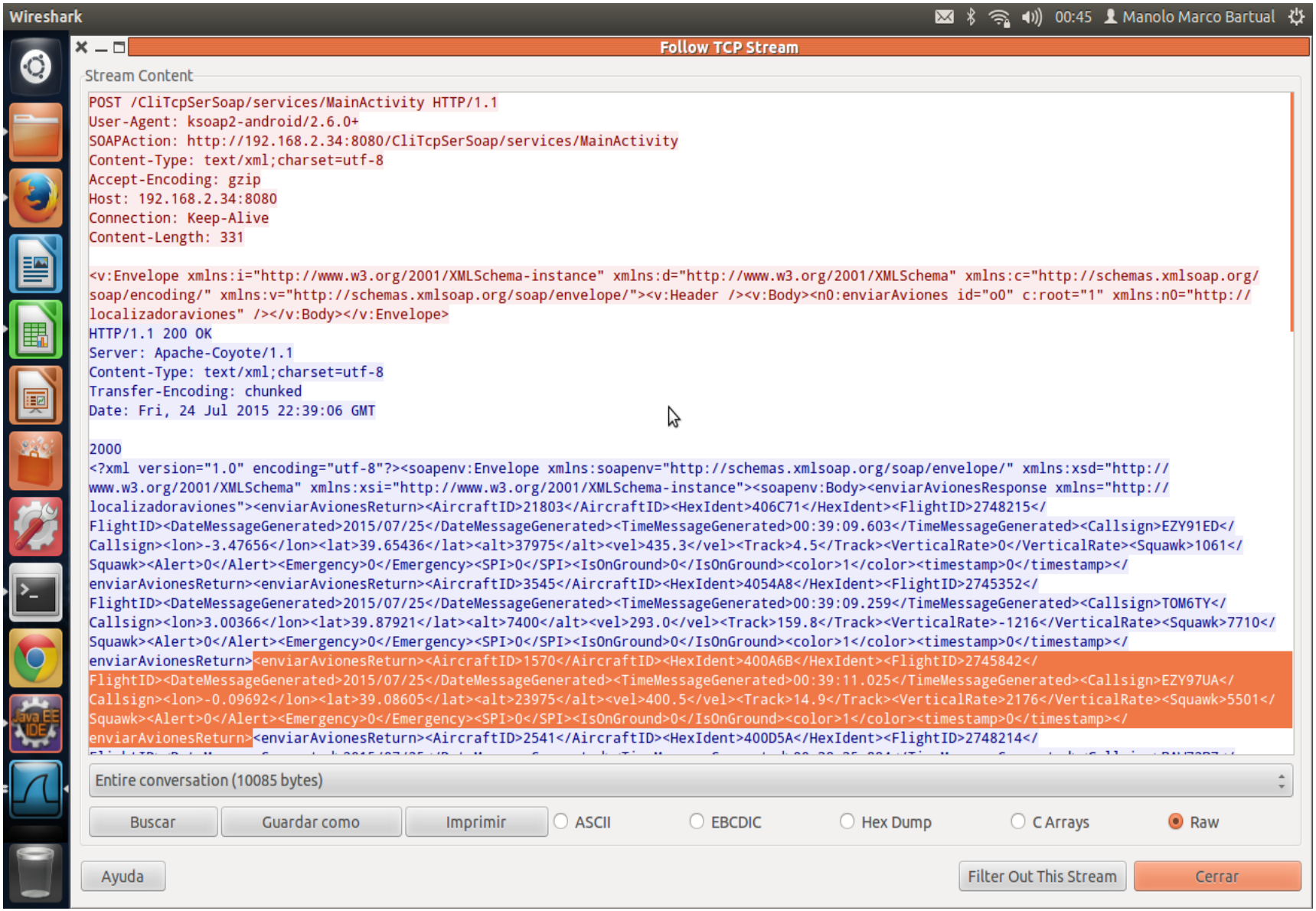

Fig. 6.2: Captura de conversación en red con Wireshark.

7. Conclusiones y trabajo futuro

El trabajo realizado en esta tesina de final de máster ha sido la implementación de un servicio web SOAP que ofrece los datos del tráfico aéreo de el servidor/antena que existe actualmente en la UPV y su posterior despliegue en un servidor. Además, también se ha desarrollado un cliente para la plataforma Android capaz de consumir tanto el servicio web SOAP creado, como los datos ofrecidos directamente desde el servidor/antena de la UPV.

Haciendo una valoración el trabajo realizado, podemos sacar conclusiones de lo que se ha conseguido, lo que se ha hecho bien y también de lo que ha quedado mejorable. Para todo lo que podía mejorarse (desde mi punto de vista) se ha decidido incluir un apartado llamado *Trabajo futuro* en el que se recogen ideas que pueden mejorar el proyecto. Para el resto de las conclusiones, se confecciona el siguiente apartado.

7.1. Conclusiones

Tomando como pruebas las aportadas en el capítulo anterior, podemos estar en condiciones de afirmar que el proyecto cumple con las especificaciones y requerimientos iniciales. Y en lo que se refiere a los objetivos personales de esta tesina final de máster, que son los de aprender el desarrollo de aplicaciones para dispositivos móviles y la creación/consumo de servicios web, se ha decidido clasificar las conclusiones en los tres siguiente apartados.

7.1.1. Aprendizaje en el desarrollo

Mi caso es el de un informático de sistemas que jamás había cursado (antes de emprender el proyecto de la tesina) alguna asignatura para el desarrollo de aplicaciones móviles, con lo que el punto de partida ha sido el de un nivel de escasos conocimientos al respecto, prácticamente nivel cero.

Se considera importante señalar, que la elección como plataforma de desarrollo al sistema operativo Android nos ha facilitado bastante el trabajo, debido a que el lenguaje de programación que se emplea es Java, un lenguaje de programación que ya se aprendió previamente durante los años de la ingeniería.

La evolución de mis conocimientos como desarrollador en aplicaciones móviles durante el tiempo empleado para la realización de esta tesina, va desde la creación de la primera aplicación típica de *Hola Mundo* hasta lo que es la aplicación que se incluye en la entrega. Un cúmulo de conocimientos adquiridos de varias fuentes como libros, páginas web, cursos *online*, etc., hasta llegar al punto de considerar que se tienen los fundamentos suficientes como para realizar cualquier tipo de aplicación de una dificultad media.

7.1.2.Contexto actual en plataformas y app móviles

Para tomar la decisión sobre la plataforma, tecnologías, etc., a utilizar en el proyecto, ha sido necesario dedicar bastante tiempo para hacerse una composición de lugar al respecto de la situación y tendencias actuales, y elegir lo más apropiado.

En las decisiones tomadas el argumento que mayor peso ha tenido, ha sido siempre el de la opción más comúnmente utilizada. Es de lógica que si la mayoría de gente emplea cierto tipo de tecnología para unos objetivos, el uso de tal tecnología para tales objetivos sea un acierto, ya que sería extraño que la mayoría de la gente estuviera equivocada. Y respecto a la elección de la plataforma, se ha optado por la de Android porque es la que tiene mayor número de usuarios en la actualidad y lo que se busca es que nuestra aplicación pueda llegar a la mayor cantidad de gente posible.

Otra de las conclusiones que se exponen en este apartado es la de resaltar la importancia que ha tenido ese tiempo empleado, ya que me ha permitido actualizar los conocimientos al respecto y estar al día dentro de un mundo tan dinámico como es el de la informática.

7.1.3.Mercado laboral

Las posibilidades de encontrar trabajo dada la situación del mercado laboral actual para los profesionales con un perfil de administrador de sistemas, no son las mismas que las que podrían tener los profesionales con un perfil de programador Java. La cantidad de ofertas de trabajo que se anuncian para los profesionales con un perfil de administrador de sistemas es mucho menor, y además se exige una mayor cantidad de conocimientos y de experiencia. Por otro lado, otro perfil de profesional (con el auge de los servidores de aplicaciones desplegando servicios web) como es el de programador Java, se está demandando con mayor asiduidad en el mercado.

Dado el contexto laboral actual, se considera la realización de esta tesina final de máster como un conocimiento de gran importancia para ser añadido al CV con la finalidad de encajar con la demanda laboral y tener así más opciones de encontrar un buen trabajo.

7.2.Trabajo futuro

En líneas generales, el trabajo realizado en este proyecto se valora positivamente, aunque existen ciertos aspectos que podrían mejorarse o incluso ampliar su funcionalidad. A continuación se expone una serie de propuestas que se considera que podrían ser oportunas para la mejora del proyecto:

- **Comunicaciones cifradas.** Hacer que los datos viajen cifrados por la red no sería una tarea descabellada teniendo en cuenta que las conexiones de red tienen un ancho de banda cada vez mayor, pero no sería algo imprescindible ya que la información que se envía está al alcance de cualquier individuo que se instale una aplicación gratuita para el rastreo de vuelos. Aunque sí que podría resultar más interesante dentro de un contexto en el que los datos sean confidenciales y en un ámbito privado.

- **Soporte REST.** La mayor parte de las aplicaciones que consumen servicios web en la actualidad lo hacen mediante servicios web REST, añadir esta funcionalidad tanto en el servidor como en la aplicación móvil podría ser una buena idea para adquirir conocimientos complementarios.

- **Mejora de la interfaz de usuario.** La interfaz de usuario que se muestra en la aplicación móvil cumple básicamente con los requisitos funcionales establecidos, pero haciendo un poco de autocrítica diremos que comparada con las interfaces que se ofrecen dentro de las aplicaciones de este mismo tipo en el mercado, ésta se queda algo simple. De todos modos, los objetivos del proyecto no se centran en el aprendizaje del diseño gráfico.

En general, el trabajo realizado en esta tesina final de máster ha sido una experiencia bastante enriquecedora ya que me ha sido útil para aprender y poner en práctica varias tecnologías de programación con las que no había trabajado anteriormente.

8.ANEXOS

El contenido de este capítulo es el código completo implementado en alguna de las clase que consideramos más importantes.

8.1.Clases del servidor web SOAP

8.1.1.MainActivity.java

```
package localizadoraviones;
import java.util.Set;
public class MainActivity  {
        private ComunicacioInt canal = null;
        private String servidor;
        private int puerto;
        private static java.util.Hashtable<String, ANave> aNaves = new java.util.Hashtable<String, ANave>();

  public MainActivity(String host, int port){
        this.servidor = host;
        this.puerto = port;
  }
  public MainActivity(){
        this("158.42.40.150", 2220);//Dirección y puerto del servidor/antena de la UPV
        }
  public ANave[] enviarAviones(){
        ANave[] r = null;
        ANave auxA;
        if(obtenerAviones("158.42.40.150", 2220)) {
                Set<String> llaves = MainActivity.aNaves.keySet();
                int i = 0;
                if (llaves.size() > 0){
                        r = new ANave[llaves.size()];
                        for(String llave: llaves){
                                auxA = aNaves.get(llave);
                                r[i] = auxA;
                                i++;
                                }
                        }
                }else{System.out.println("No se han recibido los aviones correctamente.");}
        return r;
  }
  private boolean obtenerAviones(String servidor, int puerto){
        canal = new ComunicacioRawTcp(servidor, puerto);
        byte[] rebut_byte = null;
        if(canal != null){                          //El canal esta creado correctamente
          Object aux = canal.receiveObject();
          if(aux instanceof byte[]){
             rebut_byte = (byte[])aux;
                if(rebut_byte != null){//Habemos recibido una actualización correctamente.
                aNaves.clear();
                String linia;
                java.io.ByteArrayInputStream bais = new java.io.ByteArrayInputStream(rebut_byte);
                java.io.BufferedReader br = new java.io.BufferedReader(new java.io.InputStreamReader(bais));
                boolean acabat = false;    //Típica variable de fin.
                int i = 0;//Contador de aeronave recibidas.
                 while(!acabat){
                        try{
                        linia = br.readLine();
                                if(linia == null){
                                  acabat = true;
                                  }else{
                                  //Tenemos una aeronave y la insertamos.
                                  linia = linia.trim();
                                  String[] valores = linia.split(",");
                                  if(valors.length == 17){
```

(Continúa)

```
//Nos aseguramos de que las keys de la hash serán secuenciales para hacer una búsqueda coherente.
                                    try{
                                    aNaves.put(String.valueOf(i),new ANave(valores[0],valores[1], valores[2],
valores[3], valores[4], valores[5], valores[6], valores[7], valores[8], valores[9], valores[10], valores[11],
valores[12], valores[13], valores[14], valores[15], valores[16], 1, (long)0));
                                    i++;
                                    }catch(java.lang.NoSuchMethodError e){
                                    System.err.println("NoSuchMethodError: " + e.getMessage());
                                      }
                                    }
                                  }
                              }catch(java.io.IOException e){
                              System.err.println("Error d'entrada/sortida");
                              return false;
                               }
                    }//while terminado
            }
            }else{
              System.out.println("Error de entrada/salida: No hemos recibido un byte[]. ");
              return false;
            }//si el objeto no es de tipo byte[]
            }else{
              System.out.println("Error de entrada/salida: canal es null. ");
              return false;
         }//si canal es diferente de null...
         return true;
      }

}
```

8.1.2.MainActivity.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://localizadoraviones" xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://localizadoraviones" xmlns:intf="http://localizadoraviones"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://localizadoraviones"
xmlns="http://www.w3.org/2001/XMLSchema">
   <element name="enviarAviones">
    <complexType/>
   </element>
   <element name="enviarAvionesResponse">
    <complexType>
     <sequence>
      <element maxOccurs="unbounded" name="enviarAvionesReturn" type="impl:ANave"/>
     </sequence>
    </complexType>
   </element>
   <complexType name="ANave">
    <sequence>
     <element name="AircraftID" nillable="true" type="xsd:string"/>
     <element name="HexIdent" nillable="true" type="xsd:string"/>
     <element name="FlightID" nillable="true" type="xsd:string"/>
     <element name="DateMessageGenerated" nillable="true" type="xsd:string"/>
     <element name="TimeMessageGenerated" nillable="true" type="xsd:string"/>
     <element name="Callsign" nillable="true" type="xsd:string"/>
     <element name="lon" nillable="true" type="xsd:string"/>
     <element name="lat" nillable="true" type="xsd:string"/>
     <element name="alt" nillable="true" type="xsd:string"/>
     <element name="vel" nillable="true" type="xsd:string"/>
     <element name="Track" nillable="true" type="xsd:string"/>
     <element name="VerticalRate" nillable="true" type="xsd:string"/>
     <element name="Squawk" nillable="true" type="xsd:string"/>
     <element name="Alert" nillable="true" type="xsd:string"/>
     <element name="Emergency" nillable="true" type="xsd:string"/>
     <element name="SPI" nillable="true" type="xsd:string"/>
     <element name="IsOnGround" nillable="true" type="xsd:string"/>
     <element name="color" type="xsd:int"/>
     <element name="timestamp" type="xsd:long"/>
    </sequence>
   </complexType>
  </schema>
</wsdl:types>
```

(Continúa)

```
  <wsdl:message name="enviarAvionesResponse">
      <wsdl:part element="impl:enviarAvionesResponse" name="parameters">
      </wsdl:part>
   </wsdl:message>
   <wsdl:message name="enviarAvionesRequest">
      <wsdl:part element="impl:enviarAviones" name="parameters">
      </wsdl:part>
   </wsdl:message>
   <wsdl:portType name="MainActivity">
      <wsdl:operation name="enviarAviones">
         <wsdl:input message="impl:enviarAvionesRequest" name="enviarAvionesRequest">
       </wsdl:input>
         <wsdl:output message="impl:enviarAvionesResponse" name="enviarAvionesResponse">
       </wsdl:output>
      </wsdl:operation>
    </wsdl:portType>
   <wsdl:binding name="MainActivitySoapBinding" type="impl:MainActivity">
      <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="enviarAviones">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="enviarAvionesRequest">
            <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="enviarAvionesResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="MainActivityService">
      <wsdl:port binding="impl:MainActivitySoapBinding" name="MainActivity">
         <wsdlsoap:address location="http://localhost:8080/CliTcpSerSoap/services/MainActivity"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

8.2.Clases de la aplicación móvil

8.2.1.MainActivity.java

```
package com.example.localizadoraviones;
public class MainActivity extends FragmentActivity implements OnMapClickListener {

        private ComunicacioInt canal = null;
        private SharedPreferences pref;
        private static GoogleMap mapa;
        private static LatLng UPV = new LatLng(39.481106, -0.340987);
        private java.util.Hashtable<String, ANave> aNaves = new java.util.Hashtable<String, ANave>();
        private ReceptorOperacion receptor;
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                // Comprueba el estado de Google Play Services
                int status = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);
                // Comprueba si está disponible y muestra si hay error.
                try {
                    if (status != ConnectionResult.SUCCESS) {
                        GooglePlayServicesUtil.getErrorDialog(status, this, status).show();
                    }
                } catch (Exception e) {
                    Log.e("Error: GooglePlayServiceUtil: ", "" + e);
                }
                mapa = ((SupportMapFragment)
getSupportFragmentManager().findFragmentById(R.id.map)).getMap();
                mapa.setMapType(GoogleMap.MAP_TYPE_HYBRID);
                mapa.setMyLocationEnabled(true);
                mapa.getUiSettings().setZoomControlsEnabled(false);
                mapa.getUiSettings().setCompassEnabled(true);
                mapa.moveCamera(CameraUpdateFactory.newLatLngZoom(UPV, 7));
                mapa.setOnMapClickListener(this);

                //Receptor de anuncios:
                IntentFilter filtro = new IntentFilter(ReceptorOperacion.ACTION_RESP);
                filtro.addCategory(Intent.CATEGORY_DEFAULT);
                receptor = new ReceptorOperacion();
                registerReceiver(receptor, filtro);

      }(Continúa)
```

```
      @Override
        public boolean onCreateOptionsMenu(Menu menu) {
               super.onCreateOptionsMenu(menu);
               MenuInflater inflater = getMenuInflater();
               inflater.inflate(R.menu.menu, menu);
               return true;
            }
        @Override
         public boolean onOptionsItemSelected(MenuItem item) {
       switch (item.getItemId()) {
       case R.id.acercaDe:
             lanzarAcercaDe(null);
             break;
       case R.id.config:
           Intent i = new Intent(this, Preferencias.class);
           startActivity(i);
           break;
       }
       return true;
      }
        @Override
        public void  onMapClick(LatLng puntoPulsado) {}
        public void lanzarAcercaDe(View view){
              Intent i = new Intent(this, AcercaDe.class);
                    startActivity(i);
        }
        public void actualizaMapa(View view){
                 mapa.clear();
                 //mostrarPreferencias(); Método para testear las opciones que se eligen.
                 obtenerAviones();
                 dibujarAviones();
        }
    private void obtenerAviones(){
        SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(this);
        String servidor = pref.getString("servidor", "192.168.1.33");
        String sPuerto = pref.getString("puerto", "2220");
        int puerto = Integer.parseInt(sPuerto);
        String sConexion = pref.getString("conexión", "0");
        int conexion = Integer.parseInt(sConexion);
        if(conexion == 0){
                 canal = new ComunicacioTcp(servidor, puerto, this);
        }else if(conexion == 1){
                 canal = new ComunicacioUdp(servidor, puerto, this);
        }else if(conexion == 2){
                 canal = new ComunicacioSOAP(servidor, puerto, this);
        }else if (conexion == 3){
                 canal = new ComunicacioRawTcp(servidor, puerto, this);
        }else{
                 canal = new ComunicacioRawUdp(servidor, puerto, this);
        }
        if(canal != null){                          //El canal esta creado correctamente
        Object aux = canal.receiveObject();
        switch (conexion) {
          case 0://Para la conexión tipo TCP
          case 1://Para la conexión tipo UDP
          case 3://Para la conexión tipo TCP Raw
          case 4://Para la conexión tipo UDP Raw
           if(aux instanceof byte[]){
            byte[] rebut_byte = (byte[])aux;
            if(rebut_byte != null){//Habemos recibido una actualización correctamente.
                 aNaves.clear();
                 String linia;
                 java.io.ByteArrayInputStream bais = new java.io.ByteArrayInputStream(rebut_byte);
                 java.io.BufferedReader br = new java.io.BufferedReader(new java.io.InputStreamReader(bais));
                 boolean terminado = false; //Típica variable de fin.
                 Integer i = 0;    //Contador de naves recibidas.
                 while(!terminado){
                   try{
                          linia = br.readLine();
                          if(linia == null){
                          acabat = true;
                          }else{
                          linia = linia.trim();
                          String[] valores = linia.split(",");
                          if(valors.length == 17){
                          try{
                          aNaves.put(String.valueOf(i),new ANave (valores[0],valores[1], valores[2],
valores[3], valores[4], valores[5], valores[6], valores[7], valores[8], valores[9], valores[10], valores[11],
valores[12], valores[13], valores[14], valores[15], valores[16], 1, (long)0));
                          i++;
                     }catch(java.lang.NoSuchMethodError e){
                          System.err.println("NoSuchMethodError: " + e.getMessage());
                           }
                    }
                  }
```

(Continúa)

```
                }catch(java.io.IOException e){
                  Toast.makeText(this, "Error de entrada/salida.", Toast.LENGTH_SHORT).show();
                  }
                }//while acabat
              }
        }else if(aux instanceof String) {
          String s = (String)aux;
          Toast.makeText(this, s, Toast.LENGTH_LONG).show();
          if(conexion==1 || conexion==4)
           {
             if(s.equals("Error, agotado el tiempo de espera al Servidor.\nHay que reiniciar el proceso de la
aplicación.")){
Toast.makeText(this, "Es necesario reiniciar la aplicación y matar el proceso...", Toast.LENGTH_LONG).show();
                }
            }
          }
        else{
          String s = new String("Error, se ha recibido algo inesperado");
          Toast.makeText(this, s, Toast.LENGTH_LONG).show();
          }//si el objeto no es de tipo byte[]
        break;
        case 2://Para la conexión tipo SOAP
                aNaves.clear();
                if(aux instanceof ANave[]) {
                   ANave[] arrayAviones = (ANave[])aux;
                   if (arrayAviones.length != 0) {
                         for (int i = 0;i<arrayAviones.length;i++){
                           aNaves.put(arrayAviones[i].Callsign, arrayAviones[i]);
                         }
                   }else{
                     String s = new String("El Servidor SOAP no responde.");
                         Toast.makeText(this, s, Toast.LENGTH_SHORT).show();
                         }
                   }
        break;
        default:
                String s = new String("Error, La variable \"conexion\" tiene un valor inesperado.");
                Toast.makeText(this, s, Toast.LENGTH_LONG).show();
        break;
        }
      }else{
        String s = new String("El objeto de comunicaciones (Canal) no se ha inicializado correctamente");
        Toast.makeText(this, s, Toast.LENGTH_SHORT).show();
       }//si canal es diferente de null...
    }
    private void dibujarAviones(){
        Set<String> llaves = aNaves.keySet();////Dibujamos sólo los aviones que se detectan.
        for(String llave: llaves){
                ANave aux = aNaves.get(llave);
                MarkerOptions maux = new MarkerOptions().position(new LatLng(Double.valueOf(aux.lat),
Double.valueOf(aux.lon))).icon(BitmapDescriptorFactory.fromResource(R.drawable.aereonave))
                .rotation(Float.valueOf(aux.Track)).title(aux.Callsign + "\nAlt:" +
                Math.round(Double.valueOf(aux.alt)*0.3048) + "m" + "\nVel:" +
                Math.round(Double.valueOf(aux.vel)*1.609344) +"km/h");
                mapa.addMarker(maux);
        }
      }
     public void mostrarPreferencias() {
        SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(this);
        String s = "Servidor: "+ pref.getString("servidor", "?") + ", Puerto: " +
        pref.getString("puerto", "?") + ", Conexión: " + pref.getString("conexión", "3");
        Toast.makeText(this, s, Toast.LENGTH_SHORT).show();

     }//MÉTODOS DE CICLO DE VIDA DE LA APLICACIÓN:

        @Override
        protected void onPause() {
                super.onPause();
                SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(this);

                boolean servicio = pref.getBoolean("servicio", false);
                if (servicio){
                        Toast.makeText(this, "Servicio desactivado", Toast.LENGTH_SHORT).show();
                        stopService(new Intent(MainActivity.this, Servicio.class));
                }
                //DETENER EL SERVICIO

        }
```

(Continúa)

```
@Override

        protected void onResume() {
                super.onResume();
                SharedPreferences pref = PreferenceManager.getDefaultSharedPreferences(this);

                boolean servicio = pref.getBoolean("servicio", false);
                if (servicio){
                        Toast.makeText(this, "Servicio activado", Toast.LENGTH_SHORT).show();
                        ComponentName debug = startService(new Intent(this,
com.example.localizadoraviones.MainActivity.Servicio.class));
                }
        }

        @Override
        protected void onDestroy() {
                super.onDestroy();
                unregisterReceiver(receptor);
        }

        //MÉTODOS PARA CONSERVAR EL ESTADO DE LA ACTIVIDAD:
        @Override
           protected void onSaveInstanceState(Bundle estadoGuardado){
                 super.onSaveInstanceState(estadoGuardado);
                 LatLng cameraLatLng = mapa.getCameraPosition().target;
                 float cameraZoom = mapa.getCameraPosition().zoom;
                 estadoGuardado.putInt("map_type", mapa.getMapType());
                 estadoGuardado.putDouble("lat", cameraLatLng.latitude);
                 estadoGuardado.putDouble("lng", cameraLatLng.longitude);
                 estadoGuardado.putFloat("zoom", cameraZoom);
                 estadoGuardado.putSerializable("aNaves", aNaves);

           }

           @Override
           protected void onRestoreInstanceState(Bundle estadoGuardado){
                 super.onRestoreInstanceState(estadoGuardado);
                 mapa.setMapType(estadoGuardado.getInt("map_type", GoogleMap.MAP_TYPE_NORMAL));
                 double savedLat = estadoGuardado.getDouble("lat");
                 double savedLng = estadoGuardado.getDouble("lng");
                 mapa.animateCamera(CameraUpdateFactory.newLatLngZoom(new LatLng(savedLat, savedLng),
estadoGuardado.getFloat("zoom", 7)));
                 aNaves = (java.util.Hashtable<String, ANave>)estadoGuardado.getSerializable("aNaves");
                 dibujarAviones();
           }
//CLASE DEL SERVICIO PARA LA ACTUALIZACIÓN AUTOMÁTICA DE LOS AVIONES EN EL MAPA:
        public static class Servicio extends IntentService {
                boolean corriendo;

        public Servicio () {
                super("Servicio");
                corriendo = true;
        }
        @Override
        protected void onHandleIntent(Intent intencion) {
                while(corriendo){
                        SystemClock.sleep(7000);
                        Intent i = new Intent();
                        i.setAction(ReceptorOperacion.ACTION_RESP);
                        i.addCategory(Intent.CATEGORY_DEFAULT);
                        sendBroadcast(i);
                }
        }
        @Override
        public void onDestroy() {
                corriendo = false;
                }
       }
//CLASE RECEPTOR DE ANUNCIOS PARA LA ACTUALIZACIÓN AUTOMÁTICA DE LOS AVIONES EN EL MAPA:

        public class ReceptorOperacion extends BroadcastReceiver {
        public static final String ACTION_RESP="com.example.intentservice.intent.action.RESPUESTA_OPERACION";
                @Override
                public void onReceive(Context context, Intent intent) {
                        mapa.clear();
                        obtenerAviones();
                        dibujarAviones();
                }

        }


}
```

8.2.2.AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.localizadoraviones"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="18" />
    <permission
         android:name="com.example.localizadoraviones.permission.MAPS_RECEIVE"
         android:protectionLevel="signature" />
         <uses-permission
         android:name="com.example.localizadoraviones.permission.MAPS_RECEIVE"/>
         <uses-permission
         android:name="com.google.android.providers.gsf.permission.READ_GSERVICES"/>
         <uses-permission android:name="android.permission.INTERNET" />
         <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
         <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
         <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
         <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>

         <uses-feature
        android:glEsVersion="0x00020000"
        android:required="true" />


    <application
        android:allowBackup="true"
        android:icon="@drawable/avion"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <meta-data android:name="com.google.android.gms.version"
        android:value="@integer/google_play_services_version" />
        <meta-data
            android:name="com.google.android.maps.v2.API_KEY"
            android:value="AIzaSyDwghqFNX4H9pelBirhi5MkunQ5ikNu6Sc"/>
        <activity
            android:name="com.example.localizadoraviones.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

        </activity>
        <activity android:name=".AcercaDe" android:label="Acerca de ..."
         android:theme="@android:style/Theme.Dialog"></activity>
        <activity android:label="Preferencias" android:name="Preferencias"></activity>
        <service android:enabled="true" android:name="com.example.localizadoraviones.MainActivity$Servicio" />

    </application>
    <meta-data
    android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />

</manifest>
```

8.2.3.ComunicacioSoap.java

```
package com.example.localizadoraviones;
public class ComunicacioSOAP implements ComunicacioInt{
        int port;
        String host = null;
        Context context;
        private final String NAMESPACE;
    private final String URL;
    private final String SOAP_ACTION;
    private final String METHOD_NAME;

        public ComunicacioSOAP(String host, int port, Context context){
                super();
                this.port = port;
                this.host = host;
                this.context = context;
                this.NAMESPACE = "http://localizadoraviones";
                this.URL = "http://"+host+":"+port+"/CliTcpSerSoap/services/MainActivity";
                this.SOAP_ACTION = "http://"+host+":"+port+"/CliTcpSerSoap/services/MainActivity";
                this.METHOD_NAME = "enviarAviones";
        }
        public Object receiveObject(){

                Object res = new Object();
                String[] entrada = new String[2];
                entrada[0] = host;
                entrada[1] = String.valueOf(port);
                HiloAsincrono hilo = new HiloAsincrono();
                hilo.execute(entrada);
                try {
                        res = hilo.get(7, TimeUnit.SECONDS);

                } catch (InterruptedException e) {
                        e.printStackTrace();
                } catch (ExecutionException e) {
                        e.printStackTrace();
                } catch (TimeoutException e) {
                        e.printStackTrace();
                        res = new String("Error, agotado el tiempo de espera al Servidor.");
                        hilo.progreso.cancel();
                }
                return res;
        }
class HiloAsincrono extends AsyncTask<String, Void, Object> {
        private ProgressDialog progreso;

        @Override protected void onPreExecute() {
              progreso = new ProgressDialog(context);
              progreso.setProgressStyle(ProgressDialog.STYLE_SPINNER);
              progreso.setMessage("Accediendo al servidor...");
              progreso.setCancelable(false);
              progreso.show();
        }
 @Override protected Object doInBackground(String... cantidad) {

         String host = cantidad[0];
         int port = Integer.parseInt(cantidad[1]);
         Vector vector = null;
         ANave[] arrayAvionAux = null;
         java.util.Hashtable<String, ANave> avionesAux = new java.util.Hashtable<String, ANave>();
                 if(host != null){//Estamos bien configurados
                         // Creamos el objeto soap request
                         SoapObject request = new SoapObject(NAMESPACE, METHOD_NAME);
// Creamos el sobre (envelop), será utlizado para enviar las peticiones.
                         SoapSerializationEnvelope envelope = new
SoapSerializationEnvelope(SoapEnvelope.VER11);
                envelope.setOutputSoapObject(request);
                envelope.addMapping(NAMESPACE, "SimpleObject", new ANave().getClass());
                HttpTransportSE androidHttpTransport = new HttpTransportSE(URL);
                try {
androidHttpTransport.debug = true;
                 androidHttpTransport.call(SOAP_ACTION, envelope);
                 vector = (Vector) envelope.getResponse();
                 arrayAvionAux = new ANave[vector.size()];
                  for (int i = 0; i < vector.size(); i++) {
                      SoapObject response = (SoapObject)vector.get(i);
                      if (response != null) {
```

(Continúa)

```
                ANave ret = new ANave();
                ret.setAircraftID(response.getProperty("AircraftID").toString());
                ret.setHexIdent(response.getProperty("HexIdent").toString());
                ret.setFlightID(response.getProperty("FlightID").toString());
                ret.setDateMessageGenerated(response.getProperty("DateMessageGenerated").toString());
                ret.setTimeMessageGenerated(response.getProperty("TimeMessageGenerated").toString());
                ret.setCallsign(response.getProperty("Callsign").toString());
                ret.setLon(response.getProperty("lon").toString());
                ret.setLat(response.getProperty("lat").toString());
                ret.setAlt(response.getProperty("alt").toString());
                ret.setVel(response.getProperty("vel").toString());
                ret.setTrack(response.getProperty("Track").toString());
                ret.setVerticalRate(response.getProperty("VerticalRate").toString());
                ret.setSquawk(response.getProperty("Squawk").toString());
                ret.setAlert(response.getProperty("Alert").toString());
                ret.setEmergency(response.getProperty("Emergency").toString());
                ret.setSPI(response.getProperty("SPI").toString());
                ret.setIsOnGround(response.getProperty("IsOnGround").toString());
                ret.setColor(Integer.parseInt(response.getProperty("color").toString()));
                ret.setTimestamp(Long.parseLong(response.getProperty("timestamp").toString()));
                arrayAvionAux[i] = ret;
              }
            }
         }catch (IOException e) {
            Log.e("CLASE ASINCRONA", "HA SALTADO LA IOException AL HACER LA LLAMADA SOAP");
            e.printStackTrace();
            arrayAvionAux = new ANave[0];
            return arrayAvionAux;
           }catch (XmlPullParserException e) {
               Log.e("CLASE ASINCRONA", "HA SALTADO LA XmlPullParserException AL HACER LA LLAMADA SOAP");
               e.printStackTrace();
               arrayAvionAux = new ANave[0];
               return arrayAvionAux;
               }
            }
         return arrayAvionAux;
        }
 @Override protected void onPostExecute(Object res) {
              progreso.dismiss();

        }

        @Override protected void onCancelled() {
              progreso.dismiss();
        }
    }

}
```

9. BIBLIOGRAFÍA

- Annex 10 to the Convention on International Civil Aviation: "Aeronautical Telecommunications", Volume IV (ICAO).
- Cesare Pautasso, Olaf Zimmermann, Frank Leymann,RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. April 2008.
- José Abraham Rodríguez López, "Servicios Web: Estado del Arte y Frameworks de Desarrollo".
- Avionics magazine: "Avionics News", April 2005.
- Developer Economics: "State of the developer nation QI 2015" febrero 2015.
- Jesús Tomás Gironés, "El gran libro de Android" 2013.
- http://devloper.android.com/
- http://code.google.com/
- http://docs.oracle.com
- http://www.wikipedia.org
- http://axis.apache.org
- https:/plumbr.eu/
- https://www.netmarketshare.com/
- http://qs.statconunter.com/
- http://woodair.net/
- http://www.skybrary.aero/