UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Trabajo de Fin de Grado

DESARROLLO DE UN SISTEMA PARA LA RECEPCIÓN DE LA SEÑAL TRANSPONDEDOR Y PARA LA PRESENTACIÓN Y TRANSMISIÓN DE DATOS RADAR

5 de julio de 2017

Autor: Nazariy Harvat

Tutor: Juan Antonio Vila Carbó

Cotutor: Pedro Yuste Pérez

Índice general

1. Motivación y objetivos

El objetivo de este Trabajo de Fin de Grado es el desarrollo de un sistema capaz de proporcionar un servicio telemático de radar con información de vigilancia, procedente de las aeronaves que sobrevuelan todo el territorio español, en forma de datos que puedan ser procesados por los clientes del servicio. Se basará en la ubicación de varias antenas que proporcionen cobertura a zonas geográficas concretas y en la posterior integración de esta información mediante un sistema informático.

Para cumplir estos objetivos, es necesaria, en primer lugar, la instalación del hardware adecuado. En lo que a hardware se refiere, se necesitan antenas capaces de recibir señales de las aeronaves, que sean decodificables posteriormente mediante una aplicación. Estas antenas deben ser capaces de obtener únicamente señales a una frecuencia determinada, ignorando las indeseadas, y deben estar instaladas en unos puntos clave, para permitir al sistema en su conjunto tener cobertura en toda la zona que se pretende cubrir. Los puntos clave mencionados estarían localizados en Valencia, Barcelona y Madrid en primera instancia, para después completar el sistema situando antenas en algunos puntos adicionales, si fuera necesario para cumplir con los objetivos de cobertura.

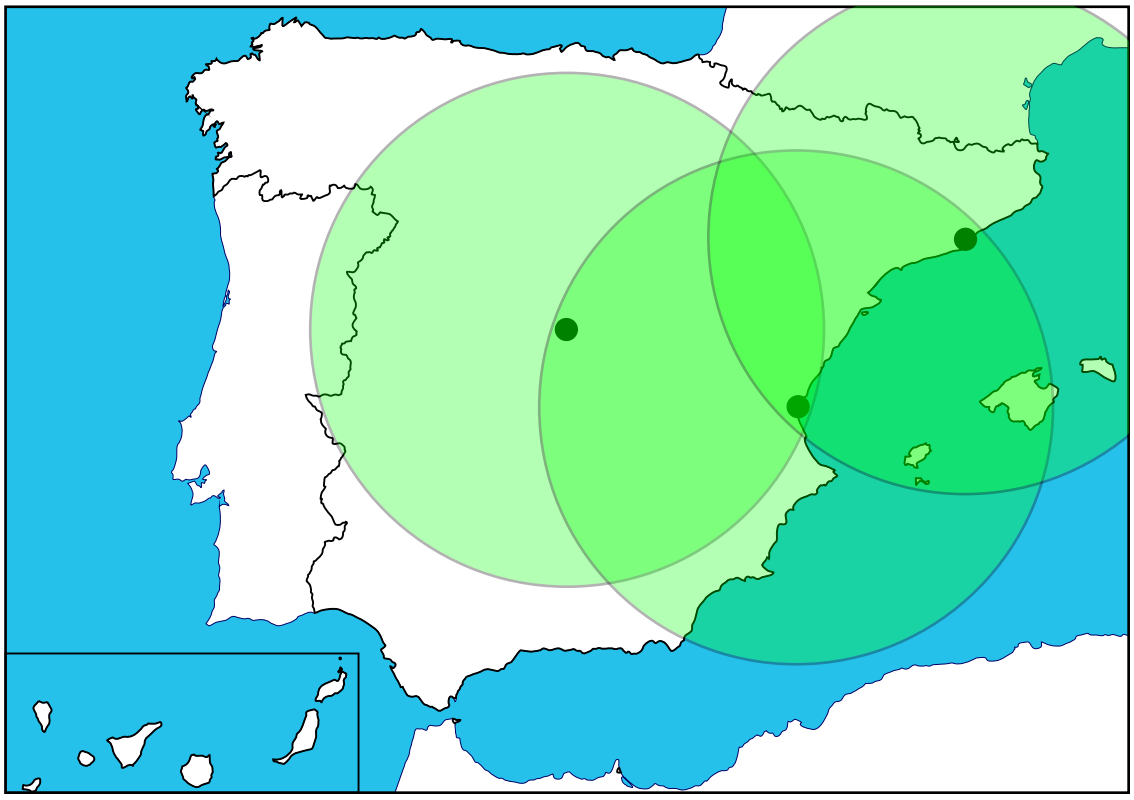


Figura 1.1: Representación gráfica aproximada de la cobertura del sistema completo con estaciones en Valencia, Barcelona y Madrid.

Por otro lado, se necesita que las señales recibidas por las antenas sean convertidas a información digital mediante un dispositivo que actúe como traductor y emitidas al exterior después de la conver-

sión. La información enviada desde estos equipos de hardware a la estación integradora de datos debe tener un formato estandarizado y transmitirse por un protocolo bien definido. Existe una normativa que define más de un formato para la información de vigilancia que se desea obtener, por lo que es necesario escoger uno de ellos, de acuerdo a ciertos criterios, y utilizar solamente el formato escogido en todo el sistema. Con la definición de un único formato y tipo de protocolo se consigue un sistema escalable, pudiendo ser ampliado para obtener información de una mayor cantidad de antenas sin necesidad de realizar cambios profundos en el diseño.

En lo que se refiere a la aplicación encargada de integrar y decodificar la información recibida por las antenas, ésta debe estar programada en un lenguaje adecuado a las necesidades del sistema y los objetivos, facilitando éste la programación de los distintos procesos mediante los recursos proporcionados. Debe tener un código robusto, optimizado y claramente organizado, para tener la capacidad de detectar errores de cualquier naturaleza y ser ampliado con facilidad si fuera necesario. Esto aporta integridad al sistema completo, evitando la transmisión de información errónea a los clientes. Además, es muy importante la resolución de ambigüedades en la recepción de la información, pues es inevitable la recepción de datos idénticos de la misma aeronave por parte de más de una estación receptora.

El servidor encargado de almacenar y transmitir los datos de tráfico debe ser capaz de proporcionar éstos a los clientes que lo deseen, contando con la capacidad de transmisión selectiva, esto es, proporcionar información específica de acuerdo al tipo de petición efectuada por el cliente. Existirían otras características configurables en lo que a transmisión de información se refiere, como puede ser la frecuencia de envío. Al igual que en el subsistema formado por las estaciones receptoras y la estación integradora, la información debe ser enviada a los clientes por un protocolo y un formato único. A diferencia de aquel subsistema, en este caso el formato de la información debe ser totalmente definido desde cero, sin la aplicación de ningún tipo de normativa. El formato en cuestión debe ser claro y, a su vez, estar bien optimizado.

El servicio ofrecido por el sistema tiene una amplia variedad de posibilidades de utilización. Las aplicaciones que lo pueden usar abarcan desde las relacionadas con la investigación de cualquier aspecto relativo al tráfico aéreo, como los flujos de tráfico o incidencias, hasta aplicaciones móviles con información para usuarios del sistema de navegación aérea. Por otro lado, el sistema está también orientado a la docencia, permitiendo ofrecer, por ejemplo, unas prácticas de laboratorio de calidad en las asignaturas orientadas al estudio del funcionamiento del tráfico y la navegación aérea.

Por último, es muy importante señalar que todo lo expuesto en las partes II (Diseño del sistema) y III (Implementación Java) , especialmente lo relativo al diseño de la aplicación decodificadora/integradora y el servidor, corresponde a la última versión del proyecto en funcionamiento en el momento de la redacción de esta memoria. El proyecto en cuestión es de una magnitud considerable y el diseño de éste se irá haciendo más completo y complejo en futuras versiones, con el objetivo de alcanzar todos los objetivos propuestos de la mejor manera posible.

Parte I

El Modo S

2. Fundamentos del Modo S

2.1. El radar secundario

El radar secundario (*SSR*[1]) es un sistema aeronáutico de vigilancia, utilizado en el control de tráfico aéreo. Se caracteriza por ser un sistema que tiene como partes fundamentales una estación interrogadora, normalmente localizada en tierra, y un transpondedor, ubicado en la aeronave. La estación de tierra es capaz, entonces, en enviar interrogaciones a la aeronave en forma de señales codificadas de manera específica, a las que el transpondedor envía una respuesta de acuerdo al tipo de interrogación. Las interrogaciones se realizan a una frecuencia de 1030 *MHz* y las respuestas, a 1090 *MHz*.

El Modo S es uno de los modos de interrogación/respuesta del *SSR*. El modo en cuestión es el que se analizará en profundidad a lo largo de esta parte del documento. Sin embargo, antes de pasar a ello, es fundamental introducir los conceptos de Modo A y Modo C, los otros modos de funcionamiento del *SSR*:

- **Modo A**: las interrogaciones del Modo A producen, por parte del transpondedor, respuestas que contienen un código de 4 dígitos octales para identificar a la aeronave. El número total de códigos posibles es 4096, en un rango de 0000 a 7777, y hay unos códigos reservados para ciertos propósitos: emergencia (7700), fallo de la radio (7600) y secuestro (7500). Las direcciones se encuentran codificadas en las respuestas mediante una serie de pulsos. El contenido de estas señales se analiza y se decodifica en una estación receptora. Las interrogaciones Modo A no son selectivas, esto es, van dirigidas a todas las aeronaves que se encuentran en la zona de cobertura de la antena.
- **Modo C**: las interrogaciones del Modo C producen respuestas que contienen, codificada, la altitud barométrica de la aeronave. La presión utilizada para establecer la altitud está referenciada a una presión estándar de 1013.25 hectopascales. La resolución de la altitud codificada es de 100 pies y se permiten 2048 valores de entre -1000 pies y 160.000 pies. Las respuestas del Modo C están codificadas de manera similar a las del Modo A, mediante una serie de pulsos decodificables. Las interrogaciones del Modo C, al igual que las del Modo A, no son selectivas.

2.2. Orígenes y beneficios del Modo S

El **Modo S** se caracteriza por permitir la interrogación selectiva de una estación en tierra a una aeronave concreta y la posibilidad de transmisión de una amplia variedad de información.

Este modo surge, fundamentalmente, debido a la incapacidad de los sistemas de vigilancia Modo A/C de hacer frente a problemas relacionados con la transmisión de señales (interferencia, reflexión y solapamiento), además de la continua evolución y aumento del tráfico aéreo, que hace que las 4096 direcciones de identificación disponibles en el Modo A sean insuficientes.

El Modo S mitiga gran parte de los problemas de transmisión debido a la selectividad de la interrogación y la consecuente creación de un enlace de datos entre tierra y aeronave. Por otro lado, en

[1] Secondary Surveillance Radar.

este modo se le asigna una dirección *ICAO* de 24 *bits* única a cada avión, lo que soluciona el problema de la saturación de direcciones, al proporcionar un total de 16.777.216 direcciones posibles.

2.3. Interoperabilidad entre modos A/C/S

Este modo es completamente compatible con los modos A y C, por lo que una aeronave equipada con un transpondedor Modo S es capaz de funcionar en los modos mencionados anteriormente, a las mismas frecuencias de funcionamiento: 1030 *MHz* para interrogación y 1090 *MHz* para respuesta. Esto permite la interrogación por parte de una estación de tierra Modo S a aeronaves equipadas con transpondedores con capacidad para todos los modos (A/C/S), siendo la respuesta de éstas dependiente de la capacidad del transpondedor que utilicen y del tipo de interrogación. Así mismo, un transpondedor Modo S debe ser capaz de responder a una estación con capacidad para Modo A/C con la información correspondiente.

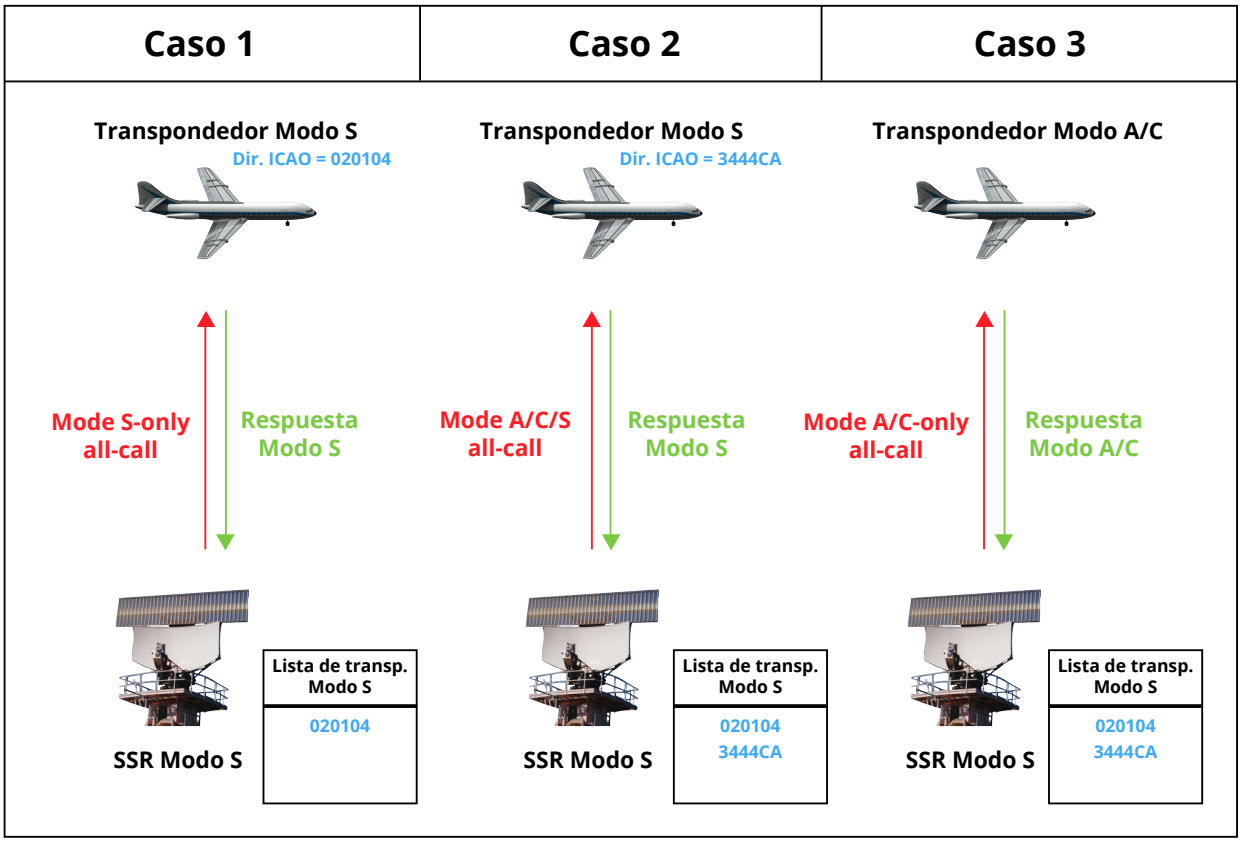


Figura 2.1: Tres casos de interrogaciones del SSR Modo S con el objetivo de identificar los transpondedores Modo S.

Las interrogaciones emitidas por un *SSR* Modo S que tienen como objetivo identificar los diferentes tipos de transpondedores son las siguientes:

- **Interrogación Mode A/C/S all-call**: se trata de una interrogación para obtener respuestas de vigilancia de los transpondedores Modo A/C e identificar los que tienen capacidad para Modo S.
- **Interrogación Mode A/C-only all-call**: es como la interrogación anterior, pero los transpondedores Modo S no responden.
- **Interrogación Mode S-only all-call**: solamente responden los transpondedores Modo S.

Tras ser identificadas las aeronaves con transpondedores Modo S, éstas se añaden a una lista y se establece una condición de bloqueo (*lockout*) en la estación de tierra para las aeronaves en cuestión, con el objetivo de dejar de recibir respuestas a interrogaciones *All-Call* de ellas.

Este comportamiento se resume en la Figura 2.1, donde se muestran tres casos de interrogación de un *SSR* Modo S a diferentes aeronaves.

2.4. La vigilancia en Modo S

2.4.1. Transacciones en Modo S

El envío de un mensaje concreto en Modo S se conoce como transacción. Las transacciones se pueden clasificar en varios grupos, dependiendo de la función del mensaje, sus protocolos de comunicación y la estructura de datos. Estos grupos son los siguientes:

- **Transacciones intermodo y Modo S-only all-call**: corresponden a las interrogaciones por parte de estaciones de tierra y las correspondientes respuestas de los transpondedores que tienen como objetivo identificar el tipo de transpondedor de las aeronaves en cobertura, esto es, si solamente tienen capacidad para Modo A/C o también para Modo S. Las respuestas a estas interrogaciones se conocen como *All-Call Reply*. Las interrogaciones correspondientes a estas transacciones son las definidas en 2.3.

- **Transacciones de vigilancia**: se refieren a los mensajes de interrogación y respuesta con el objetivo de obtener información correspondiente al Modo A/C. Estos mensajes contienen datos de vigilancia e información sobre control de comunicación y en la respuesta se transmite el código de identidad Modo A o el de altitud Modo C de la aeronave.

- **Transacciones de comunicación de longitud estándar**: se refieren al intercambio de mensajes que se conocen como *SLMs* (*Standard Length Messages*) mediante un enlace de datos entre la estación de tierra y la aeronave. La información a intercambiar se obtiene de los registros del transpondedor, conteniendo cada uno de ellos una información concreta. La interrogación correspondiente a esta transacción se conoce como *Comm-A* y la respuesta, *Comm-B*. Esta transacción permite la concatenación de hasta cuatro segmentos *Comm-A* o *Comm-B* unidos. Un conjunto de este tipo se conoce como *frame*. La respuesta *Comm-B* puede ser generada de distintas formas:

  - Mediante el protocolo **GICB (Ground Initiated Comm-B)**, que permite a la estación de tierra especificar qué información de los registros del transpondedor se pretende obtener, mediante un código *BDS* (*Comm-B Data Selector*) de 8 *bits*.
  - Mediante el protocolo **AICB (Air Initiated Comm-B)**, que permite a la aeronave especificar que tiene un mensaje *Comm-B* listo para ser transmitido.
  - Mediante el protocolo **Comm-B Broadcast**, que permite la transmisión de un mensaje *Comm-B* por difusión, esto es, a más de una estación.

- **Transacciones de comunicación de longitud extendida**: se refieren al intercambio de mensajes conocidos como *ELMs* (*Extended Length Messages*). Como su nombre indica, son mensajes que tienen una longitud superior para su "carga de pago"que los *SLM*. La interrogación se realiza por un protocolo conocido como *Comm-C* y la respuesta, por el *Comm-D*.

- **Transacciones aire − aire**: corresponden al intercambio de información referente al sistema *ACAS (Airborne Collision Avoidance System)*[2].

[2]Sistema de la aeronave encargado de detectar y ayudar a resolver conflictos con otras aeronaves cuando éstos se producen.

- **Transacciones squitter**: no son transacciones propiamente dichas, pues son transmisiones por difusión. Los *squitters* son mensajes emitidos espontáneamente, de manera periódica, por parte del transpondedor y sin interrogación externa, aunque algunos servicios especiales explicados en el apartado siguiente los transmiten de forma distinta. Dos son los tipos de *squitters* existentes en Modo S:
  - **Acquisition Squitter**: se transmiten con el mismo formato que los *All-Call Reply*. Se utilizan para dar soporte a *ACAS* y al sistema de vigilancia terrestre en los aeropuertos, entre otros.
  - **Extended Squitter (ES)**: se transmiten con un formato específico y el contenido del mensaje se obtiene de los registros del transpondedor. Existen varios tipos de *Extended Squitter*, siendo cada uno de ellos identificable mediante lo que se conoce como *Type Code (TC)*.

2.4.2. Técnicas que utilizan el Modo S

El Modo S es utilizado para las técnicas *ADS-B (Automatic Dependent System – Broadcast)* y *TIS-B (Traffic Information Service – Broadcast)*. Por un lado, *ADS-B* utiliza sistemas embarcados (por ejemplo, aquellos con capacidad *GNSS*[3]) para transmitir información como puede ser la posición o la velocidad. Los sistemas con capacidad de transmisión de esta información se conocen como *ADS-B OUT* y los que tienen capacidad de recepción de la misma, *ADS-B IN*.

Esta información se puede transmitir mediante dos tipos de enlaces o protocolos. Uno de ellos, el más extendido, es el *Extended Squitter* (de hecho, *ES* se utiliza exclusivamente para el funcionamiento del sistema *ADS-B*). El otro, conocido como como *UAT (Universal Access Transciever)*, funciona a 978 *MHz* y es un enlace que ha sido introducido en los Estados Unidos.

Por otro lado, la técnica *TIS-B* consiste en la transmisión de información radar (de estaciones Modo S, *MLAT*[4], etc.) por parte de estaciones de tierra a aeronaves con capacidad *ADS-B IN*. Permite a las aeronaves en cuestión visualizar la información disponible para un controlador aéreo y así poder localizar su posición en el espacio aéreo junto a las demás aeronaves en todo momento.

Existe también una técnica adicional conocida como *ADS-R (Rebroadcast)*, que busca alcanzar la interoperabilidad entre sistemas *ADS-B* que utilizan *data-links* distintos, concretamente los mencionados *ES* y *UAT*. Esto se consigue mediante el formateo de un mensaje *UAT* en una estación terrestre a uno *ES* y su transmisión a 1090 *MHz*.

[3]Global Navigation Satellite System.

[4]Multilateración, una técnica de vigilancia basada en la medida de diferencias de tiempos de transmisión.

3. Normativa ICAO

Existe una normativa *ICAO (International Civil Aviation Organization)* que explica cómo se realiza la codificación de la información en Modo S. La finalidad y objetivos de este trabajo exigen un estudio en profundidad de esta normativa, establecida fundamentalmente en los siguientes documentos:

- *Doc 9871, "Technical provisions for Mode S services and Extended Squitter"*: en este documento se especifican las provisiones técnicas para los formatos y protocolos del Modo S y *Extended Squitter*, además de directrices para la implementación e información sobre futuros servicios de este modo que se encuentran en fase de desarrollo.
- *Anexo 10, "Aeronautical Telecommunications" – Volumen IV, "Surveillance and Collision Avoidance Systems"*. Se trata de un documento más general que define estándares y recomendaciones para el *SSR* y *ACAS*.

Se ha utilizado también el siguiente documento para completar información insuficiente acerca de algunos conceptos explicados en los documentos anteriores:

- *RTCA DO-260B, "Minimum Operational Performance Standards for 1090 MHz Extended Squitter Automatic Dependent Surveillance – Broadcast (ADS-B) and Traffic Information Services – Broadcast (TIS-B)"*. Se trata de un análogo al Anexo 10 de la *ICAO*, pero tratando solamente los estándares para *ADS-B* y *TIS-B Extended Squitter*. Fue publicado por la *RTCA (Radio Technical Commission for Aeronautics)* estadounidense.

3.1. Los niveles de transpondedor

La *ICAO* establece una normativa a cumplir por los transpondedores Modo S, clasificándolos en diferentes niveles de acuerdo a sus capacidades, parte de ellas ya introducidas anteriormente:

- **Nivel 1**. Las capacidades de estos transpondedores deben ser las siguientes:
  - Transmisión de información de Modo A/C.
  - Transacciones intermodo/*Mode S-only all-call*.
  - Transmisión de información de Modo A/C mediante formato de Modo S.
  - Protocolos de bloqueo (al adquirir la dirección *ICAO* de un transpondedor Modo S).
  - Protocolos básicos de datos, menos el *Data Link Capability Report* (se explicarán más adelante).
  - Transacciones aire-aire y de *squitters*.
- **Nivel 2**. A las capacidades del nivel 1 se le añaden las siguientes:
  - Transacciones de *SLMs*.
  - Transmisión del *Data Link Capability Report*.
  - Transmisión de la identificación de la aeronave.
  - Transmisión de información de paridad (se explicará más adelante).

- **Niveles 3, 4 y 5**. Añaden las capacidades para transacciones *ELM* y mejoran el protocolo *Comm-B*.

Según la normativa, todos los transpondedores utilizados en aviación civil internacional deben ser de, al menos, nivel 2.

3.2. Los registros del transpondedor

En lo referente al segmento aire, el transpondedor Modo S es el encargado de recibir interrogaciones, procesarlas y enviar las respuestas correspondientes. También se ha visto que es capaz de difundir información sin necesidad de interrogación, ya sea mediante *squitters* o enviando un *Comm-B broadcast*.

La información que el transpondedor es capaz de transmitir está alojada en una serie de registros del mismo. En total son 255 registros, con 56 *bits* de información cada uno. Los datos almacenados en éstos son los que se introducen en unos campos definidos para cada mensaje. Cada registro tiene asignado un código, el *BDS (Comm-B Data Selector)*, que permite a una solicitud *GICB*, por ejemplo, extraer la información correspondiente para transmitirla en la respuesta. El *BDS* está formado por dos grupos de 4 *bits*.

Figura 3.1: Representación gráfica de dos ejemplos de petición y transmisión de información de los registros.

3.2.1. Requisitos para la validez de la información

Aunque la información de los registros debe ser actualizada inmediatamente después de obtener nueva información, cada registro tiene asignada una tasa mínima de actualización. En caso de que un registro no se actualice en un periodo del doble de esta tasa mínima o dos segundos (el tiempo mayor

de los dos), se considera que el contenido del registro es inválido y un *flag* de estado pasa a indicar esta condición.

En lo referente a la validez de la información de gran parte de los registros (menos algunos concretos que no necesitan verificar su validez), ésta depende de los valores siguientes:

- Un *bit* del registro correspondiente al informe de capacidad de enlace de datos (*Data Link Capability Report*). La información de este registro es de una gran importancia, pues indica las capacidades del transpondedor referentes a los protocolos *Comm-A*, *Comm-B*, *ELM* y *ACAS*. Esta información es generalmente adquirida una vez se adquiere la dirección *ICAO* de la aeronave por una llamada *All-Call* o *Mode S-only All-Call* y se va actualizando en el transpondedor una vez por segundo, lanzando un mensaje por el protocolo de difusión *Comm-B* en caso de que se haya actualizado la información del registro en cuestión.
- Un *bit* que indica el soporte al servicio *GICB*, que se encuentra en unos registros determinados.
- Un *bit* de validez que indica si la información desde ese *bit* hasta el *bit* de validez siguiente es válida. Este es el *flag* mencionado anteriormente, que salta cuando la tasa de actualización es demasiado lenta.

3.3. Los mensajes Modo S

Ya se introdujeron anteriormente los distintos tipos de transacciones existentes en el Modo S. Se ha visto que cada tipo transacción tiene asociados una serie de mensajes. La *ICAO* define las características de cada uno de estos mensajes.

Una de esas características es el identificador asignado a cada uno de ellos, ya sea en interrogación o en recepción. Este identificador es único para cada mensaje y recibe el nombre de *Uplink Format (UF)* para las interrogaciones y *Downlink Format (DF)* para las respuestas.

A continuación se presenta una tabla con los identificadores correspondientes a cada mensaje, además de otra característica, la longitud del mensaje. La característica más compleja, esto es, la estructura de cada mensaje, se abordará en un capítulo posterior:

UF	DF	Interrogación	Respuesta	Longitud (bits)
0	0	Short ACAS Interrogation	Short ACAS Reply	56
4	4	Altitude Request	Altitude Reply	56
5	5	Identity Request	Identity Reply	56
11	11	Mode S-Only All-Call Interrogation	All-Call Reply	56
16	16	Long ACAS Interrogation	Long ACAS Reply	112
X	17	X	Extended Squitter	112
X	18	X	Extended Squitter	112
20	20	Comm-A Altitude Request	Comm-B Altitude Reply	112
21	21	Comm-A Identity Request	Comm-B Identity Reply	112
24	24	Comm-C	Comm-D	112

Tabla 3.1: Mensajes del Modo S

Nota 1: El valor DF = 18 se utiliza en los sistemas distintos a un transpondedor que transmiten mensajes ES, esto es, por ejemplo, estaciones de tierra. Por ello, es un formato que usa la técnica TIS-B.

Nota 2: Aunque cada mensaje Comm-A o Comm-B tiene una longitud de 112 bits (de los cuales 56 bits corresponden a información extraída de los registros), son mensajes que se pueden concatenar, como ya se vio, por lo que la longitud total puede variar, hasta alcanzar los 224 bits de información de los registros.

Nota 3: Como también se vio, se permite el envío de grupos de hasta 16 mensajes Comm-C/Comm-D, por lo que se puede realizar una transmisión de hasta 1280 bits de información de los registros.

3.3.1. Los mensajes Extended Squitter

Actualmente, existen dos versiones diferentes de los mensajes *Extended Squitter*: Versión 0 y 1. La Versión 1 introduce mejoras en los indicativos de precisión e integridad de los mensajes de posición y velocidad, además de otros parámetros adicionales y los formatos para *TIS-B* y *ADS-R*. Ambas versiones son compatibles entre sí, por lo que un receptor de una versión concreta debe ser capaz de procesar y decodificar los mensajes de la otra versión.

3.3.1.1. Extended Squitter Versión 0

Los mensajes *Extended Squitter*, que corresponden con los *DF* 17 y 18, se clasifican de acuerdo al valor de unos *bits* contenidos en los mismos mensajes que reciben el nombre de *Type Code (TC)*. En total tenemos 33 *TCs* (de 0 a 32) y, en Versión 0, cada uno especifica lo siguiente:

- **Contenido del mensaje** (información alojada en unos registros determinados), que puede ser:
  - Identificación y categoría de la aeronave (*Aicraft identification and category*)
  - Posición de superficie (*Surface position*)
  - Posición aérea (*Airborne position*)
  - Velocidad aérea (*Airborne velocity*)
  - Estado de prioridad/emergencia de la aeronave (*Emergency/priority status*)
  - Estado operacional de la aeronave (*Aicraft operational status*)
  - Reservado para otros propósitos (*Reserved*)

  Todos estos mensajes y su estructura se analizarán posteriormente. Cabe mencionar que el *TC* 0 se utiliza para indicar que no hay información de posición disponible.
- **El límite de protección horizontal (*HPL*)** para los mensajes de posición, que corresponde al radio de un círculo centrado en la aeronave que define un área en la que se asegura que se va a encontrar la misma.
- **El radio de contención del error** horizontal y/o vertical con una probabilidad del 95 %, también para los mensajes de posición. El vertical sólo está disponible cuando la fuente de altura/altitud es *GNSS*.
- **Tipo de altitud/altura**. Puede darse que:
  - No haya información de altitud/altura
  - Se transmita la altitud barométrica
  - Se transmita la altura *GNSS* (Altura sobre el elipsoide o *HAE*[1]).
  - En el mensaje de velocidad, se transmite la diferencia entre la altitud barométrica y la altura *GNSS (HAE)* o altitud *GNSS (MSL*[2]*)*.

[1] Height Above Ellipsoid.
[2] Mean Sea Level.

- **El NUC (Navigation Uncertainty Category)** para los mensajes, que es un indicativo de calidad, que se utiliza tanto para la información de posición como para la velocidad. Este indicativo no especifica si se refiere a la precisión o a la integridad de de la información, puede estar referido a cualquiera de las dos características.

En condiciones normales de funcionamiento, la fuente de información del mensaje a transmitir proporcionará el *HPL* y a partir de este valor se establecerá el *TC*. De todas formas, pueden darse las situaciones siguientes:

- Si la información *HPL* está disponible, entonces se calculará el *TC* del mensaje de posición aérea a partir de éste y la fuente de información de la altitud.
- Si la información *HPL* no está disponible temporalmente, se usarán los radios de contención del error y el tipo de altitud para calcular el *TC* del mensaje de posición aérea.
- Si la información de posición está disponible pero la información de precisión/integridad no lo está, se utilizan los *TC* correspondientes a los máximos valores de *HPL* para cada tipo de mensaje.

3.3.1.2. Cambios en Extended Squitter Versión 1

Los cambios más relevantes que introduce la Versión 1 se pueden resumir como sigue:

- El mensaje *Emergency/Priority Status* pasa a ser *Aircraft Status* y la información del registro correspondiente cambia. Este nuevo mensaje tiene subtipos, el mismo *Emergency/Priority Status* y uno nuevo, el *Extended Squitter ACAS RA Broadcast*. Es decir, aparece un nuevo tipo de *ES*.
- Las asignaciones de *TC* cambian. El *HPL*, los radios de contención del error y el *NUC* se sustituyen por:
  - El **radio de contención horizontal (Rc)**, análogo al *HPL*. Se puede obtener del mensaje de posición en caso de que la fuente de ésta sea *GNSS*. Se conoce también como *HIL* (Límite de integridad horizontal).
  - El **límite de protección vertical (VPL)** junto con el *Rc*.
  - El **NIC (Navigation Integrity Category)** es un indicativo de integridad de la información que define el *TC* de cada mensaje de posición (con sus correspondientes valores de *Rc* y *VPL*) y un código suplementario *NIC (NIC Supplement)* que se transmite en el mensaje *Aircraft Operational Status ES*. La zona definida por *Rc* y *VPL* se conoce, entonces, como una región de contención de integridad.
  - El **NAC (Navigation Accuracy Category**), tanto para la posición como para la velocidad, un indicativo de precisión. Se utilizan lo que se conocen las figuras de mérito horizontal y vertical, *HFOM* y *VFOM*, para definir los radios de contención del error con una confianza del 95 %. También se conocen como *EPU (Estimated Position Uncertainty)* y *VEPU (Vertical EPU)* para el caso de la posición, si no existe fuente *GNSS* para estimar ésta.
  - El **SIL (Surveillance Integrity Level)**, que describe la probabilidad de exceder la región de contención de integridad descrita por el *NIC*.

Estos y otros cambios adicionales se analizarán en detalle en la sección siguiente. A continuación se presenta una tabla de los mensajes *Extended Squitter* en *V0* y *V1*, con los *TC* asociados a cada tipo de mensaje. Se indica en qué mensajes existe diferencia en cuanto a formato entre *ES V0* y *V1* en la columna correspondiente a los tipos de mensajes. Estas diferencias se analizarán en la sección siguiente.

TC	Mensaje	Tipo de altitud/altura	Cambio de formato entre V0 y V1
0	No hay posición	Altitud baro No hay info. de posición	No
1 - 4	Identification ES	X	No
5 - 8	Surface Position ES	No hay info. de altitud	Sí
9 - 18	Airborne Position ES	Altitud barométrica	No
19	Airborne Velocity ES	Diferencia entre Altitud baro. y altitud/altura GNSS	Sí
20 - 22	Airborne Position ES	Altura GNSS (HAE)	No
23 - 27	Reservado	X	X
28	V0: Emergency/Priority Status ES V1: Emergency/Priority Status o ACAS RA Broadcast	X	Sí
29 - 30	Reservado	X	X
31	Aircraft Operational Status	X	Sí

Tabla 3.2: Mensajes Extended Squitter V0/V1

Nota 1: En la V0, el valor de NUC está reflejado en el TC. En la V1, sólo el NIC está reflejado en el TC, mientras que los demás indicativos (NAC y SIL) se transmiten en otros mensajes, como se verá posteriormente.

Nota 2: Los valores de NUC van de 0 a 9, siendo 9 el valor correspondiente al mejor nivel. Los de NIC tienen un rango de 0 a 11, siendo 11 el mejor valor.

Nota 3: Para los mensajes de posición, el NUC (V0), el NIC(V1) y los radios de contención horizontal y/o vertical empeoran conforme aumenta el valor de TC.

3.4. Estructura y decodificación

Se va a pasar a analizar en detalle la estructura de cada uno de los tipos de mensaje Modo S. En la sección 2.4.1 se definieron los tipos de transacciones existentes en este modo de vigilancia, por lo que se seguirá esa misma clasificación para agrupar los distintos tipos de mensajes.

Teniendo en cuenta los objetivos del presente trabajo, se analizará solamente la estructura de los mensajes de respuesta. Además, en lo que se refiere a los mensajes enviados por el protocolo *Comm-B*, debido a la enorme cantidad de información procedente de los registros del transpondedor que se puede insertar en estos mensajes y teniendo en cuenta que una gran parte de ellos se transmite mediante un enlace de datos entre una estación de tierra y la aeronave, escapa a los objetivos del proyecto (al menos, en la versión actual del mismo) el análisis de la "carga de pago"de estos mensajes en concreto. Es decir, en lo referente a estos mensajes, se analizará toda la estructura menos esa "carga de pago", que corresponde a la información extraíble de los registros mediante el *BDS*.

En primer lugar, se analizarán los campos generales, esto es, aquellos que están presentes en todos los mensajes Modo S. Después, para cada uno de los mensajes analizados, se realizará una representación gráfica de la estructura del mismo, en la que se indicarán los campos contenidos en cada rango de *bits*. Después, se procederá a explicar el significado de cada uno de los campos específicos y cómo se codifica la información dentro de ellos.

3.4.1. Formato general de los mensajes Modo S

Campo	Bits
Downlink Format (DF) = 17	5
Payload	27/83
Parity	24

Tabla 3.3: Estructura del mensaje Modo S

3.4.1.1. Downlink Format (DF)

Es un código que indica qué tipo de mensaje se está transmitiendo, de todos los mostrados en la tabla 3.16.

3.4.1.2. Payload

Contiene el mensaje específico correspondiente al *DF*, es decir, son bits del mensaje variables de acuerdo a ese valor. Algunos mensajes almacenan aquí, en un subcampo, la dirección *ICAO* de 24 *bits* de la aeronave.

3.4.1.3. Parity

Se trata de un campo que se utiliza para comprobación de errores. En la recepción de un mensaje Modo S se debe realizar lo que se conoce como cálculo de paridad a partir de la información del mensaje en cuestión y, mediante una comparación con el campo de paridad, determinar si la secuencia de *bits* es consistente con la estructura del código.

Este campo recibe nombres diferentes dependiendo del mensaje: *PI (Parity/Identification)* y *AP (Address/Parity)*. En concreto, depende de cuál es la información que se combina con unos bits de comprobación de paridad. El cálculo de paridad es un proceso algo complejo, pero existe un método definido por Eurocontrol para realizar el mismo, en el documento *"CRC Calculation for Mode-S Transponders"*. Cabe destacar que:

- Para las respuestas *All-Call Reply*, en las que el campo de paridad es el *PI*, del campo en cuestión se puede extraer la *ID* (código de identificación) del interrogador.
- Para los mensajes *Extended Squitter*, el campo de paridad también es *PI*, y se utiliza para detectar mensajes erróneos, mediante lo que se conoce como *CRC (Cyclic Redundancy Check)*.
- Para las respuestas Modo S que no son *All-Call Reply* y *Extended Squitter*, este campo es el *AP*, y se utiliza para extraer la dirección *ICAO* de la aeronave.

3.4.2. Transacciones intermodo y Mode S-only all-call

3.4.2.1. Formato del mensaje All-Call Reply

Campo	Bits
Downlink Format (DF) = 11	5
Capability (CA)	3
Address Announced (AA)	24
Parity	24

Tabla 3.4: Estructura de All-Call Reply

3.4.2.1.1 Downlink Format (DF)

Ver 3.4.1.1.

3.4.2.1.2 Capability (CA)

Campo que contiene información acerca del nivel de transpondedor (los distintos niveles ya fueron especificados en 3.1).

3.4.2.1.3 Address Announced (AA)

Es la dirección *ICAO* que identifica a la aeronave.

3.4.2.1.4 Parity/Identification (PI)

Ver 3.4.1.3

3.4.3. Transacciones de vigilancia

3.4.3.1. Formato del mensaje Altitude Reply

Campo	Bits
Downlink Format (DF) = 11	5
Flight Status (FS)	3
Downlink Request (DR)	5
Utility Message (UM)	6
Altitude Code (AC)	13
Address/Parity (AP)	24

Tabla 3.5: Estructura de Altitude Reply

3.4.3.1.1 Downlink Format (DF)

Ver 3.4.1.1.

3.4.3.1.2 Flight Status

Indica la existencia de un estado de alerta en la aeronave por algún motivo, si la aeronave está en tierra o en el aire (o se desconoce) y si la condición *SPI (Special Position Identifier)* está activa. La condición *SPI* indica el envío de un pulso adicional junto al mensaje *Altitude Reply* y se utiliza cuando hay conflicto de respuestas entre aeronaves. Los distintos valores de *FS* indican diferentes combinaciones para toda esta información.

3.4.3.1.3 Downlink Request (DR)

Indica si la aeronave ha generado una petición para alguna información. Este campo puede referirse a una petición para transmitir un mensaje *Comm-B* listo para ser enviado (ya sea por enlace de datos o por difusión), una petición relacionada con *ACAS* o con un *ELM*.

3.4.3.1.4 Utility Message (UM)

Contiene información relacionada con la comunicación del transpondedor. Esta información consta de un código de identificación de interrogador (no es el mismo código que el utilizado en las transacciones intermodo/*Mode S only-call*) y un código que indica el tipo de reserva realizada por el interrogador en cuestión[3].

3.4.3.1.5 Altitude Code (AC)

Es el código que contiene la altitud de la aeronave. Contiene unos campos que indican la unidad de medida de la altitud (metros o pies) y, si la unidad definida son pies, también se indica la resolución de la altitud (25 o 100 pies). La resolución de 100 pies corresponde a los mensajes de altitud Modo C. En el caso de una resolución de 25 pies, la codificación de la altitud se realiza de manera distinta a la correspondiente al Modo C.

[3]Este campo tiene utilidad en lo que se conoce como *Multi-Site Communications*, concepto que escapa a los objetivos de este apartado.

3.4.3.1.6 Address/Parity (AP)

Ver 3.4.1.3

3.4.3.2. Formato del mensaje Identity Reply

Campo	Bits
Downlink Format (DF) = 11	5
Flight Status (FS)	3
Downlink Request (DR)	5
Utility Message (UM)	6
Identity (ID)	13
Address/Parity (AP)	24

Tabla 3.6: Estructura de Identity Reply

3.4.3.2.1 Downlink Format (DF)

Ver 3.4.1.1.

3.4.3.2.2 Flight Status

Ver 3.4.3.1.2

3.4.3.2.3 Downlink Request (DR)

Ver 3.4.3.1.3

3.4.3.2.4 Utility Message (UM)

Ver 3.4.3.1.4

3.4.3.2.5 Identity (ID)

Es el código que contiene la identidad Modo A de la aeronave, codificada por el mismo método que en los mensajes del modo en cuestión.

3.4.3.2.6 Address/Parity (AP)

Ver 3.4.1.3

3.4.4. Transacciones de longitud estándar

3.4.4.1. Formato del mensaje Comm-B

Los mensajes de respuesta Comm-B se pueden clasificar en dos tipos:

- **Comm-B Altitude Reply**: tienen los mismos campos que un mensaje de vigilancia *Altitude Reply*, pero añadiendo 56 *bits* de *payload* (información obtenida de los registros).
- **Comm-B Identity Reply**: tienen los mismos campos que un mensaje de vigilancia *Identity Reply*, pero añadiendo 56 *bits* de *payload*.

De acuerdo a lo expuesto, la estructura de ambos mensajes es la siguiente:

Campo	**Bits**
Downlink Format (DF) = 20/21	5
Flight Status (FS)	3
Downlink Request (DR)	5
Utility Message (UM)	6
Altitude Code (AC)/Identity (ID)	13
Payload	56
Address/Parity (AP)	24

Tabla 3.7: Estructura de Comm-B Altitude/Identity Reply

3.4.5. Transacciones de longitud extendida

3.4.5.1. Formato del mensaje Comm-D

Campo	Bits
Downlink Format (DF) = 24	2
X	1
Control, ELM (KE)	1
Number of D-segment	4
Message, Comm-D	80
Address/Parity (AP)	24

Tabla 3.8: Estructura de Comm-D

3.4.5.1.1 Downlink Format (DF)

Ver 3.4.1.1.

3.4.5.1.2 Control, ELM (KE)

Este campo establece el contenido de los campos siguientes. Define si se trata de una transmisión *ELM downlink* o si por el contrario del reconocimiento de un *ELM uplink*.

3.4.5.1.3 Number of D-segment (ND)

Como se vio, los mensajes *ELM* se pueden agrupar en hasta 16 segmentos. Este campo indica el número de segmento de la información contenida en el campo siguiente.

3.4.5.1.4 Message Comm-D (MD)

Este campo contiene o bien uno de los segmentos de la secuencia *ELM* transmitida o bien códigos de control destinados al interrogador para transmisiones *uplink*.

3.4.5.1.5 Address/Parity (AP)

Ver 3.4.1.3

3.4.6. Transacciones aire - aire

3.4.6.1. Formato del mensaje Short Air - Air

Campo	Bits
Downlink Format (DF) = 0	5
Vertical Status (VS)	1
Cross-Link Capability (CC)	1
X	1
Sensitivity Level (SL), ACAS	3
X	2
Reply Information (RI)	4
X	2
Altitude Code (AC)	13
Address/Parity (AP)	24

Tabla 3.9: Estructura de Short Air - Air

3.4.6.1.1 Downlink Format (DF)

Ver 3.4.1.1.

3.4.6.1.2 Vertical Status (VS)

Indica si la aeronave se encuentra en tierra o en el aire.

3.4.6.1.3 Cross Link Capability (CC)

El mensaje Long Air – Air, que se analizará a continuación, permite transmitir información de los registros del transpondedor de acuerdo una interrogación de tierra (al igual que ocurre con un *Comm-B*), concretamente de acuerdo a lo que se conoce como *Data Selector (DS)*. Esto es un campo que permite seleccionar los registros en los mensajes *uplink*.

La capacidad *CC* indica si el transpondedor es capaz de recibir interrogaciones con *UF = 0*, es decir, las correspondientes a *Short Air – Air* y responder con mensajes *DF = 16*, es decir, *Long Air – Air*, con uno de sus campos conteniendo la información de los registros solicitada mediante el *DS*.

3.4.6.1.4 Sensitivity Level (SL)

Es un nivel correspondiente al sistema *ACAS* de la aeronave que depende de la altitud de la misma y define los niveles de protección alrededor de la misma. Estos niveles de protección se utilizan para determinar cuándo es necesario hacer saltar las alarmas del sistema. Estas alarmas se clasifican en *Traffic Advisory (TA)* o *Resolution Advisory (RA)*.

El *TA* se activa cuando la separación (que puede ser medida como distancia pero también como tiempo al punto de mínima separación) entre la aeronave en cuestión y otra aeronave sobrepasa un límite determinado y es un indicador de conflicto potencial. Esto es, indica que podría producirse

conflicto entre las aeronaves. El *RA* se activa cuando se sobrepasa un límite interno de distancia o tiempo e indica que, si la aeronave no maniobra, se producirá un conflicto.

3.4.6.1.5 Reply Information (RI)

Indica la máxima velocidad verdadera (TAS[4]) de la aeronave o no, dependiendo de un campo del mensaje de interrogación conocido como AQ *(Acquisition)*, que solicita esta información.

3.4.6.1.6 Altitude Code (AC)

Ver 3.4.3.1.5

3.4.6.1.7 Address/Parity (AP)

Ver 3.4.1.3

[4]True Airspeed.

3.4.6.2. Formato del mensaje Long Air - Air

Campo	Bits
Downlink Format (DF) = 0	5
Vertical Status (VS)	1
Cross-Link Capability (CC)	1
X	1
Sensitivity Level (SL), ACAS	3
X	2
Reply Information (RI)	4
X	2
Altitude Code (AC)	13
Message, ACAS (MV)	56
Address/Parity (AP)	24

Tabla 3.10: Estructura de Long Air - Air

Como se puede apreciar en la figura, la estructura es la misma que la de un mensaje *Short Air – Air*, pero añadiendo el campo con la información de los registros solicitada por el interrogador. El campo en cuestión también se utiliza para lo que se conoce como respuestas de coordinación aire – aire (*air - air coordination*). Estas respuestas tienen la siguiente estructura:

Campo	Bits
V-Definition Subfield (VDS)	8
RA Terminated Indicator (RAT)	1
Active RAs (ARA)	14
Resolution Advisory Complement (RAC)	4
Multiple Threat Encounter (MTE)	1

Tabla 3.11: Sub-estructura de Air - Air Coordination (sólo bits MV)

3.4.6.2.1 V-Definition Subfield (VDS)

Define el contenido de los siguientes campos del mensaje. Se trata de una pareja de grupos de 4 *bits*, *VDS1* y *VDS2*. En caso de que *VDS1 = 3* y *VDS2 = 0*, el mensaje contendrá los campos definidos a continuación. Si no tiene estos valores, significa que la información transmitida no es la correspondiente a una mensaje de coordinación aire - aire, sino que es otra información extraída de los registros del transpondedor.

3.4.6.2.2 Multiple Threat Encounter (MTE)

Indica cuántas amenazas están siendo procesadas por el algoritmo del *ACAS*, es decir, con cuántas aeronaves existe conflicto potencial o inminente[5].

3.4.6.2.3 Active RA (ARA)

Este campo indica las características del *RA* generado, si se ha generado alguno. Su contenido está definido por una combinación entre un bit del propio campo y el valor del campo *MTE*.

En caso de que *MTE* indique que se está procesado una amenaza (o ninguna), el campo *ARA* indica la naturaleza del *RA* (si es preventivo o correctivo). Tanto si el *MTE* está indicando una o más de una amenaza, las demás características están relacionadas con la componente vertical de la maniobra indicada por el *RA* (el sentido de movimiento indicado, si existe inversión de sentido, variación de tasa vertical, etc.).

3.4.6.2.4 Resolution Advisory Complement (RAC)

Es la información que una aeronave envía a otra para restringir sus maniobras por indicación del *ACAS*.

3.4.6.2.5 RA Terminated (RAT)

Indica si un *RA* previamente generado por el *ACAS* ha cesado su actividad o sigue activo.

[5]No se ha seguido el orden habitual para definir los campos debido a que es necesario explicar el campo *MTE* para poder entender el significado del siguiente campo

3.4.7. Transacciones squitter

Como se explicó, este tipo de transacciones son más bien emisiones, pues no existe interrogación. También se especificó la estructura de los conocidos como *squitters* de adquisición, que es la misma que la de una respuesta *All-Call Reply*, así que se procederá directamente a analizar los mensajes *Extended Squitter*.

A la hora de analizar éstos, se procederá a explicar en primer lugar el formato general de los mensajes y, después, para cada uno de los mensajes, se analizarán solamente los bits variables en un *ES*, es decir, los que se conocen como bits *ME*.

Debido a las diferencias existentes entre los mensajes *ES* de las versiónes 0 y 1, se procederán a analizar en primer lugar los correspondientes a la versión 0 y, posteriormente, se explicarán los cambios introducidos en la versión posterior con respecto a ésta.

3.4.7.1. Formato general Extended Squitter

Campo	Bits
Downlink Format (DF) = 17/18	5
Capability (CA)	3
Address Announced (AA)	14
Message, Extended Squitter (ME)	56
Parity/Identification (PI)	24

Tabla 3.12: Estructura del Extended Squitter

3.4.7.1.1 Downlink Format (DF)

Ver 3.4.1.1

3.4.7.1.2 Capability (CA)

Ver 3.4.2.1.2

3.4.7.1.3 Address Announced (AA)

Ver 3.4.2.1.3

3.4.7.1.4 Message, Extended Squitter (ME)

Es la payload del mensaje *Extended Squitter*, que depende del *TC*, definido en este mismo campo.

3.4.7.1.5 Parity/Identification (PI)

Ver 3.4.1.3

3.4.7.2. Formato del ME de Airborne Position ES, V0

Campo	Bits
Type Code (TC)	5
Surveillance Status (SS)	2
Single Antenna Flag (SAF)	1
Altitude	12
Time Synchronization (T)	1
Compact Position Reporting Format (F)	1
Enconded Latitude	17
Enconded Longitude	17

Tabla 3.13: Estructura de Airborne Position ES, V0 (sólo bits ME)

3.4.7.2.1 Type Code (TC)

Indica el tipo de mensaje ES.

3.4.7.2.2 Surveillance Status (SS)

Contiene información relacionada con el estado de alerta de la aeronave. Si el estado está activo, puede referirse a lo que se conoce como una alerta temporal, una alerta permanente o indicar que se está transmitiendo el pulso adicional *SPI*.

3.4.7.2.3 Single Antenna Flag (SAF)

Indica el tipo de sistema de transmisión que se está usando para transmitir el *ES*, antena única o un sistema dual de antenas.

3.4.7.2.4 Altitude

La altitud transmitida depende del *TC*, que indica si es barométrica o *GNSS*. En caso de ser barométrica, es equivalente a la Altitud Modo C (3.4.3.1.5), pero siempre con una resolución de 25 ft. Si es *GNSS*, se trata de la altitud sobre el elipsoide *WGS-84*. Ésta última se transmite en caso de que la barométrica no esté disponible.

3.4.7.2.5 Time synchronization (T)

Los mensajes *Extended Squitter* de posición tienen tiempos de utilización determinados. Esto es, al sobrepasar cierto tiempo, la información deja de ser válida si no es actualizada. Este campo indica se existe sincronización entre el tiempo de utilidad del mensaje con el tiempo *UTC*[6]. Los datos de posición se deben cargar en los registros del transpondedor un tiempo antes del comienzo del tiempo de utilización de los mismos. Los tiempos de utilización se definen como sigue:

- Si el campo *T* indica que existe sincronización, los mensajes de posición tienen tiempos de utilización que son denominados como "épocas *UTC* de 0.2 segundos". A partir de aquí, otro campo del mensaje conocido como *Compact Position Reporting Format (F)* determinará si el

[6]Coordinated Universal Time.

tiempo de utilización es una época par o impar. Una época par se refiere a una época o periodo que ocurre un número par de intervalos de 0.2 segundos después de un segundo *UTC* par (por ejemplo, 12 s, 12.4 s, 12.8 s...) Una época impar se refiere a una época que ocurre un número impar de intervalos de 0.2 segundos después de un segundo *UTC* par (por ejemplo, 12.2 s, 12.6 s. . .). Así, el valor *F* se va alternando entre épocas sucesivas de 0.2 segundos, indicando utilización par o impar.

- Si no existe sincronización, el tiempo de utilización no puede ser mayor de 0.2 segundos distinto al tiempo que la información permanece en el registro correspondiente del transpondedor.

3.4.7.2.6 Compact Position Reporting Format (F)

Además de indicar, como se explicó anteriormente, características del tiempo de validez del mensaje de posición, el valor *F* especifica también el tipo de mensaje de posición para determinar si la posición es decodificable o no a la hora de recibir un mensaje.

El nombre de este campo se refiere al método de codificación de posición para los mensajes *ES*. Se dice que es una codificación compacta por omitir *bits* de orden superior cuya transimisión es innecesaria, por el hecho de que se mantienen constantes durante largos periodos de tiempo (por ejemplo, el valor que indica en qué hemisferio se encuentra la aeronave).

Como estos *bits* de orden superior no se transmiten, debería existir ambigüedad a la hora de determinar la posición a partir de un mensaje. Sin embargo, la técnica utiliza dos formatos de codificación, el par y el impar, que permite decodificar de manera no ambigua la posición en caso de que un mensaje par y otro impar se reciban alternadamente en un periodo de menos de 10 segundos, pues de esta pareja de mensajes se pueden calcular esos *bits* de orden superior omitidos.

En el momento de recepción de un mensaje par o impar, es posible también determinar la posición sin haber recibido una pareja en mensajes, en caso de que la última posición válida se haya calculado, como mucho, 640 segundos antes de la recepción del mensaje. El periodo de tiempo mencionado es válido para los mensajes *Airborne Position*; para los mensajes *Surface Position*, que se analizarán posteriormente, el tiempo se multiplica por 4.

En resumen, es posible realizar una decodificación global de la posición con dos mensajes de codificación *CPR* opuesta y una decodificación local con un solo mensaje, en caso de cumplir ciertos requerimientos de tiempo. El algoritmo de decodificación *CPR* global y local se expone y se explica en el *Doc 9871*.

3.4.7.2.7 Enconded Latitude, Enconded Longitude

En estos campos se codifica la posición de acuerdo al algoritmo *CPR*. En caso de que la posición contenida en estos campos no esté disponible, pero la altitud sí lo esté, el *TC* pasa a ser nulo.

3.4.7.3. Formato del ME de Surface Position ES, V0

Campo	Bits
Type Code (TC)	5
Movement	7
Ground Track	8
Time Synchronization (T)	1
Compact Position Reporting Format (F)	1
Enconded Latitude	17
Enconded Longitude	17

Tabla 3.14: Estructura de Surface Position ES, V0 (sólo bits ME)

3.4.7.3.1 Type Code (TC)

Ver 3.4.7.2.1

3.4.7.3.2 Movement

Indica la velocidad sobre la superficie (GS[7]) del avión.

3.4.7.3.3 Ground Track

Es el ángulo de la velocidad del avión con respecto al norte geográfico, medido en sentido de las agujas del reloj. Tiene un subcampo que indica la validez de la información del campo en cuestión.

3.4.7.3.4 Time Synchronization (T)

Ver 3.4.7.2.5

3.4.7.3.5 Compact Positiong Reporting Format (F)

Ver 3.4.7.2.6

3.4.7.3.6 Encoded Latitude, Encoded Longitude

Ver 3.4.7.2.7

[7]Ground Speed.

3.4.7.4. Formato del ME de Identification and Category ES, V0

Campo	Bits
Type Code (TC)	5
Aircraft Category	3
Aircraft Identification	48

Tabla 3.15: Estructura de Identification and Category ES, V0 (sólo bits ME)

3.4.7.4.1 Type Code (TC)

Ver 3.4.7.2.1

3.4.7.4.2 Aicraft Category

Dependiendo del valor del *TC*, la información de este campo se interpretará de diferentes maneras. Es decir, un valor determinado para este campo tiene un significado distinto dependiendo del *TC*. La información de categoría de la aeronave puede estar definida fundamentalmente bien por su peso o por su tipo (planeador, ultraligero, *UAV*, etc.).

3.4.7.4.3 Aicraft Identification

También se conoce como *callsign*. Se trata de un código de identificación para la aeronave, definido en el plan de vuelo y distinto a la dirección *ICAO* de 24 *bits*. El *callsign* está compuesto por 8 caracteres, cada uno de ellos codificado en este campo por un subcampo de 6 *bits*. La codificación de caracteres se realiza por las directices del *International Alphabet Number 5 (IA-5)*, que define qué combinaciones de *bits* producen cada uno de los caracteres.

3.4.7.5. Formato general del ME de Airborne Velocity ES, V0

Estos mensajes pueden contener información diferente dependiendo de uno de sus campos. La estructura común a estos mensajes es la siguiente:

Campo	Bits
Type Code (TC)	5
Subtype	3
Intent Change Flag	1
IFR Capability Flag	1
Navigation Uncertainty Category For Velocity (NUC-R)	3
GS/Airspeed	22
Vertical Rate	11
Turn Indicator	2
GNSS-Baro Altitude Difference	8

Tabla 3.16: Estructura de Airborne Velocity ES, V0 (sólo bits ME)

3.4.7.5.1 Type Code (TC)

Ver 3.4.7.2.1

3.4.7.5.2 Subtype

Indica qué información contiene el mensaje en cuestión. En la siguiente tabla se puede ver la información del mensaje en función de este campo:

Código	Velocidad	Tipo
1	GS	Normal
2	GS	Supersónica
3	IAS/TAS	Normal
4	IAS/TAS	Supersónica

Tabla 3.17: Subtipos de velocidad en Airborne Velocity ES

3.4.7.5.3 Intent Change Flag

Indica que se ha producido la inserción de nueva información en ciertos registros. Este *bit* se activa 4 segundos después de la detección de nuevo información y permanece así durante 18 segundos (con un margen de 1 segundo).

3.4.7.5.4 IFR Capability Flag

Indica si la aeronave tiene o no capacidad para detección de conflictos mediante *ADS-B* y otras aplicaciones de este sistema.

3.4.7.5.5 Navigation Uncertainty Category for Velocity (NUC-R)

En 3.3.1.1 se explicó este concepto. En los mensajes *Airborne Velocity ES* se transmite el *NUC* en un campo del mensaje separado del *TC* (no es lo que ocurre con los mensajes de posición). Se define un error de velocidad horizontal y uno vertical, ambos con una confianza del 95 %.

3.4.7.5.6 Vertical Rate

Indica la velocidad vertical (*VR*[8]) de la aeronave. El campo tiene un indicador que especifica la fuente de este valor de velocidad (*GNSS* o medida barométrica).

3.4.7.5.7 Turn Indicator

Indica el sentido de giro de la aeronave.

3.4.7.5.8 GNSS-Baro Altitude Difference

Indica la diferencia entre el valor de altitud o altura *GNSS* y la altitud barométrica.

[8] Vertical Rate.

3.4.7.6. Formato del ME de Airborne Velocity ES V0, Subtype 0/1

Campo	Bits
Type Code (TC)	5
Subtype	3
Intent Change Flag	1
IFR Capability Flag	1
Navigation Uncertainty Category For Velocity (NUC-R)	3
East-West Velocity	11
North-South Velocity	11
Vertical Rate	11
Turn Indicator	2
GNSS-Baro Altitude Difference	8

Tabla 3.18: Estructura de Airborne Velocity ES V0, Subtype 0/1 (sólo bits ME)

3.4.7.6.1 Type Code (TC)

Ver 3.4.7.2.1

3.4.7.6.2 Subtype

Ver 3.4.7.5.2

3.4.7.6.3 Intent Change Flag

Ver 3.4.7.5.3

3.4.7.6.4 IFR Capability Flag

Ver 3.4.7.5.4

3.4.7.6.5 NUC

Ver 3.4.7.5.5

3.4.7.6.6 East-West Velocity

Como el nombre del campo indica, es la velocidad medida en el eje este-oeste. Un subcampo indica cuál es el sentido de la velocidad en este eje.

3.4.7.6.7 North-South Velocity

Velocidad medida en el eje norte-sur. Un subcampo indica el sentido de la velocidad en este eje. En ambas velocidades, dependiendo del subtipo (velocidad subsónica o supersónica), los valores estarán codificados con una resolución diferente. La resolución para valores subsónicos es de 1 nudo (kt[9]) y para los supersónicos, 4 nudos.

[9]Knots o millas náuticas por hora.

3.4.7.7. Formato del ME de Airborne Velocity ES V0, Subtype 2/3

Campo	Bits
Type Code (TC)	5
Subtype	3
Intent Change Flag	1
IFR Capability Flag	1
Navigation Uncertainty Category For Velocity (NUC-R)	3
Ground Track	11
Airspeed (IAS/TAS)	11
Vertical Rate	11
Turn Indicator	2
GNSS-Baro Altitude Difference	8

Tabla 3.19: Estructura de Airborne Velocity ES V0, Subtype 2/3 (sólo bits ME)

3.4.7.7.1 Type Code (TC)

Ver 3.4.7.2.1

3.4.7.7.2 Subtype

Ver 3.4.7.5.2

3.4.7.7.3 Intent Change Flag

Ver 3.4.7.5.3

3.4.7.7.4 IFR Capability Flag

Ver 3.4.7.5.4

3.4.7.7.5 NUC

Ver 3.4.7.5.5

3.4.7.7.6 Ground Track

Ver 3.4.7.3.3

3.4.7.7.7 Airspeed (IAS/TAS)

Velocidad aérea, que puede ser la indicada (IAS[10]) o la verdadera (TAS[11]). Un subcampo indica cuál de las dos velocidades es la que se está transmitiendo. La resolución de la información de *Airspeed*

[10]Indicated Airspeed. Es la velocidad medida por el sensor correspondiente de la aeronave.

[11]True Airspeed. Se puede obtener a partir de la IAS realizando una correción, pues el sensor que mide la IAS mide en realidad la presión dinámica, variable con la altitud.

codificada es diferente dependiendo del subtipo, al igual que ocurre en los mensajes de subtipos 0 y 1. El criterio para la resolución es el mismo que en esos mensajes.

3.4.7.8. Formato del ME de Emergency/Priority Status ES V0

Campo	Bits
Type Code (TC)	5
Subtype Code = 1	3
Emergency State	3
Reserved	45

Tabla 3.20: Estructura de Emergency/Priority Status ES V0 (sólo bits ME)

3.4.7.8.1 Type Code (TC)

Ver 3.4.7.2.1

3.4.7.8.2 Subtype Code

Indica si existe información acerca del estado de prioridad/emergencia o no. En estos mensajes, este código siempre será 1, indicando que hay una emergencia.

3.4.7.8.3 Emergency State

Indica el tipo de estado de emergencia que se pretende transmitir. Puede ser una emergencia general, médica, relacionada con el combustible, las comunicaciones, etc.

3.4.7.8.4 Reserved

Reservado para otros propósitos.

3.4.7.9. Formato del ME de Aircraft Operational Status ES V0

Campo	Bits
Type Code (TC)	5
Subtype Code = 0	3
Capabilities	48

Tabla 3.21: Estructura de Aircraft Operational Status ES V0 (sólo bits ME)

3.4.7.9.1 Type Code (TC)

Ver 3.4.7.2.1

3.4.7.9.2 Subtype Code

En estos mensajes este valor siempre será 0.

3.4.7.9.3 Capabilities

Indica el estado operacional de capacidades y aplicaciones relacionadas con ATC[12] de la aeronave. Estas capacidades pueden ser las relacionadas con con la fase en ruta, TMA[13], despegue/aterrizaje, superficie. Cuatro *bits* se usan para indicar cada una de las capacidades y otros cuatro, para indicar su estado.

[12] Air Traffic Control.
[13] Terminal Manouvering Area.

3.4.7.10. Cambios a nivel de formato en Extended Squitter V1

En 3.3.1.2 se introdujeron los cambios más importantes introducidos en la Versión 1 de *Extended Squitter*, en lo que se refiere a los nuevos conceptos añadidos. Estos cambios se reflejan en el formato de algunos de los mensajes analizados anteriormente.

3.4.7.10.1 Cambios en Surface Position ES

Se transmite el heading o el track, dependiendo de un campo en *Aircraft Operational Status ES V1* (3.4.7.12. Por lo demás, ver 3.4.7.3.

3.4.7.10.2 Cambios en Airborne Velocity ES

Se deja de transmitir el *NUC-R* y este campo pasa estar ocupado por el *NAC*. Además, la referencia para el track (norte magnético o verdadero) transmitido por este mensaje se indica en el *Aircraft Operational Status ES*. Por lo demás, ver 3.4.7.5

3.4.7.10.3 Cambios en Emergency/Priority Status ES

Este mensaje pasa a ser Aircraft Status, que puede ser de dos tipos:

- **Emergency/Priority Status**: Ver 3.4.7.8
- **ACAS RA Broadcast**: es un nuevo mensaje, sólo disponible en la Versión 1. Ver 3.4.7.11.

3.4.7.10.4 Cambios en Aircraft Operational Status

Este mensaje cambia completamente con respecto a la Versión 0. Ver 3.4.7.12.

3.4.7.11. Formato del ME de ACAS RA Broadcast ES

Campo	Bits
Type Code (TC)	5
Subtype Code = 3	3
Active Resolution Advisories (ARA)	14
Resolution Advisory Complement (RAC)	4
RA Terminated (RAT)	1
Multiple Threat Encounter (MTE)	1
Threat Type Indicator (TTI)	2
Threat Identity Data (TID)	26

Tabla 3.22: Estructura de ACAS RA Broadcast ES (sólo bits ME)

3.4.7.11.1 Type Code (TC)

Ver 3.4.7.2.1

3.4.7.11.2 Subtype Code

En estos mensajes este valor siempre será 3.

3.4.7.11.3 Active Resolution Advisories (ARA

Ver 3.4.6.2.3

3.4.7.11.4 Resolution Advisory Complement (RAC)

Ver 3.4.6.2.4

3.4.7.11.5 RA Terminated (RAT)

Ver 3.4.6.2.5

3.4.7.11.6 Multiple Threat Encounter (MTE)

Ver 3.4.6.2.2

3.4.7.11.7 Threat Type Indicator (TTI)

Indica qué información es la que está contenida en el campo siguiente, que hace referencia a la identidad de la amenaza detectada por *ACAS*. La información contenida puede ser la dirección *ICAO* o la altitud, distancia y *bearing*[14] a la amenaza.

3.4.7.11.8 Threat Identity Data (TID

De acuerdo al valor de *TTI*, contendrá una información u otra. Cabe destacar que, en caso de contener la altitud, ésta está codificada en Modo C.

[14] Ángulo que forma la línea que une una aeronave con la otra respecto al norte.

3.4.7.12. Formato del ME de Aicraft Operational Status ES V1

Campo	Bits
Type Code (TC)	5
Subtype Code = 0/1	3
Airborne/Surface Capability Class (CC) Codes + Length/Width Codes	16
Operational Mode (OM) Codes	16
Version Number	3
NIC Supplement	1
Navigation Accuracy Category - Position (NAC-R)	4
Barometric Altitude Quality (BAQ)/Reserved	2
Surveillance Integrity Level (SIL)	2
Barometric NIC (NIC-Baro)/Track-heading Indicator	1
Horizontal Reference Direction (HRD)	1
Reserved	2

Tabla 3.23: Estructura de Aicraft Operational Status ES V1 (sólo bits ME)

3.4.7.12.1 Type Code (TC)

Ver 3.4.7.2.1

3.4.7.12.2 Subtype Code

En estos mensajes este valor siempre será 0 ó 1. El valor condiciona el contenido de algunos de los campos de este mensaje.

3.4.7.12.3 Airborne/Surface Capability Class (CC) Codes + Length/Width Codes

- **Subtype Code = 0**: Este campo se conoce entonces como *Airborne Capability Class (CC)* Codes y contiene información de *ACAS* (indicando si está o no operativo o si se desconoce esta información), del *CDTI*[15] (si está operativo o no), de la capacidad de transmitir la *ARV*[16] e información relacionada con los objetivos de la aeronave.
- **Subtype Code = 1**: Este campo contiene dos subcampos que se conocen como *Surface Capability Class (CC) Code* y *Length/Width Code*.
  - **Surface Capability Class (CC) Code**: indica si el *CDTI* está operativo, si hay un *offset* aplicado en la información de posición y cuál es la potencia del transpondedor (en vatios).
  - **Length/Width Code**: indica la cantidad de espacio que la aeronave ocupa, con un código de longitud y otro de anchura.

[15] Cockpit Display of Traffic Information.
[16] Air-Referenced Velocity.

3.4.7.12.4 Operational Mode (OM) Codes

Indica si el *ACAS RA* está activado, si el *IDENT Switch*[17] está activado y si la aeronave está recibiendo servicios *ATC*.

3.4.7.12.5 Version Number

Indica la versión de *ES* que se está utilizando.

3.4.7.12.6 NIC Supplement

Valor que complementa al *NIC* transmitido en el *TC* del mensaje. Así, el *NIC* se transmite parcialmente en el *TC*, como vimos antes, y parcialmente en este suplemento. Ver 3.3.1.2.

3.4.7.12.7 Navigation Accuracy Category - Position (NAC-R)

Ver 3.3.1.2.

3.4.7.12.8 Barometric Altitude Quality (BAQ)/Reserved

- **Subtype Code = 0**: en este caso, el campo contiene el *BAQ*, indicativo de calidad de la altitud barométrica. Siempre se cumplirá *BAQ = 0*.
- **Subtype Code = 1**: en este caso, el campo contiene información reservada.

3.4.7.12.9 Surveillance Integrity Level (SIL)

Ver 3.3.1.2

3.4.7.12.10 Barometric NIC (NIC-Baro)/Track-Heading Indicator

- **Subtype Code = 0**: en este caso, se transmite *NIC-Baro*, indicativo de integridad de la presión barométrica.
- **Subtype Code = 1**: en este caso, el campo indica si *Surface Position ES* transmite el *track* o el *heading* de la aeronave.

3.4.7.12.11 Horizontal Reference Direcion (HRD)

Indica la referencia de ángulos, norte magnético o norte verdadero.

3.4.7.12.12 Reserved

Reservado para otros usos.

[17] Se utiliza para ayudar a los controladores a identificar una aeronave. El piloto pulsa un botón en la cabina que envía información de identificación.

Parte II

Diseño del sistema

Ya se introdujo en el Capítulo 1 de esta memoria la estructura del sistema a diseñar, necesaria para cumplir los objetivos de este Trabajo de Fin de Grado. En este capítulo se va a proceder a explicar esta estructura en profundidad, en lo que a *hardware* y a *software* se refiere. En la Figura 3.2 se representa gráficamente la estructura general del sistema que, como se puede observar, se divide fundamentalmente en cuatro partes.

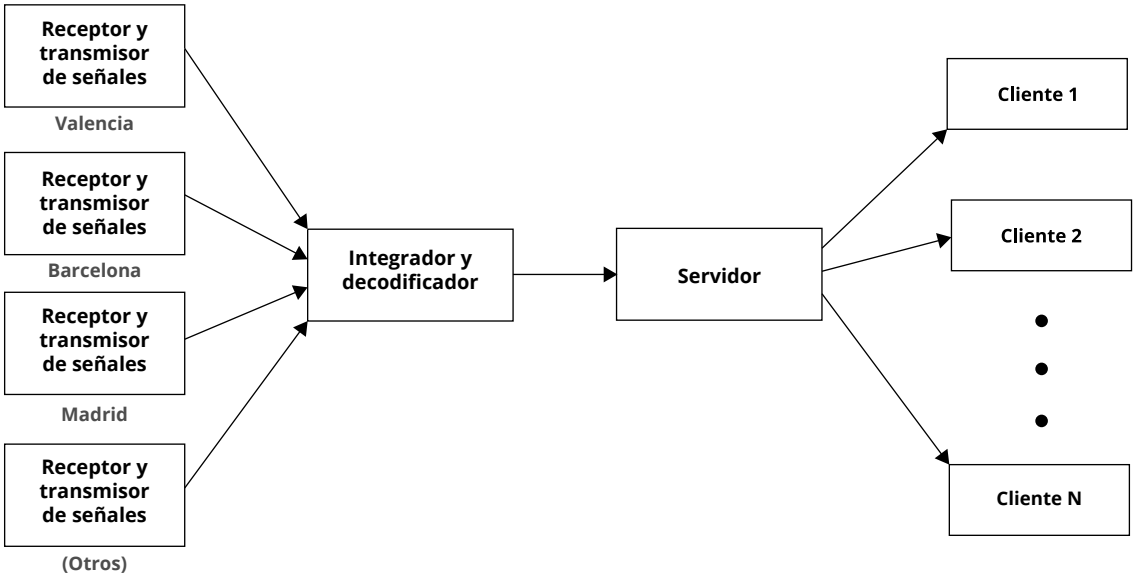


Figura 3.2: Diagrama que representa la estuructura general del sistema completo

En primer lugar, las señales a recibir por los receptores son las respuestas/difusiones Modo S de las aeronaves, introducidas y analizadas en la Parte 1. Estas señales son, entonces, convertidas a información digital y transmitidas al integrador y decodificador de información, que extrae los datos contenidos. Estos datos pasan al servidor, que distribuye la información decodificada a los clientes.

Es necesario conocer esta estructura general para entender el funcionamiento del sistema, aunque realmente la estructura tanto desde el punto de vista del hardware como del software es algo distinta. En los siguientes capítulo ésta se irá introduciendo y explicando cada una de sus partes.

4. Parte hardware del sistema

A continuación se presentan los elementos hardware del sistema, se describen y se analiza su interacción mútua.

4.1. Antena receptora con filto

Debido a que la frecuencia definida para las transmisiones aire-tierra del Modo S es de 1090 MHz, se necesita una antena capaz de captar las señales de las aeronaves a esta frecuencia y con una cobertura suficiente para cubrir el espacio aéreo deseado. Además, el filto paso-banda se utiliza para reducir el ruido *RF* (de radiofrecuencia) y aumentar el número de señales recibidas. Como la antena cuenta con un conector coaxial de tipo *N* macho, que no es del mismo tipo que el del filtro (el cual cuenta con un conector *SMA*[1]), se necesita un adaptador para realizar la conexión entre ambos componentes. El equipo en cuestión (antena y filtro) es producido por la compañía *FlightAware*[2].

Figura 4.1: Antena 1090 MHz de FlightAware

Figura 4.2: Filtro paso-banda de FlightAware

4.2. Dispositivo receptor de datos

Es el elemento que funciona como procesador de las señales que recibe la antena. Cuenta con lo que se conoce como un *SDR*, que son las siglas de *Software Defined Radio*, componente encargado de demodular las señales. Se llama así porque es un sistema que integra como parte del *software* componentes radioeléctricos que tradicionalmente han sido basados en *hardware*. El dispositivo cuenta también con un amplificador de señal. En lo que a conexiones se refiere, tiene un *SMA* que permite conectarlo a la antena y un conector *USB*. Al igual que los componentes anteriores, es distribuido por *FlightAware*.

[1] SubMiniature Version A.

[2] Compañía dedicada a software relacionado con la aviación y a servicios de datos aeronáuticos.

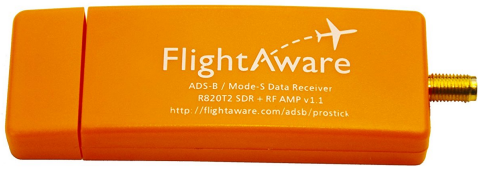


Figura 4.3: Dispositivo receptor de datos Modo S de FlightAware

4.3. Computador Raspberry Pi (RPi) y dispositivo GPS

La *RPi* es un computador de placa reducida encargado de ejecutar una aplicación capaz de enviar por uno de sus puertos la información procedente del dispositivo *SDR*. La aplicación en cuestión se llama *dump1090* y tiene una multitud de funciones que se explicarán cuando se aborde la parte software del sistema.

Actualmente existen varios modelos distintos *RPi*. A lo largo de los años, desde la primera versión, han ido apareciendo versiones más avanzadas que proporcionan una variedad más amplia de funcionalidades, y seguirán apareciendo modelos nuevos durante los años siguientes. A pesar de que los últimos modelos ofrecen más posibilidades que los anteriores, esto no significa que sean más o menos recomendables para este proyecto. Por ejemplo, el modelo 3 de la *RPi* es más avanzado que el 2, pero se calienta más, por lo que son necesarios disipadores adicionales para evitar problemas.

Además, en lo referente a las necesidades de este proyecto, es recomendable que el modelo utilizado sea compatible con un dispositivo *GPS* que proporcione una referencia de tiempo precisa al computador. La importancia de la precisión en las medidas de tiempo se puede intruir de lo explicado en relación, por ejemplo, al proceso de decodificación de posición *CPR* y se podrá entender mejor más adelante. De momento, es importante saber qué modelos son los compatibles con un dispositivo de este tipo y con otras necesidades, como son un conector *USB* para conectar el dispositivo *SDR* y un conector *Ethernet* para conectar la *RPi* a la red. Este último proceso requiere de un cable *Ethernet*, que se necesita adquirir de forma independiente a la Raspberry Pi (pues el paquete *RPi* que se distribuye oficialmente no cuenta con un cable de este tipo), y un *router* al que conectar la *RPi*.

Los modelos que cumplen actualmente con todos estos requisitos son el *A+*, el *B+* y el 2, aunque, como se ha dicho, probablemente aparezcan más versiones que los cumplan en un futuro.

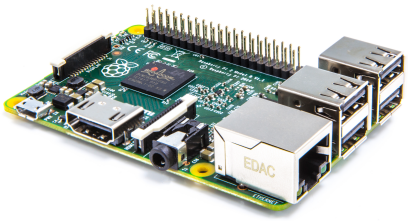


Figura 4.4: Raspberry Pi 2

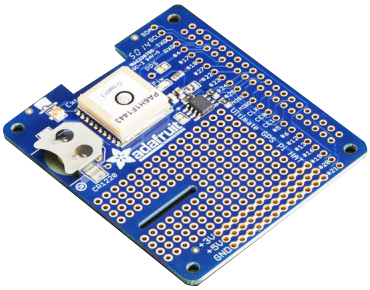


Figura 4.5: Dispositivo GPS para Raspberry Pi A+/B+/2

Es necesario un computador para configurar la *RPi* mediante *SSH (Secure Shell)*, protocolo para acceder a máquinas remotas a través de la red. Es fundamental hacerlo para instalar y configurar *dump1090* para enviar los datos por el puerto correspondiente. El sistema operativo de este ordenador no es verdaderamente relevante, pero debe tener capacidad para la conexión *SSH*.

4.4. Equipo integrador, decodificador y distribuidor (único)

Se necesita un computador para ejecutar el programa encargado de integrar y decodificar la información, además de crear el servidor para transmitirla a los clientes. El equipo debe ser lo suficientemente potente como para hacer funcionar el programa completo. Nos referiremos a este equipo a lo largo de la memoria simplemente como integrador.

5. Parte software del sistema

5.1. Los formatos de mensaje

Como ya se vio en el Capítulo 3, la *ICAO* define la estructura de cada tipo de mensaje transmitido en Modo S. Así, conociendo esta normativa, los mensajes son decodificables si se reciben en un formato que permita analizar la estructura *bit* a *bit*.

Como veremos a continuación, el sistema es capaz de recibir la información de las aeronaves en dos formatos distintos, conteniendo ambos la información introducida y explicada en aquel capítulo de manera distinta y añadiendo datos adicionales.

5.1.1. Formato Beast

Es el formato que contiene un mensaje sin decodificar, que permite su análisis mediante lo expuesto en la normativa. Es posible recibir dos subtipos de este formato, binario o caracteres *ASCII*. La estuructura del subtipo binario aparece representada en la figura siguiente (véase que esta vez organizamos la información en grupos de *bytes*, no *bits*).

Campo	**Valor**	**Bytes**
Identificador de inicio	0x1a	1
Identificador de tipo	0x32 (Mensaje Modo S corto) 0x33 (Mensaje Modo S largo)	1
Timestamp	Variable	6
Nivel de señal	Variable	1
Mensaje Modo S	Variable	7/14

Tabla 5.1: Estructura del mensaje Beast

5.1.1.0.1 Identificadores y nivel de señal

Como se puede ver, el identificador de inicio tiene siempre el mismo valor, para indicar el comienzo de un mensaje y permitir la decodificación del mismo. Por otro lado, el identificador de tipo puede tener dos valores distintos para los mensajes Modo S. Cada uno de ellos indica de qué clase (en lo que a longitud se refiere) es el mensaje Modo S que contiene, de 56 *bits* (7 *bytes*, 14 caracteres *ASCII*) o 112 *bits* (14 *bytes*, 28 caracteres *ASCII*). Por último, podemos obtener también el nivel de la señal emitida por la aeronave.

5.1.1.0.2 Timestamp

El *timestamp* es la marca de tiempo del mensaje, esto es, el momento en el que mensaje ha sido enviado. Esta marca de tiempo, cuando es transmitida en *Beast*, se actualiza a una frecuencia de 12

MHz, es decir, aproximadamente 83 nanosegundos están contenidos en un "tick[1]". Por tanto, para obtener el tiempo envío de un mensaje Beast en unidades de nanosegundos, será necesario hacer una conversión de unidades a la hora de realizar la implementación.

5.1.1.0.3 Mensaje Modo S

Dependiendo del valor del identificador de tipo, la longitud de este campo será de 7 o de 14 *bytes* y contendrá uno de los mensajes analizados en el Capítulo 3.

5.1.2. Formato BaseStation

Es un formato que contiene información decodificada de la aeronave, pero sólo una parte de ésta. Esta información se recibe de una estación base que procesa los datos. Los campos del formato están separados por comas y son los siguientes (en orden):

- *MSG*: identificador para señalar el comienzo de un mensaje.
- *Tipo de transmisión*: se trata de un número del 1 al 8 que indica el tipo de mensaje.
- *ID de la sesión*: número relativo a la sesión de la base de datos de la que se está obteniendo la información.
- *ID de la aeronave*: número identificador de la aeronave establecido en la base de datos.
- *Dirección ICAO*: identificador *ICAO* de 24 bits de la aeronave, en formato hexadecimal.
- *ID del vuelo*: número identificador del vuelo establecido en la base de datos (diferente de la dirección *ICAO* y la identidad Modo A).
- *Fecha de generación del mensaje*
- *Hora de generación del mensaje*
- *Fecha de carga del mensaje*
- *Hora de carga del mensaje*
- *Callsign*
- *Altitud Modo C*
- *Velocidad sobre tierra (GS)*
- *Track*
- *Latitud*
- *Longitud*
- *Velocidad vertical (VR)*
- *Squawk*: Código de identificación Modo A
- *Flag de alerta*: indica que el *squawk* ha sido modificado.
- *Flag de emergencia*: indica que existe una emergencia.
- *Flag SPI*: indica que el *IDENT Switch* se ha activado.
- *Flag IsOnGround*: indica si la aeronave se encuentra en tierra.

[1]El "tick" es la resolución del tiempo, la cantidad mínima medible entre una marca y otra.

Así, algunos ejemplos de estos mensajes son los siguientes:

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,,,,,

Tabla 5.2: Ejemplos de mensajes Basestation

Nota 1: Como se puede observar, aunque la información de algunos campos no esté disponible, éstos se dejan "en blanco", para mantener una estructura descifrable.

5.1.3. Comparación y elección

Como se ha visto, una gran parte de la información presente en los mensajes Modo S se encuentra omitida en *BaseStation*, como la información de *ACAS* o los niveles de precisión e integridad de la posición y la velocidad. Además, *Beast* permite obtener la marca de tiempo del mensaje con una precisión mayor. Las únicas ventajas que presenta *BaseStation* es la presencia de tres identificadores más (*ID* de la sesión, *ID* de la aeronave, *ID* del vuelo), que no tienen tanta importancia como la información extra de *Beast*, en lo que a necesidades del proyecto se refiere.

Así, debido a esta diferencia en la cantidad de información obtenible, el formato escogido para ser enviado al integrador de información es el *Beast*, de ahí que se necesite en decodificador capaz de analizar los mensajes y obtener la información contenida en ellos.

5.2. Dump1090

Se trata de una aplicación instalada en cada una de las estaciones receptoras de señales, concretamente en los computadores *RPi*, encargado de enviar la información digital recibida por un puerto hacia el equipo integrador. El protocolo utilizado para el envío de esta información por el puerto en cuestión es el *TCP/IP*[2]. Para la instalación de *dump1090* se necesita haber instalado previamente el sistema operativo *Raspbian*, una distribución de *GNU/Linux*, en la *RPi*. También es necesario llevar a cabo un proceso de configuración de la *RPi* y el dispositivo *SDR* antes de proceder a instalar *dump1090*.

Toda la configuración de la *RPi* se realiza por *SSH* por medio de otro computador con capacidad para usar este protocolo. Concretamente, existe un programa llamado "*putty*", para *Windows*, que permite realizar esta conexión. El programa accede a la consola de comandos de la máquina remota, desde donde ésta se puede configurar. Este proceso de configuración, que requiere tener conocimiento acerca de ciertos comandos de *Linux*, se abordará en profundidad en la parte de la implementación.

De momento, en lo que a diseño se refiere, necesitamos definir el puerto por el que se lanzará la información que será transmitida al equipo integrador. El programa define unos puertos por los que enviar (o recibir) datos y por cada puerto se intercambia información en formato diferente. Nosotros debemos conocer cuál es el puerto que envía datos en formato *Beast*, que es el 30005. De nuevo, se verá a la hora de implementar el programa cómo indicar al mismo que envíe información por este puerto.

Por otro lado, también es importante saber que el demodulador que se encarga de obtener el *timestamp* recibido en el mensaje *Beast* muestrea a una frecuencia de 2 MHz. Sin embargo, el *timestamp*

[2]Transmission Control Protocol/Internet Protocol.

enviado por *dump1090* está escalado a 12 MHz para alcanzar la compatibilidad con lo establecido en 5.1.1.0.2. Esto significa que la marca de tiempo recibida desde el puerto 30005 de la *RPi* es de 12 MHz, pero con una resolución que, realmente, es peor que 12 MHz. Además, *dump1090* proporciona esta marca de tiempo con su origen en el momento en el que la aplicación arranca. Esto tiene importantes consecuencias en lo que a integración de información se refiere, como se verá posteriormente, pues impide que todos los sistemas receptores tengan un origen de tiempos común.

5.3. Aplicación integradora

Esta aplicación es ejecutada en el equipo de *hardware* correspondiente. Se ha utilizado el lenguaje *Java* para realizar la programación de la misma, pues es un lenguaje orientado a objetos con una infinidad de recursos adecuados para el sistema a diseñar. Así, el lenguaje en cuestión divide al programa en distintas partes (objetos) y cada uno de ellos tiene una función específica en el conjunto global. A continuación se introducirán y se explicarán los conceptos y funcionalidades del lenguaje en cuestión utilizados en el desarrollo de la aplicación y se verá cómo cada una de estas funcionalidades es aplicada en el mismo.

5.3.1. Mapas de tráfico

Para almacenar la información decodificada de las aeronaves es necesaria la creación de un espacio actualizable en el que estén localizados todos los datos. Los contenidos de este espacio deben ser fácilmente localizables, esto es, se debe de poder acceder a la información de una aeronave en cuestión mediante algún tipo de identificador correspondiente a la misma. El espacio debe de tener también capacidad de expansión y contracción, esto es, debe de ser dinámico, para permitir el almacenamiento de una cantidad variable de información.

Para cumplir con estos objetivos se utilizan los *hash maps* de *Java*. Un *hash map* (clase *HashMap*) es un mapa de datos en el que la información se organiza en parejas: un identificador o clave y un valor correspondiente a la clave en cuestión. Los *hash maps* permiten la extracción de los valores deseados indicando los identificadores correspondientes. Así, cada una de las aeronaves tiene una clave y la información relativa a la misma está asociada a esa clave.

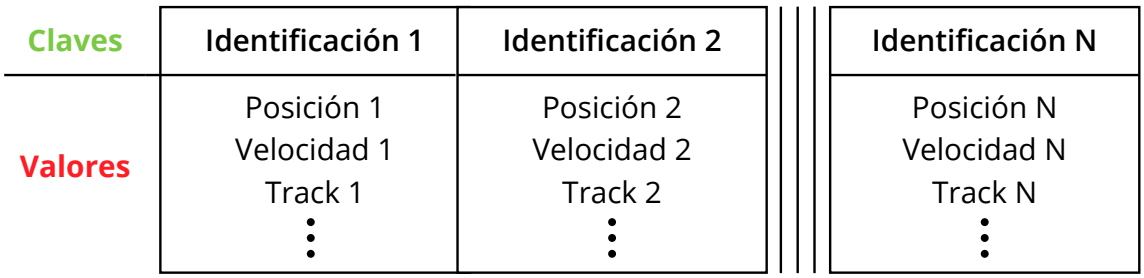


Figura 5.1: Representación gráfica del *hash map* utilizado para almacenar la información.

Cuando se pretende analizar un *hash map* en busca de una información concreta, se utiliza un iterador (interfaz *Iterator*). El iterador es un objeto que permite pasar por cada una de las claves y sus valores asociados cuando, por ejemplo, se pretende conocer el estado de unos atributos concretos. Para utilizar un iterador es necesario extraer la lista de claves del *hash map* y crear, a partir de ellas, un *set* (clase *Set*) de claves. Un *set* es una lista de objetos que sólo tiene un objeto de cada tipo (es decir, si contiene un objeto y se intenta añadir un objeto igual[3], el *set* no lo permite). A partir de aquí,

[3]El concepto de igualdad en lo que a objetos se refiere es relativo en Java, pues aunque dos objetos puedan ser iguales en sus atributos y/o métodos, tienen referencias distintas (son instancias separadas). De momento, no necesitamos conocer este concepto en profundidad.

se crea un iterador sobre este *set*, que puede ir "saltando"de valor en valor, teniendo la capacidad de detectar antes si ya no hay más valores que analizar.

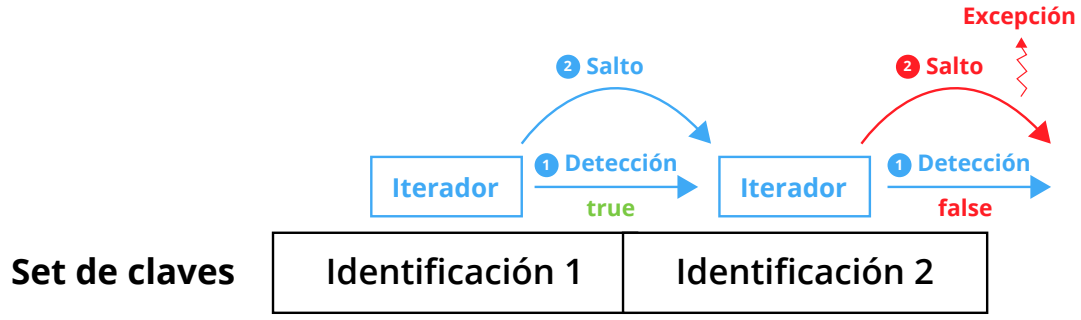


Figura 5.2: Representación gráfica del funcionamiento del iterador en Java. En primer lugar, se realiza la detección de valor siguiente y después el salto, lanzándose una excepción en caso de no haber más valores.

Los iteradores se van a utilizar para, entre otras cosas, localizar aeronaves de las que no se ha recibido información durante demasiado tiempo, es decir, las que ya no están en cobertura y por tanto deben ser eliminadas del mapa.

Nuestro programa tiene, fundamentalmente, tres *hash maps*. Veremos posteriormente que una integración eficiente de la información exige plantearse la ampliación del número de mapas, aunque de momento consideraremos que tenemos los mapas siguientes.

Uno de ellos es una lista de las aeronaves en cobertura, que contendrá como claves las direcciones *ICAO* de las mismas. El valor asociado a cada clave es un objeto que contiene como atributos cada tipo de mensaje recibido para la aeronave. Cada uno de estos tipos de mensaje es un objeto con unos atributos y métodos diferentes, permitiendo estos últimos obtener la información codificada en el mensaje en cuestion. Los valores del mapa también almacenan un tipo que tiene como atributos la latitud, longitud y altitud decodificadas de la aeronave, obtenibles mediante métodos del tipo. Nos referiremos a este mapa como "mapa de mensajes".

Otro de los *hash maps* es también una lista de aeronaves con las direcciones *ICAO* como claves, pero cuyos valores son objetos cuyos atributos son los datos decodificados que van a ser proporcionados a los clientes. Estos datos se obtienen accediendo a los métodos de los objetos asociados a cada mensaje o tipo que almacena el mapa de mensajes. Nos referiremos a este mapa como "mapa de datos".

El tercer *hash map* es una lista necesaria para la obtención de la posición, que está codificada por el algoritmo *CPR* (3.4.7.2.6). Como se vio, para calcular la posición de una aeronave es necesaria o bien una pareja de mensajes de posición recibidos en un periodo de tiempo determinado (para decodificación global), o bien un mensaje de posición junto con un valor de posición ya decodificado, ambos obtenidos también en un periodo de tiempo limitado (para decodificación local). Por ello, en este último *hash map* se almacena, asociado a la dirección *ICAO* de la aeronave, un objeto que contiene como atributos los últimos mensajes de posición (*Airborne* y *Surface*) junto con la codificación de cada uno (par o impar) y la última posición decodificada. Analizando todos estos elementos para una aeronave, este objeto decodifica mediante un método la posición, si es posible hacerlo.

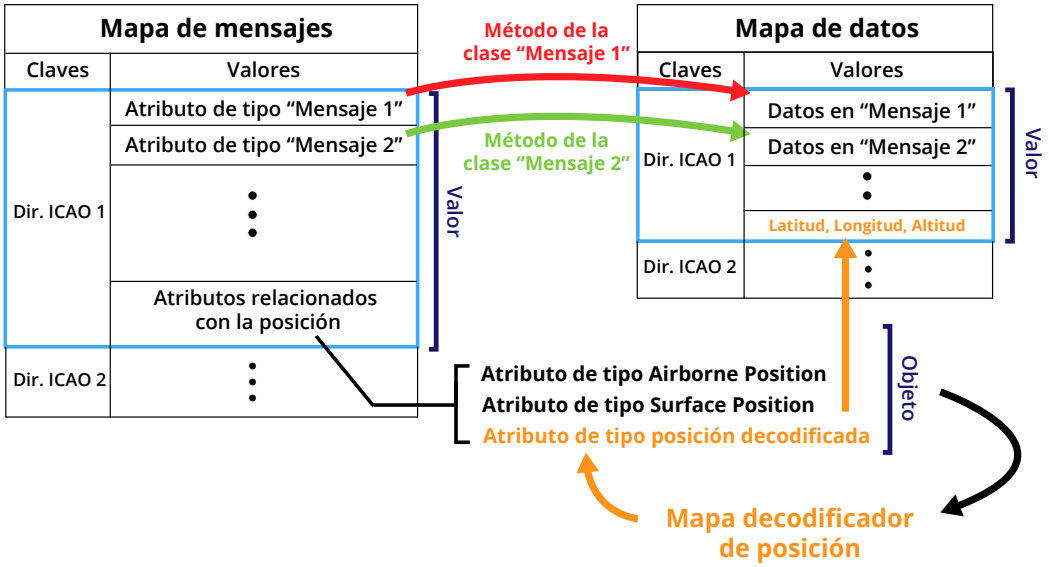


Figura 5.3: Representación gráfica del funcionamiento de los hash maps definidos.

Es fundamental también establecer marcas de tiempo para cada tipo de mensaje de posición recibido, pues es necesario para la decodificación de la posición, y para el último mensaje recibido para una aeronave, para poder averiguar cuándo ésta pasa a estar fuera de cobertura.

5.3.2. Hilos de ejecución

Algunos de los objetos que componen el programa generan procesos que necesitan ser ejecutados concurrentemente, lo que hace necesario el uso de un mecanismo conocido como *multithreading*. Este mecanismo, utilizable en *Java*, permite la creación de lo que se conoce como un hilo (*thread*) por cada uno de los procesos que tienen que ser ejecutados de manera concurrente. Los procesos en cuestión son el actualizador y el limpiador de tráfico, además del servidor que proporciona la información a los clientes.

5.3.2.1. Los hilos en funcionamiento

5.3.2.1.1 Actualizadores

Son los hilos encargados de recibir, decodificar e introducir nueva información recibida en los mapas de tráfico. Se necesita un hilo actualizador por cada antena receptora de señales. Detectan el tipo de mensaje recibido y lo asignan a la aeronave correspondiente. Se realiza la actualización del mapa de mensajes y, a partir de éste, la del mapa de datos, con los métodos correspondientes. También asigna una clave en los mapas para cada una de las aeronaves nuevas que entran en cobertura.

Como ya se ha visto y se detallará más tarde, los mensajes individuales se reciben el forma de grupos de *bytes*. Como se conoce al estructura de los mismos, se pueden decodificar leyendo *byte* a *byte* (o grupos de *bytes*), comprobando si coinciden con lo establecido en 5.1.1, extrayendo la información correspondiente y pasándola por el decodificador. Éste se analizará en detalle más tarde.

5.3.2.1.2 Limpiador

Es el hilo encargado de eliminar de los mapas de tráfico las aeronaves que pasan a estar fuera de cobertura. Detecta, con un iterador, cuándo una de las aeronaves de los mapas no ha sido actualizada durante un determinado tiempo y procede a eliminarla si es necesario. Esto se consigue mediante el

mecanismo de las marcas de tiempo (*timestamps*), proporcionadas por los propios mensajes, lo que permite saber el momento en el que se ha generado cada mensaje.

5.3.2.1.3 Servidor

Es el hilo encargado de leer la información del mapa de datos y proporcionarla a los clientes que la soliciten por un protocolo y en un formato definidos. Se crea un hilo por cada cliente conectado al sistema. El funcionamiento detallado del servidor se abordará posteriormente.

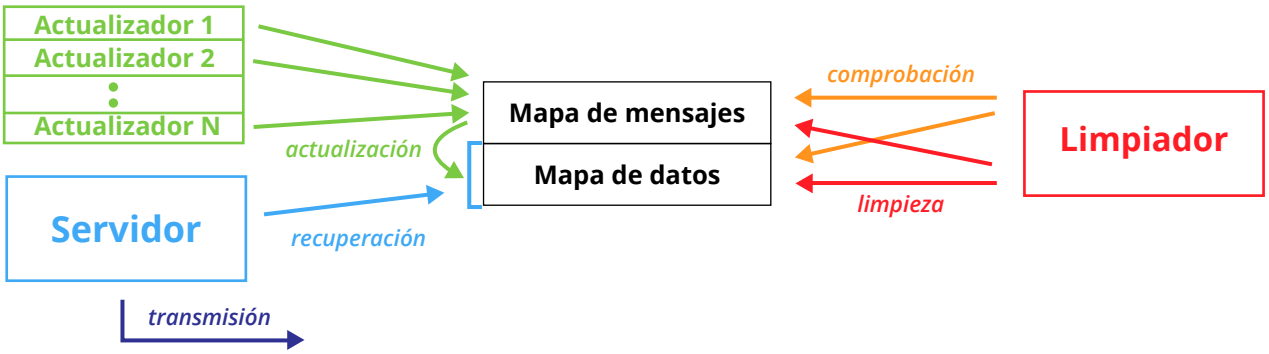


Figura 5.4: Representación gráfica de los hilos de ejecución y sus funciones.

El uso del *multithreading* trae consigo la necesidad de evitar conflictos relacionados con el acceso a la misma información (o un mismo objeto) por parte de programas ejecutados en *threads* distintos.

5.3.2.2. La sincronización de objetos

Los conflictos antes mencionados se solucionan aplicando el concepto de monitor de la programación concurrente. Un monitor es un objeto que puede ser utilizado por más de un hilo sin peligro de generación de conflictos. Se dice que los métodos del objeto en cuestión son ejecutados con exclusión mútua, esto es, que sólo pueden ser accedidos por más de un hilo al mismo tiempo, bloqueando el acceso de otros hilos cuando uno de ellos lo está utilizando[4].

Los objetos de *Java* pueden adquirir esta propiedad mediante la aplicación de que se conoce como sincronización. Sincronizar un objeto o un método significa, básicamente, convertirlo en un monitor. Así, cuando un proceso ejecutándose en un hilo realiza una tarea que implica el acceso a un objeto o método sincronizado, el acceso al objeto o método estará bloqueado hasta que el hilo en cuestión haya terminado la tarea correspondiente. Los demás procesos no podrán acceder hasta el objeto o método quede liberado del bloqueo.

[4]La definición completa de monitor es más compleja, pues los monitores tienen más características aparte de las mencionadas, pero nosotros con quedaremos con las verdaderamente relevantes para nuestras necesidades.

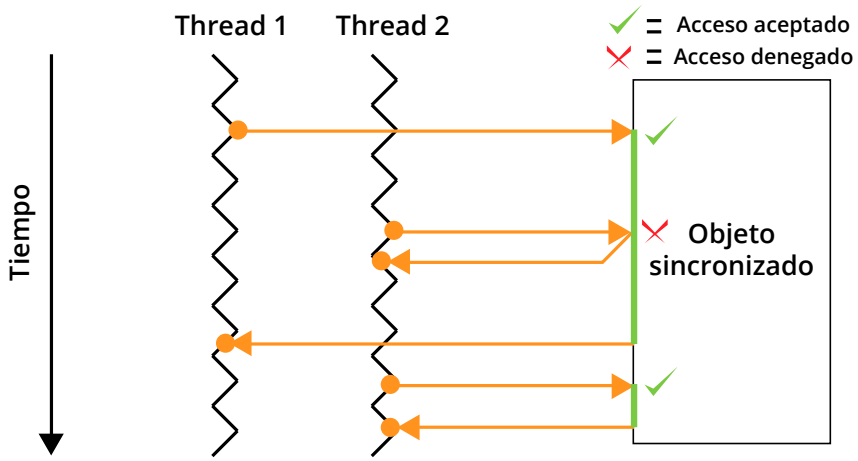


Figura 5.5: Representación gráfica del funcionamiento de un monitor u objeto sincronizado.

En nuestro caso, necesitamos sincronizar los mapas de tráfico definidos, pues van a acceder a ellos procesos en ejecución en *threads* distintos.

5.3.3. Decodificación

El decodificador de mensajes es un método que toma los mensajes Modo S recibidos en forma de cadena de caracteres *ASCII* junto con su *timestamp* (como tipo de dato numérico) y la información útil de contienen. Por tanto, antes de pasar al decodificador, es necesario convertir los valores que éste va a tomar a los tipos correspondientes.

El método se encarga de analizar el mensaje en varias fases. En primer lugar, lo analiza como mensaje Modo S general, esto es, extrae la información común a todos los mensajes (ver la Tabla 3.3), creando un objeto que contiene esta información. Uno de los datos contenidos en esa información es el *Downlink Format (DF)*. El decodificador entonces toma el *DF* y comprueba si es igual a alguno de los definidos por la *ICAO* (ver Tabla 3.16). Si se da el caso, se crea un nuevo objeto más "específico", es decir, que contiene (además de la información común) los datos correspondientes al mensaje en cuestión.

Por último, si se trata de un mensaje *Extended Squitter*, aparece un nivel más en la decodificación. El objeto correspondiente al mensaje *Extended Squitter* contiene el *TC*, que indica de qué *ES* se trata. Así, se comprueba si este *TC* coincide con alguno de los de la Tabla 3.2 y se crea el objeto correspondiente al mensaje.

Es necesario realizar un correcto manejo de las excepciones que se pueden lanzar a la hora de decodificar la información cuando, por ejemplo, el mensaje a partir del cuál se crea un objeto no tiene la longitud que debería o si se intenta obtener un atributo del mensaje (con un *getter*[5]) que no está definido en el mismo. Estas excepciones se definirán y se explicarán posteriormente.

[5]Un "getter."[es] un método que permite obtener un atributo de un objeto. Por otro lado, un "setter."[es] un método que permite establecer el valor de un atributo del mismo.

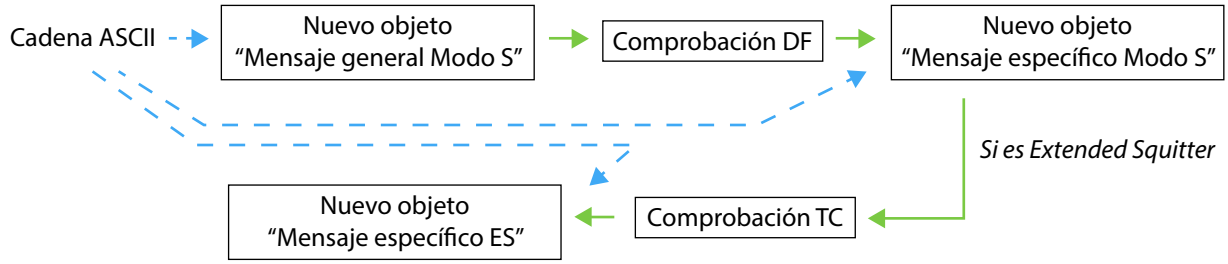


Figura 5.6: Representación gráfica del funcionamiento general del decodificador (suponiendo que se reciben los mensajes adecuados, esto es, que no se generan excepciones).

Nota : Las líneas discontinuas se utilizan para referirse al parámetro utilizado para la creación de los objetos y las continuas, para indicar los pasos que sigue el hilo decodificador para descifrar un mensaje.

5.3.3.1. El extrapolador de posición

Puede darse el caso de que una aeronave deja de enviar mensajes de posición, pero envia otros mensajes (de velocidad, *track*, etc.) que permiten calcular la posición en un instante determinado. Estos mensajes se pueden aprovechar para extrapolar la posición en estos casos y así tener un valor más o menos preciso de la posición de manera continua. Debido a la inexactitud que siempre existe al realizar esta estimación, es necesario realizar una distinción entre la última posición conocida de la aeronave (calculada por *CPR*) y la posición extrapolada. Es decir, esta última se proporciona como un valor adicional, no como sustituto de la posición real y precisa de la aeronave.

Para permitir el cálculo de la posición extrapolada, se han definido cuatro "*flags*[6].[en] la clase cuyas instancias están contenidas dentro del mapa de datos (Ver 5.3.1). Estos "flags"son activados/desactivados por el decodificador y el extrapolador para indicar ciertas condiciones relacionadas con la extrapolación.

- **Flag de reinicio**: se activa cada vez que se obtiene un nuevo valor de posición a partir de mensajes de posición.
- **Flag de altitud**: se activa cada vez que se obtiene un nuevo valor de altitud (en un mensaje Modo C, *Long ACAS*, *Short ACAS*, etc.).
- **Flag de extrapolación**: se activa cuando se dan las condiciones para realizar la extrapolación y permite la realización de la misma.
- **Flags de valores para extrapolación**: se activan cuando se actualiza alguno de los valores utilizados para realizar la extrapolación (velocidad, *track*, etc.).

El extrapolador actúa después de haberse recibido un mensaje Modo S válido. Para realizar la extrapolación, se deben dar las condiciones necesarias, entre las que están una diferencia entre el *timestamp* del último mensaje recibido y el de la última posición obtenida de, al menos, 3 segundos. Así, se necesita un *timestamp* adicional, referido a la última posición obtenida. La figura siguiente muestra su funcionamiento general.

[6] Atributo de tipo booleano.

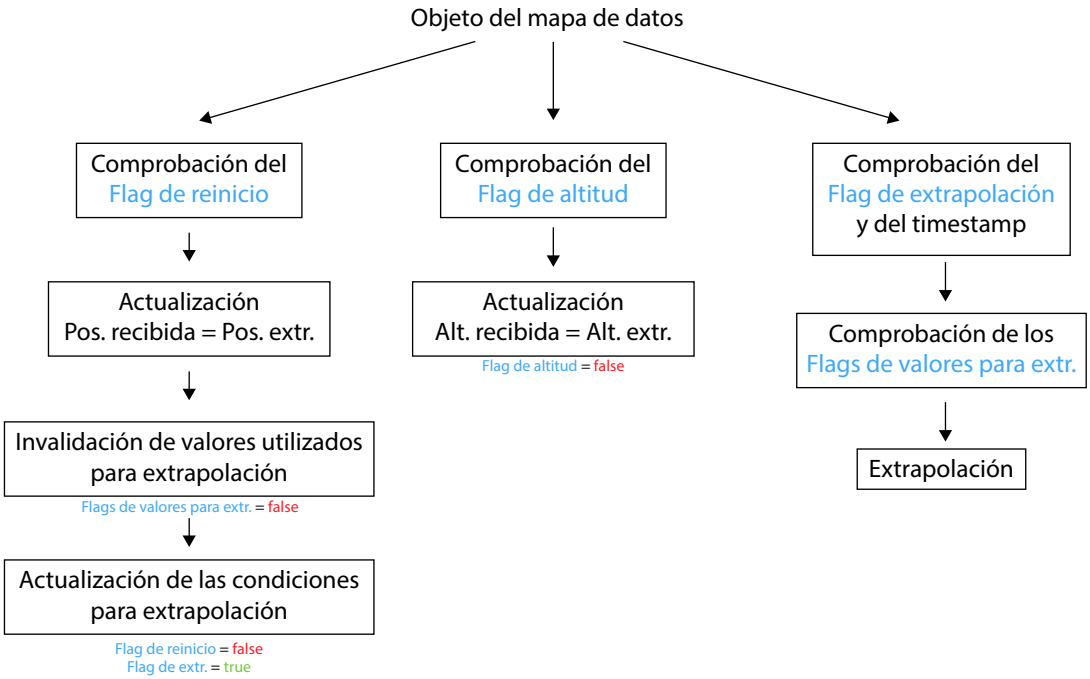


Figura 5.7: Representación gráfica del funcionamiento general del extrapolador.

Cuando se recibe un mensaje Modo S válido, en primer lugar se comprueba si se ha actualizado la posición de la aeronave. En tal caso, la posición extrapolada se iguala a la última recibida y se invalidan los valores de usados para extrapolación (velocidad, *track*, etc.) hasta recibir nuevos datos de este tipo. Se activa la capacidad de extrapolación de la aeronave.

Por otro lado, si se recibe un mensaje de altitud, simplemente se actualiza la altitud de la aeronave, pero no se invalida la información usada en extrapolación y ésta se sigue usando para extrapolar.

Por último, se comprueba si la aeronave tiene capacidad de extrapolación y se evalúa su *timestamp*, que debe tener, como hemos dicho, una diferencia de 3 segundos con el *timestamp* del último mensaje de posición para que se considere necesaria la extrapolación. Además, se comprueba si hay datos válidos para realizar la extrapolación y, si se da este caso, ésta se realiza.

La extrapolación en sí se realiza mediante un algoritmo de estimación de posición loxodrómica. El concepto de distancia loxodrómica entre dos puntos sobre la superficie terrestre se refiere a la distancia que sigue un rumbo constante, en contraste con la ortodrómica, que es la mínima distancia entre dos puntos. Así, conociendo el rumbo de la aeronave junto con otros datos, se puede calcular la posición siguiente de la misma en la línea loxodrómica. Se han utilizado solamente los datos de *track, GS, VR* y tiempo para realizar esta estimación.

5.3.4. Limpieza de tráfico

La limpieza de tráfico es fundamental porque es totalmente ineficiente almacenar en una lista todas las aeronaves que ha detectado el sistema sin descartar las que pasan a estar fuera de cobertura, pues los mapas van almacenando información que no es útil, lo que hace, por ejemplo, que los iteradores tarden cada vez más tiempo en recorrer los mapas para realizar sus funciones. Así, si no se produce una limpieza, finalmente el servicio proporcionado a los clientes no es el deseado. Por ello, el sistema cuenta con un limpiador de tráfico, que elimina de ambos mapas las aeronaves que se considera que han salido del área cubierta por el sistema. Esta salida de la cobertura se detecta cuando se dejan de recibir señales de una aeronave concreta en un periodo de 60 segundos, que se considera bastante razonable.

Esta necesidad de limpieza se debe también a lo que se conocen como direcciones *ICAO* temporales. Cuando en un área cubierta por varios radares *SSR* uno de éstos quiere distinguir su respuesta de las respuestas a otros radares, incluye en la interrogación un código *ICAO* temporal con el que la aeronave responderá. Esta dirección es distinta a la fija de la aeronave, esto es, la asignada para las demás transacciones Modo S. Estas direcciones generalmente se escogen de entre las que no están en uso y no se usan en mensajes de identificación Modo S.

La cuestión es que, para distinguir las direcciones *ICAO* temporales de las fijas, se necesitaría escuchar la interrogación de los radares, algo de lo que nuestro sistema no es capaz. Sin embargo, éstas se pueden detectar por el hecho de que, después de recibirse un mensaje con la dirección temporal, generalmente no se reciben más con esta dirección. Por ello, la misma limpieza para eliminar aeronaves fuera de cobertura sirve a su vez para eliminar las direcciones *ICAO* temporales.

Para realizar esta limpieza, necesitamos almacenar tanto el *timestamp* del último mensaje enviado por parte de una aeronave como el *timestamp* "actual", que corresponde al último *timestamp* recibido para cualquiera de las aeronaves. Así, cada segundo, ambos *timestamps* se comparan para decidir si una aeronave se debe eliminar de los mapas, lo que se consigue mediante un iterador que recorre los mapas.

5.3.5. Transmisión de información

Nuestra aplicación debe encargarse de establecer la comunicación entre el equipo integrador y las máquinas o sistemas que necesitan intercambiar información con él. Esta comunicación debe realizarse por unos protocolos definidos, que se adapten a las necesidades del proyecto y faciliten la obtención de información decodificada de tráfico por parte de los clientes.

En todo sistema se utiliza un único protocolo de transmisión, el *TCP/IP*. *Java* proporciona recursos para establecer fácilmente la comunicación entre sistemas mediante este protocolo. También tenemos una multitud de clases que permiten la transmisión y obtención de datos. Veamos cuáles son los recursos de *Java* utilizados para los fines mencionados.

5.3.5.1. Establecimiento de la conexión

Java proporciona unas clases llamadas *Socket* y *ServerSocket*, que permiten establecer la conexión mediante *TCP* entre sistemas. Un *socket* es el punto final de comunicación entre dos máquinas. En el caso de la clase *Socket*, en una máquina se crea un *socket* que se conecta en otra, conociendo la dirección *IP* y un puerto de ésta. En el caso de *ServerSocket*, se crea un *socket* en la propia máquina y se esperan conexiones externas para enviarles una información determinada. Es importante que los puertos en cuestión no estén bloqueados por el cortafuegos o de alguna otra forma; en caso contrario, no es posible establecer una conexión. En nuestro sistema, necesitamos crear, en primer lugar, la conexión entre las *RPi* y el equipo integrador, y en segundo lugar, entre el equipo integrador y los clientes.

- **Conexión RPi - equipo integrador**: para esta parte, en lo que se refiere a las *RPi* sólo se necesita ejecutar *dump1090* con las órdenes correspondientes para enviar información por el puerto 30005. Por parte del equipo integrador, nuestro programa *Java* debe crear el *Socket* para "escuchar"la información que llega por el puerto en cuestión. Es importante que las direcciones *IP* de las *RPi* en funcionamiento sean fijas, para evitar cambios continuos en el código.
- **Conexión integrador (servidor) - clientes**: para esta parte, no conocemos las direcciones *IP* y los puertos con los que debemos conectarnos, sino que debemos enviar nuestra información por un puerto y esperar a que alguien se conecte. Esto se consigue mediante el *ServerSocket*. Para poder conectarse, un cliente debe conocer la dirección *IP* (externa o interna, dependiendo de si éste se encuentra o no en la red interna del sistema) y el puerto en el que se ha creado

ServerSocket. Así, una vez se detecta una solicitud de conexión, esta se acepta y la conexión se establece. Este proceso se detallará un poco más posteriormente.

Figura 5.8: Esquema resumen de las conexiones del sistema.

5.3.5.2. Recepción y envío

Para recibir los datos de la *RPi* y enviar la información decodificada a los clientes se utilizan unas clases basadas en los conceptos *input stream* y *output stream*. Un *input stream* es el flujo de datos entre dos máquinas desde el punto de vista de la lectura y un *output stream* lo es desde el punto de vista de la escritura. Es decir, si nuestra intención es leer unos datos transmitidos, necesitaremos acceder al *input stream* de los datos en cuestión y si queremos añadir información al flujo de datos, accederemos al *output stream*.

En *Java*, las clases *InputStream* y *OutputStream* permiten obtener los *streams* correspondientes de los *sockets* creados. Es decir, para leer el flujo que llega desde la *RPi* obtendremos el *input stream* enviado por el *socket* creado en el puerto 30005 y para escribir información sobre el flujo transmitido a los clientes, obtendremos el *output stream* enviado por el *socket* del puerto correspondiente.

Sin embargo, sólo obtener los *streams* no es suficiente. En el caso de la recepción por parte del integrador, necesitamos que la información recibida sea legible y pueda ser decodificada. Es decir, es necesario obtener los contenidos el *input stream* en forma de un *array*[7] de *bytes*, que puedan ser leídos para decodificar la información. En el caso del envío a los clientes, necesitamos tener la capacidad de escribir información en forma de *bytes* sobre el *output stream*. Java permite todo esto mediante las clases *DataInputStream* y *DataOutputStream*, que ofrecen la posibilidad de leer y escribir tipos de datos primitivos en un *input/output stream*, con independencia de la máquina utilizada por el programa.

Con lo mencionado anteriormente, ya tenemos diseñado el proceso de intercambio de información entre los sistemas. Los hilos actualizadores se encargarán de recibir la información en forma de *bytes* de las *RPi*, decodificarla y actualizarla y el hilo servidor, de aceptar conexiones de clientes y enviarles datos también en forma de *bytes*.

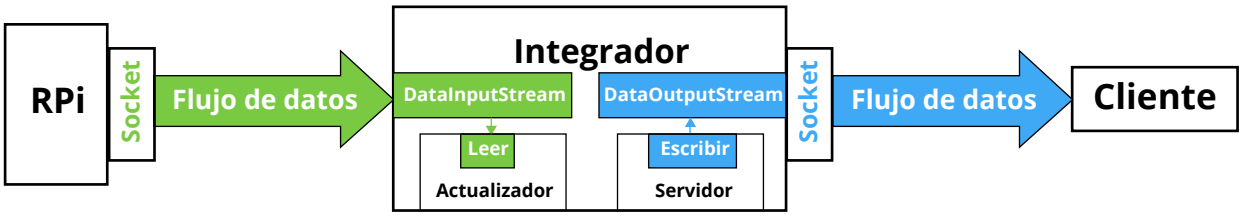


Figura 5.9: Esquema resumen de los procesos de intercambio de información.

[7]Un array se refiere a un conjunto de datos referenciados a una variable o atributo, ordenados y localizables mediante un identificador numérico.

5.3.5.3. El funcionamiento del servidor

Como se ha visto, el servidor obtiene la información contenida en el mapa de datos y la envía a los clientes. Es necesario, por tanto, diseñarlo de forma que sea capaz de esperar conexiones de clientes y aceptarlas, momento en el que debe comenzar a formatear los datos que van a ser enviados, para hacerlos legibles y comprensibles.

La clase *SocketServer* de *Java* resuelve la parte de la espera de conexiones con un método implementable que se encarga, simplemente, de aceptar a los clientes que establecen una conexión *TCP* con el servidor. Una vez se acepta una conexión, se crea un *socket* y un hilo que se va encargar de enviar datos por ese *socket*. El servidor sigue esperando y aceptando nuevas conexiones.

Cuando se crea una conexión, el hilo correspondiente llama un método que lee el mapa de datos con un iterador y escribe la información que va a ser enviada al cliente, con el siguiente formato:

D: *Fecha y hora* N: *Número de aeronaves*
**,índice,icao24,callsign,squawk,lat,lon,alt,lat_extr,lon_extr,alt_extr,gs,ias,tas,track,vr,timestamp*

La cursiva hace referencia a valores variables en cada transmisión. En primer lugar, la fecha y hora hacen referencia al momento en el que el método al que el servidor llama comienza a recorrer el mapa de datos, que prácticamente coincide con el momento en el que la información es enviada al cliente. En segundo lugar, tenemos el número de aeronaves cuya información es transmitida en el mensaje servidor-cliente, que coincide con el número de líneas que vienen después de la actual.

Después, cada línea de información es ocupada por datos de una aeronave en concreto que se encuentra en la zona de cobertura del sistema. Al igual que en los mensajes *Beast* recibidos desde *dump1090*, al comienzo de cada línea se coloca un identificador para indicar que a continuación viene información de una aeronave diferente. Todos los datos de las aeronaves transmitidos van separados por comas, lo que facilita la lectura de la información y le da uniformidad al conjunto.

El índice se utiliza para identificar la línea en la que se encuentra cada aeronave. Además, como se puede observar, se transmite tanto la posición recibida como la extrapolada, además del *timestamp* del último mensaje recibido para la aeronave en cuestión.

5.3.6. Integración y selectividad

Como se ha visto, nuestra aplicación cuenta con dos mapas en los que almacenar la información de tráfico, un mapa de mensajes y un mapa de datos. El funcionamiento del proceso de integración en esta versión de la aplicación es muy sencillo, pues simplemente se van actualizando ambos mapas cada vez que se recibe un mensaje válido. Además, no se produce ninguna comprobación relacionada con la resolución de ambigüedades que aportase robustez a la aplicación. Al final de la memoria se presentarán posibles soluciones con el objetivo de mejorar este proceso de integración, que a su vez permitiría al sistema tener la capacidad de ofrecer un servicio selectivo.

Si se realizan mejoras en integración y selectividad, y además se conoce de manera precisa la cobertura de cada una de las antenas, cuando un cliente solicite la información correspondiente a un área concreta y se sabe que este área está cubierto por, digamos, dos antenas (identificables de alguna manera), al cliente en cuestión se le enviará la información correspondiente a estas antenas, es decir, la de sus mapas.

Por otro lado, una correcta integración implica unos requisitos en las marcas de tiempo de los mensajes. En primer lugar, una alta precisión, que se consigue mediante la extracción del *timestamp* del mensaje

Beast. Sin embargo, otro requisito es la uniformidad en el origen de tiempos para estas marcas. Todo esto se debe a que las áreas cubiertas por las antenas se solapan en gran parte, lo que significa que es inevitable recibir los mensajes de una misma aeronave en estaciones diferentes. En estas situaciones, debe haber alguna forma de descartar un mensaje que llega a una estación después de haber llegado a otra.

Como se mencionó en 5.2, *dump1090* proporciona una marca de tiempo para el mensaje *Beast* con origen en el momento de inicio de ejecución de la aplicación. Por ello, es fundamental aplicar algún tipo de corrección a esta marca de tiempo para alcanzar la uniformidad buscada.

5.3.7. Manejo de excepciones

El manejo de excepciones es una parte fundamental en cualquier aplicación *Java* y, sobre todo en una tan compleja como la que se trata aquí, es difícil crear un programa robusto que pueda solventar todas las situaciones posibles. Una excepción se lanza cuando durante el funcionamiento de la aplicación ocurre algo inusual o inesperado, que no corresponde con su funcionamiento normal. En el caso de nuestro sistema, estas situaciones pueden ser, por ejemplo, la recepción de señales inválidas o errores de conexión e intercambio de información.

La última versión de la aplicación en cuestión tiene cierta robustez. Por ejemplo, los mensajes recibidos que no corresponden con los del Modo S generalmente se descartan. Esto se consigue lanzando una serie de excepciones que se han creado para manejar ciertas situaciones:

- **Excepción de error de formato**: se lanza generalmente cuando se intenta crear un objeto de tipo mensaje Modo S a partir de una cadena de caracteres *ASCII* de longitud inválida para el mensaje en cuestión. Por ejemplo, si se intenta crear un objeto de tipo mensaje Modo S general con una cadena de caracteres *ASCII* de longitud distinta a 14 o 28 caracteres, se lanza esta excepción.
- **Excepción de información inexistente**: se lanza cuando se intenta acceder a un atributo de un mensaje con un *getter* pero el atributo en cuestión no está definido. Por ejemplo, si en el mensaje *Surface Position ES* el *bit* de validez del *heading* indica que el valor de éste es inválido, si se intenta acceder al campo del *heading* con un *getter*, se lanzará esta excepción.
- **Excepción de decodificación de posición**: se lanza cuando se intenta decodificar la posición por el algoritmo *CPR* a partir de información incompatible entre sí. Por ejemplo, si se intenta decodificar la posición a partir de dos mensajes entre los que existe una transición de latitud (el número de zonas de longitud es diferente), se lanzará esta excepción.

Otra excepción lanzada por la aplicación es la *IOException*, que corresponde a una excepción general para situaciones relacionadas con la conexión *TCP* entre máquinas. Veremos en la parte de la implementación la variedad de las situaciones en las que se lanza ésta y los diferentes significados que puede tener.

5.4. La representación de datos

Desde el punto de vista del cliente, como se ha visto, la información enviada por el servidor es recibida en forma de *array* de *bytes*. Así, simplemente conociendo la estructura explicada en 5.3.5.3, el cliente puede extraer los datos de tráfico para utilizarlos como desee.

Se ha desarrollado un ejemplo de aplicación de cliente que se conecta al servidor y representa la información recibida en una tabla. Una aplicación de este tipo se conoce como *display*. Ésta se ha diseñado e implementado también en *Java* aunque se podría realizar con cualquier otro lenguaje que tuviera las capacidades necesarias. El lenguaje *Java* se caracteriza por proporcionar una enorme cantidad de recursos para el desarrollo de representaciones o interfaces gráficas y para esta parte se

utilizarán algunas de ellas. En la parte de la implementación se explicarán en detalle las clases utilizadas. De momento, vamos a estudiar el funcionamiento general.

El *display* es un hilo de ejecución. Como ya hemos visto, en primer lugar se debe crear un *socket* para recibir la información del servidor. Esta información es transmitida en un flujo de datos que se lee línea a línea por la aplicación de representación. Como se conoce la estructura de los datos recibidos, éstos se pueden decodificar mediante un algoritmo que detecte las diferentes partes de la estructura en cuestión y extraiga la información correspondiente a cada una. Cada uno de los datos transmitidos (dirección *ICAO*, *callsign*, *squawk*, etc.) se almacena en un *array* de *String*[8], de una longitud determinada por el número de aeronaves cuya información se transmite (el número de aeronaves está presente en la estructura, como se ha visto).

Así, al leer el principio de cada bloque de datos, se redefinen los *arrays* de *String* para ajustarlos al número de aeronaves en cobertura. Después, hasta el final del bloque, se lee cada dato y se almacena en el *array* correspondiente. Todo lo anterior se puede ver representado en la figura siguiente.

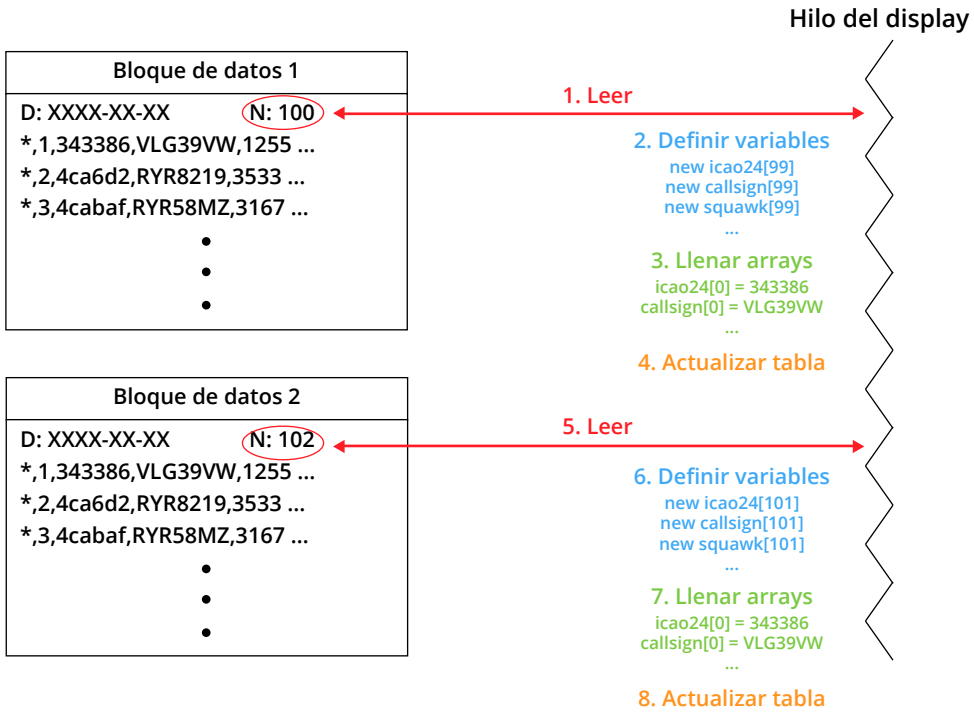
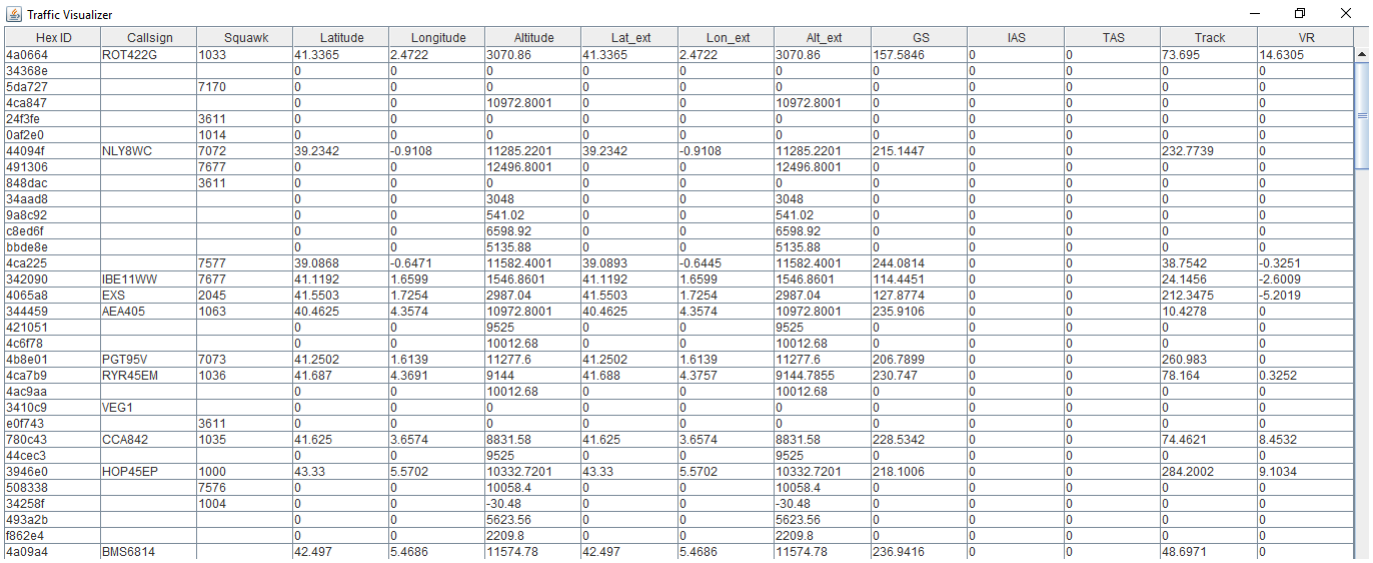


Figura 5.10: Representación del funcionamiento del hilo del display.

De momento, sólo hemos conseguido extraer y almacenar cada uno de los datos. Como se puede ver en en gráfico, el último paso a la hora de trabajar con un bloque de datos es actualizar una tabla que represente la información. Esta tabla debe tomar la información de los *arrays* cada vez que ésta se actualice y reflejar estos cambios. *Java* proporciona una clase llamada *AbstractTableModel*, que es una tabla predefinida en la que se necesitan sobreescribir los métodos correspondientes a la información contenida en la misma. Cada vez que se produce la actualización de los *arrays*, un método de la tabla redefine los datos y se le lanza un aviso para indicar esta redefinición, lo que hace que los valores mostrados en la tabla se actualicen.

A continuación, se muestra el aspecto del *display* creado, mostrando información de tráfico en un instante determinado.

[8]Tipo de atributo que corresponde a una cadena de caracteres.



Hex ID	Callsign	Squawk	Latitude	Longitude	Altitude	Lat_ext	Lon_ext	Alt_ext	GS	IAS	TAS	Track	VR
4a0664	ROT422G	1033	41.3365	2.4722	3070.86	41.3365	2.4722	3070.86	157.5846	0	0	73.695	14.6305
34368e			0	0	0	0	0	0	0	0	0	0	0
5da727		7170	0	0	0	0	0	0	0	0	0	0	0
4ca847			0	0	10972.8001	0	0	10972.8001	0	0	0	0	0
24f3fe		3611	0	0	0	0	0	0	0	0	0	0	0
0af2e0		1014	0	0	0	0	0	0	0	0	0	0	0
44094f	NLY8WC	7072	39.2342	-0.9108	11285.2201	39.2342	-0.9108	11285.2201	215.1447	0	0	232.7739	0
491306		7677	0	0	12496.8001	0	0	12496.8001	0	0	0	0	0
848dac		3611	0	0	0	0	0	0	0	0	0	0	0
34aad8			0	0	3048	0	0	3048	0	0	0	0	0
9a8c92			0	0	541.02	0	0	541.02	0	0	0	0	0
c8ed6f			0	0	6598.92	0	0	6598.92	0	0	0	0	0
bbde8e			0	0	5135.88	0	0	5135.88	0	0	0	0	0
4ca225		7577	39.0868	-0.6471	11582.4001	39.0893	-0.6445	11582.4001	244.0814	0	0	38.7542	-0.3251
342090	IBE11WW	7677	41.1192	1.6599	1546.8601	41.1192	1.6599	1546.8601	114.4451	0	0	24.1456	-2.6009
4065a8	EXS	2045	41.5503	1.7254	2987.04	41.5503	1.7254	2987.04	127.8774	0	0	212.3475	-5.2019
344459	AEA405	1063	40.4625	4.3574	10972.8001	40.4625	4.3574	10972.8001	235.9106	0	0	10.4278	0
421051			0	0	9525	0	0	9525	0	0	0	0	0
4c6f78			0	0	10012.68	0	0	10012.68	0	0	0	0	0
4b8e01	PGT95V	7073	41.2502	1.6139	11277.6	41.2502	1.6139	11277.6	206.7899	0	0	260.983	0
4ca7b9	RYR45EM	1036	41.687	4.3691	9144	41.688	4.3757	9144.7855	230.747	0	0	78.164	0.3252
4ac9aa			0	0	10012.68	0	0	10012.68	0	0	0	0	0
3410c9	VEG1		0	0	0	0	0	0	0	0	0	0	0
e0f743		3611	0	0	0	0	0	0	0	0	0	0	0
780c43	CCA842	1035	41.625	3.6574	8831.58	41.625	3.6574	8831.58	228.5342	0	0	74.4621	8.4532
44cec3			0	0	9525	0	0	9525	0	0	0	0	0
3946e0	HOP45EP	1000	43.33	5.5702	10332.7201	43.33	5.5702	10332.7201	218.1006	0	0	284.2002	9.1034
508338		7576	0	0	10058.4	0	0	10058.4	0	0	0	0	0
34258f		1004	0	0	-30.48	0	0	-30.48	0	0	0	0	0
493a2b			0	0	5623.56	0	0	5623.56	0	0	0	0	0
f862e4			0	0	2209.8	0	0	2209.8	0	0	0	0	0
4a09a4	BMS6814		42.497	5.4686	11574.78	42.497	5.4686	11574.78	236.9416	0	0	48.6971	0

Figura 5.11: Aspecto del display programado.

Parte III

Implementación

6. Implementación hardware

La implementación del *hardware* del sistema se realiza conectando los distintos componentes introducidos y explicados en 4. El proceso explicado a continuación se debe llevar a cabo en cada una de las estaciones receptoras de señales, para después poder realizar la configuración *software* de manera correcta.

En primer lugar, se debe realizar la conexión de la antena con el filtro paso-banda. Como se mencionó, se necesita un cable que funcione como adaptador para realizar la conexión entre ambos componentes, que cuentan con conectores de tipo diferente (tipo *N* en la antena y *SMA* en el filtro). Posteriormente, se conecta el filtro, mediante el conector *SMA* macho al dispositivo SDR, que cuenta con un conector *SMA* hembra. Después, el dispositivo *SDR* se conecta a la *RPi* mediante el conector *USB*.

A continuación, se debe realizar la conexión de la *RPi* a un *router* mediante un cable *Ethernet*. Esto permite la configuración posterior de la placa mediante *SSH*. Por último, la placa se conecta a una fuente de alimentación mediante un conector *micro USB* del misma, para empezar a funcionar. El cable para realizar esta conexión está incluido en el paquete de *RPi* distribuido oficialmente.

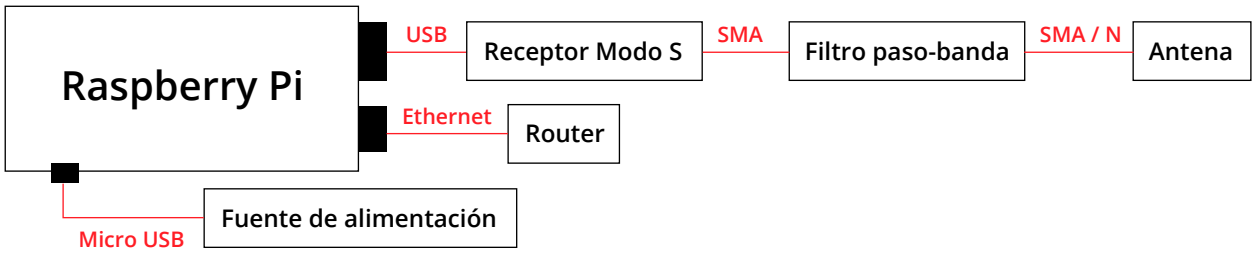


Figura 6.1: Esquema de las conexiones del sistema receptor.

Por otro lado, en lo que a la *RPi* se refiere, se necesita realizar el montaje de unos disipadores de calor que ésta incorpora. Estos disipadores son necesarios para evitar el sobrecalentamiento excesivo de la placa durante su funcionamiento. La placa cuenta con dos bases que permiten el montaje de dos disipadores, uno en cada una de ellas. Los disipadores se deben colocar con precisión en estas bases. La placa debe quedar de la forma que aparece en la imagen siguiente después de realizar el montaje (considerando que estamos usando la *Raspberry Pi B+*).

Figura 6.2: Raspberry Pi B+ con los disipadores montados.

Finalmente, en caso de contar con un dispositivo *GPS* (el que se describió en la parte del diseño), se debería realizar el montaje de éste. Sin embargo, en esta versión del proyecto aún no se ha contado con un dispositivo de este tipo, por lo que no se ha podido realizar su implementación. Ésta se deja para futuras versiones.

7. Configuración de la Raspberry Pi

7.1. Instalación de Raspbian y establecimiento de la IP

La parte central de la configuración de la RPi es la instalación de la aplicación *dump1090*, que se explicó en 5.2. Antes de proceder a ello, se debe instalar el sistema operativo sobre el que la placa va a funcionar y realizar una configuración inicial de la misma.

El sistema operativo, como ya se mencionó, es *Raspbian*. Se debe de descargar una imagen desde la página web correspondiente (*https://www.raspberrypi.org/downloads/raspbian/*), concretamente la *RASPBIAN JESSIE WITH PIXEL*. A continuación, la imagen descargada se debe montar sobre una tarjeta *Micro SD* mediante un software como *Win32DiskImager*. En la web antes mencionada aparecen las instucciones para realizar el montaje. Es extremadamente importante asegurarse de que se está realizando el montaje sobre la tarjeta *Micro SD* y no sobre otro dispositivo, pues si, por ejemplo, se realiza por error sobre el disco duro del equipo en el que se está realizando esta configuración, puede producirse un formateo del disco en cuestión. La tarjeta *Micro SD* con *Raspbian* montado se conecta a la *RPi* mediante el conector correspondiente.

A continuación, es fundamental conocer la dirección *IP* asignada a la *RPi* dentro de la red en la que se va a realizar la configuración de la misma. Lo más recomendable es fijar la dirección dentro de la red en cuestión. Sin embargo, para poder hacer esto, se necesita conocer lo que se conoce como dirección *MAC* o física, única para cada dispositivo dentro de la red, que permite identificarlo y asignarle una *IP*.

7.2. Configuración inicial e instalación de dump1090

Toda la configuración que se explica a continuación se realiza desde la consola de comandos inicializada por *SSH* en el programa "*putty*". Una vez iniciado el programa, se introduce la *IP* asignada a la placa y el puerto 22 (que es el correspondiente al protocolo *SSH*).

7.2.1. Actualización de Raspbian, selección del time zone e instalación de Git

En primer lugar, se debe actualizar *Raspbian* a la última versión. Para hacer esto, igual que para gran parte de la órdenes que se utilizarán a lo largo del proceso de configuración, se necesita incluir la palabra *sudo* antes de escribir la órdenes. Esto permite ejecutar operaciones con privilegios de seguridad. Las órdenes necesarias para las actualizaciones son:

sudo apt-get update
sudo apt-get dist-upgrade

Al ejecutar estas órdenes, el *SO* pregunta si se quiere proporcionar cierta cantidad de espacio de disco para realizar la actualización, algo que ocurrirá cada vez que se pretenda instalar algo. Es necesario dar el consentimiento.

A continuación, es posible que aparezca un mensaje en consola que informa de la posibilidad de

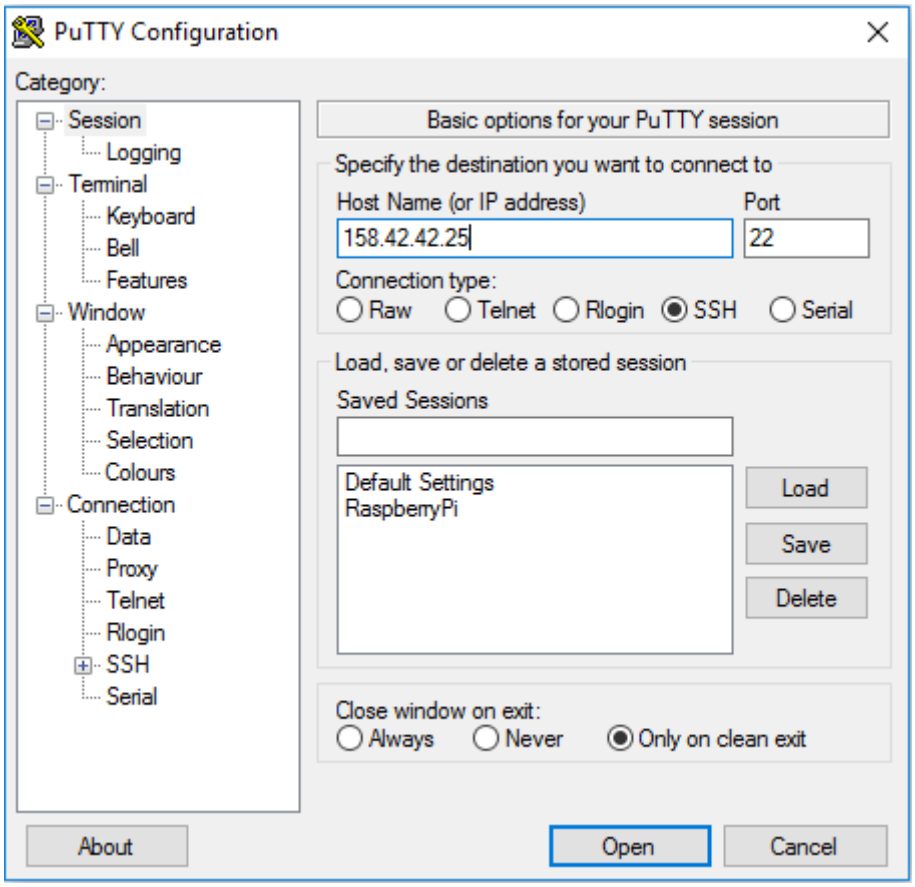


Figura 7.1: Configuración de "putty" para realizar la conexión SSH (la IP depende de la red, pero el puerto es fijo).

```
pi@raspberrypi: ~
pi@raspberrypi:~ $ ifconfig
eth0      Link encap:Ethernet  HWaddr b8:27:eb:da:79:26
          inet addr:158.42.42.25  Bcast:158.42.43.255  Mask:255.255.252.0
          inet6 addr: 2001:720:101c:43:4385:7e64:3ad4:bf0b/64 Scope:Global
          inet6 addr: fe80::3210:9984:3f7d:17f0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:25675803 errors:0 dropped:290924 overruns:0 frame:0
          TX packets:6350632 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2821117496 (2.6 GiB)  TX bytes:1162657930 (1.0 GiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:472384 errors:0 dropped:0 overruns:0 frame:0
          TX packets:472384 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:29301966 (27.9 MiB)  TX bytes:29301966 (27.9 MiB)

wlan0     Link encap:Ethernet  HWaddr b8:27:eb:8f:2c:73
          inet6 addr: fe80::daad:56e5:f250:4724/64 Scope:Link
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:804844 errors:0 dropped:804844 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:278281958 (265.3 MiB)  TX bytes:0 (0.0 B)
```

Figura 7.2: Consola de comandos de la RPi mostrando la respuesta a la orden "ifconfig".

instalar *Plymouth*, una aplicación que permite mostrar una animación en pantalla mientras tiene lugar el proceso de inicio del *SO*. No es necesario instalarlo.

Se debe realizar un *reboot* o reinicio del sistema para continuar, lo que se consigue con:

sudo reboot

También se debe comprobar si quedan actualización por llevar a cabo mediante la orden:

sudo apt-get -u update

Si el sistema indica que quedan actualizaciones, es necesario proceder a su instalación.

A continuación, debemos seleccionar el *time zone* o zona horaria en la que estamos trabajando. Esto se consigue ejecutando la siguiente orden y seleccionando *Europe - Madrid*:

sudo dpkg-reconfigure tzdata

Finalmente, se instala lo que se conoce como *Git*, que es un sistema de control de versiones de *Linux*, necesario para configurar la placa. Para ello, ejecutamos:

sudo apt-get install git-core

Es probable que no sea necesario realizar la instalación, si se tiene la última versión de *Git* instalada por defecto, en cuyo caso el sistema muestra un mensaje en la consola indicándolo.

7.2.2. Configuración del dispositivo SDR

Con el dispositivo SDR desconectado de la placa, vamos a proceder a su configuración, o lo que es lo mismo, a instalar el *driver* del dispositivo.

En primer lugar, instalamos *CMake*, una herramienta que se utiliza para controlar el proceso de compilación de código, pues necesitaremos realizar una compilación a lo largo de esta configuración. Ejecutamos entonces lo siguiente:

sudo apt-get install cmake

Ahora, instalamos *libusb*, una librería en *C* que permite obtener acceso a dispositivos *USB* desde cualquier *SO*. La necesitamos para trabajar con nuestro dispositivo *USB*:

sudo apt-get install libusb-1.0.-0.dev

El último paso para terminar el *set up* para la instalación del *driver* es instalar la herramienta *build-essential*, que contiene una lista de paquetes esenciales para la creación de paquetes *Debian*.

sudo apt-get install build-essential

A continuación, instalamos el *driver*. Para ello, realizamos una copia de los archivos alojados en *git.osmocom.org* correspondientes al driver de nuestro dispositivo *SDR* (concretamente del chip *RTL-2832U* que este contiene). Estos archivos están alojados en un directorio llamado *rtl-sdr*.

git clone git://git.osmocom.org/rtl-sdr.git

Cambiamos el directorio de trabajo en la placa a aquel que acabamos de crear mediante la orden

cd:

cd rtl-sdr

Creamos un nuevo directorio dentro de éste, llamo *build*, con la orden *mkdir*:

mkdir build

Accedemos a este directorio:

cd build

Ejecutamos la siguiente orden para configurar el dispositivo como usuario *non-root*. Hay una diferencia entre un administrador del sistema y un usuario *non-root*, pues este último tiene control total sobre el sistema, mientras el admin tiene algunas acciones restringidas.

cmake ../ -DINSTALL_UDEV_RULES=ON

Debemos usar la orden *make* para buscar qué archivos deben ser recompilados, teniendo en cuenta los cambios que hemos realizado anteriormente.

make

A continuación, se realiza la instalación como tal, en la que se copian los archivos compilados en los directorios apropiados.

sudo make install

Finalmente, se debe realizar una configuración adicional, que crea los enlaces necesarios para las nuevas librerías compartidas encontradas después de la instalación.

sudo ldconfig

Conectamos el dispositivo *SDR* a la placa y realizamos el paso anterior de nuevo. Debemos detallar los permisos que va a tener el dispositivo conectado en el sistema mediante las siguientes órdenes:

cd
sudo cp ./rtl-sdr/rtl-sdr.rules /etc/udev/rules.d/
sudo reboot

Ejecutamos la siguiente orden para ver información del dispositivo:

rtl_test

Nos aparece un mensaje de este tipo:

Found 1 device(s):
0: Generic RTL2832U
Using device 0: Generic RTL2832U
Kernel driver is active, or device is claimed by second instance of librtlsdr.
In the first case, please either detach or blacklist the kernel module
(dvb_usb_rtl28xxu), or enable automatic detaching at compile time.
usb_claim_interface error -6

Failed to open rtlsdr device #0.

Si nos aparece el error de las últimas líneas, es porque el *SO*, después de las actualizaciones que hemos realizado, tiene instalado un driver *DVB* (que permite la recepción de señal de TV). Esto interfiere con el funcionamiento del dispositivo que tenemos conectado, pues éste está usado por este *driver*. Por ello, tenemos que hacer lo siguiente.

Debemos crear un fichero con extensión *.conf* (por ejemplo, llamado *rtl.conf*) dentro de la carpeta *modprobe.d*.

cd /etc/modprobe.d
sudo nano no-rtl.conf

La orden *sudo nano* abre un editor básico de texto que permite configurar el archivo creado. Dentro de este archivo, añadimos las siguientes líneas:

blacklist dvb_usb_rtl28xxu
blacklist rtl2832
blacklist rtl2830

Hacemos *reboot* y ejecutamos *rtl_test* de nuevo, comprobando que el error ya no aparece.

7.2.3. Instalación y configuración de dump1090

Para instalar *dump1090*, cambiamos al directorio predeterminado del sistema en caso de que no nos encontremos en él:

cd /home/pi/

Copiamos el repositorio donde se encuentra alojada la aplicación desde el servidor correspondiente:

git clone git://github.com/MalcolmRobb/dump1090.git

Cambiamos al nuevo directorio:

cd dump1090

Buscamos los archivos que deben ser recompilados:

make

Ahora, *dump1090* está listo para ser ejecutado. Si todas las conexiones y la configuración de hardware descrita en 4 está hecha, deberíamos poder visualizar la información de tráfico que recibe la antena al ejecutar la siguiente orden:

./dump1090 –interactive

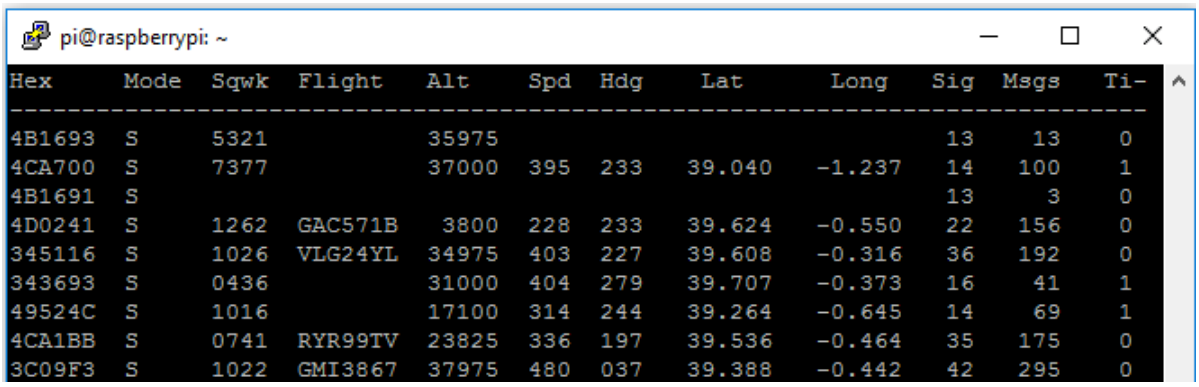


Figura 7.3: Aplicación dump1090 ejecutándose y mostrando la información de tráfico recibida.

La orden mencionada anteriormente sólo nos permite visualizar la información en consola, nada más. Lo que necesitamos es hacer que la placa envíe por un puerto determinado la información de tráfico en *Beast*. Para ello, podemos acceder a la pantalla de ayuda de la aplicación mediante la orden siguiente:

./dump1090 –help

Entre todas las órdenes posibles que permite la aplicación, localizamos la línea siguiente:

–net-bo-port ⟨port⟩ TCP Beast output listen port (default: 30005)

Ésta nos indica que el puerto utilizado por defecto por la aplicación para enviar la información *Beast* es el 30005.

Tenemos también las líneas siguientes:

–net Enable networking
–net-beast TCP raw output in Beast binary format

Estas son las órdenes que debemos añadir al ejecutar la aplicación para que ésta envíe toda la información de tráfico que recibe por el puerto correspondiente en *Beast*. Así, el sistema receptor de señales ya está montado y funcionando, y la aplicación integradora puede recibir la información proporcionada.

Para terminar, es conveniente mencionar que la pantalla de ayuda de *dump1090* muestra una gran cantidad de órdenes adicionales. Por ejemplo, se puede enviar también la información en formato *BaseStation*, cambiar la frecuencia de recepción (1090 MHz por defecto), ajustar la ganancia, etc. Para el propósito de este proyecto, estas órdenes no se han utilizado, quizá algunas de ellas puedan ser de utilidad en caso de un funcionamiento incorrecto del sistema receptor (por ejemplo, quizá la ganancia por defecto no esté ajustada correctamente y se produzca una saturación en la recepción de señales). Por ello, es conveniente explorar todas las posibilidades de configuración.

Por otro lado, todo el proceso de configuración anterior está explicado en *http://www.satsignal.eu/raspberry-pi/dump1090.html*. Las soluciones a algunos de los posibles errores que pueden surgir a lo largo de este proceso aparecen en esta web.

8. Aplicación Java

A continuación, se va a introducir y detallar la implementación en *Java* de la aplicación diseñada en 5.3. Como ya se ha explicado, se trata de la última versión en funcionamiento en el momento de redacción de la memoria y en sucesivas versiones el diseño del sistema se irá actualizando, lo que, seguramente, traerá cambios considerables en la implementación.

8.1. Paquete RadarServer

8.1.1. Clase Airplane

Clase que contiene la información de una aeronave en forma de atributos de tipo mensaje Modo S (Ver 8.3.1), un atributo de tipo *Position* (Ver 8.3.3.1), la dirección *ICAO* y la marca de tiempo del mensaje.

8.1.1.1. Declaración

public class Airplane

8.1.1.2. Constructores

- **public Airplane(String HexIdent)** - Crea un objeto con una dirección *ICAO* como parámetro. **Parámetros**
  - **String HexIdent** - Dirección *ICAO* como cadena de caracteres *ASCII*.

8.1.1.3. Atributos

- **private final String HexIdent** - Dirección *ICAO* de la aeronave.
- **private Position position** - Posición de la aeronave.
- **private AltitudeReply altRep, private IdentifyReply identRep ...** - Atributos de tipo mensaje específico Modo S.
- **private long timestamp** - Marca de tiempo del último mensaje válido recibido para la aeronave. Cuando no se indique lo contrario, una marca de tiempo estará en unidades de nanosegundo.

8.1.1.4. Métodos

Los únicos métodos de esta clase son los básicos setters (para establecer los atributos definidos) y getters (para recuperar los atributos).

8.1.2. Clase AirplaneInfo

Clase que contiene atributos de la aeronave que van a ser transmitidos. Los atributos se definen a partir de la clase *Airplane*. También contiene el estado de extrapolación de la aeronave en cuestión y *flags* que indican si se han recibido nuevos datos necesarios para extrapolación.

8.1.2.1. Declaración

public class AirplaneInfo

8.1.2.2. Constructores

- **public AirplaneInfo(String HexIdent)** - Crea un objeto con una dirección *ICAO* como parámetro. **Parámetros**
  - **String HexIdent** - Dirección *ICAO* como cadena de caracteres *ASCII*.

8.1.2.3. Atributos

- **private final String HexIdent** - Dirección *ICAO* de la aeronave.
- **private String callsign, squawk** - Dirección Modo A y *squawk* de la aeronave.
- **private double latitude, longitude, altitude, lat_ext, lon_ext, alt_ext** - Atributos de posición recibida y extrapolada.
- **private double ias, tas, gs, vr, track** - Otros datos de la aeronave.
- **private boolean vr_baro, alt_baro** - *Flags* que indican la fuente de velocidad vertical y de la altitud (barométrica o *GNSS*).
- **private long timestamp, timestamp_previous, position_timestamp** - Marca de tiempo del último mensaje, del anterior y de la última posición calculada por *CPR*.
- **private boolean reset_extrapolation, reset_alt_ext, extrapolating** - *Flags* que indican, por orden, si se debe reiniciar la extrapolación, si se ha actualizado la altitud por algún mensaje y si existe capacidad de extrapolación.
- **new_gs, new_track, new_vr** - *Flags* que indican si se han recibido nuevos valores para realizar la extrapolación.

8.1.2.4. Métodos

- **public void setTimestamp(long timestamp), public void setTimestamp_previous(long timestamp_previous), public void setPosition_timestamp(long position_timestamp)** - *Setters* de las marcas de tiempo definidas.

  Parámetros
  - **long timestamp**, **long timestamp_previous**, **long position_timestamp** - Marca de tiempo a establecer.
- **public void setCallsign(Airplane ap), public void setSquawk_IdentRep(Airplane ap), public void setGS_SurPos(Airplane ap) ...** - Todos los métodos de este tipo actualizan los atributos de la clase a partir de un atributo de tipo mensaje Modo S de un objeto de la clase *Airplane*. Así, por ejemplo, la altitud puede ser actualizada a partir de una multitud de mensajes (*Short ACAS*, *Long ACAS*, *Altitude Reply*, etc.), por lo que tendrá un método por cada tipo de mensaje del que se puede obtener la altitud.

Parámetros

- **Airplane ap** - Instancia de la clase *Airplane*.

- *Los demás métodos son los getters y setters habituales.*

8.1.3. Clase PositionExtrapolator

Clase cuya instancia permite la extrapolación de posición de una aeronave a partir de cierta información conocida.

8.1.3.1. Declaración

public class PositionExtrapolator

8.1.3.2. Constructores

- **public PositionExtrapolator()** - Constructor por defecto.

8.1.3.3. Atributos

- **private long timestamp, ts_seconds, ts_position_seconds** - En orden, marca de tiempo del último mensaje de la aeronave cuya posición se va a extrapolar, la misma marca de tiempo en segundos y marca de tiempo de la última posición *CPR* en segundos.
- **private double gs, track, vr, lat_ex, lon_ex, alt_ex ...** - Datos necesarios para realizar la extrapolación. Algunos de ellos son atributos que se definen temporalmente en el algortimo de la loxodrómica para realizar el cálculo.

8.1.3.4. Métodos

- **public void extrapolate(long timestamp, AirplaneInfo ap2)** - Método que realiza la extrapolación. Para comprender su funcionamiento, ver 5.3.3.1.

  Parámetros

  - **long timestamp** - Marca de tiempo del último mensaje para la aeronave que se va a extrapolar.
  - **AirplaneInfo ap2** - Instancia de tipo *AirplaneInfo* de la aeronave que se va a extrapolar.

8.1.4. Clase TrafficAll

Clase que contiene los mapas de tráfico, el mapa para decodificación de posición, el método que realiza la decodificación y el que pasa la información del mapa de datos a un *array* de *bytes*. Además, en esta clase se realiza la extrapolación.

8.1.4.1. Declaración

public class TrafficAll

8.1.4.2. Constructores

- **public TrafficAll()** - Constructor que instancia los mapas y el extrapolador.

8.1.4.3. Atributos

- **private ConcurrentHashMap ⟨String, Airplane⟩ TrafficMap** - Mapa de mensajes.
- **private ConcurrentHashMap ⟨String, AirplaneInfo⟩ Airplanes** - Mapa de datos.
- **private PositionDecoder posDec** - Atributo de tipo decodificador de posición (Ver 8.3.3.2).
- **private ConcurrentHashMap ⟨String, PositionDecoder⟩ posDecMap** - Mapa de decodificadores de posición.
- **private PositionExtrapolator posEx** - Extrapolador de posición.
- **private ModeSReply modes** - Atributo de tipo mensaje Modo S general (Ver 8.3.1).
- **private long ts_seconds** - Marca de tiempo del mensaje recibido en segundos.
- **public static long last_ts** - Último *timestamp*, el anterior al recibido en el mensaje que se está decodificando.
- **private Airplane ap** - Atributo de tipo *Airplane*.
- **private AirplaneInfo ap2** - Atributo de tipo *AirplaneInfo*.
- **private String icao24** - Dirección *ICAO* contenida en el mensaje recibido.

8.1.4.4. Métodos

- **public ConcurrentHashMap ⟨String, Airplane⟩ getTrafficMap()** - *Getter* del mapa de mensajes.
- **public ConcurrentHashMap ⟨String, AirplaneInfo⟩ getAirplanes()** - *Getter* del mapa de datos.
- **public void updateTrafficMap(long timestamp, String raw_message) throws BadFormatException, UnspecifiedFormatException, MissingInformationException** - Método decodificador que actualiza los mapas de tráfico con el mensaje *ASCII* recibido. Para entender el funcionamiento del decodificador, ver 5.3.3. Cuando se utiliza este método, los mapas de tráfico se sincronizan.

  Parámetros

  - **long timestamp** - Marca de tiempo extraída del mensaje.
  - **String raw_message** - Mensaje Modo S en forma de cadena de caracteres *ASCII*.

  Excepciones

- **BadFormatException** - Ver 8.3.2.1.
- **UnspecifiedFormatException** - Ver 8.3.2.4.
- **MissingInformationException** - Ver 8.3.2.2.

- **public byte[] getByteArray()** - Método que escribe la información del mapa de datos en un *array* de *bytes*, con la estructura definida en 5.3.5.3. A la hora de recorrer el mapa de datos y escribir en el flujo de salida, el mapa se sincroniza.

  Salida

  - **byte[] byte_array** - El *array* de *bytes*.

8.1.5. Clase ReceiverThread

Clase que ejecuta el hilo responsable de recibir los mensajes de una antena en concreto y pasarlos al decodificador.

8.1.5.1. Declaración

public class ReceiverThread extends Thread

8.1.5.2. Constructores

- **public ReceiverThread(TrafficAll tm, String host, int port) throws IOException** - Crea y ejecuta un hilo a partir de un objeto de tipo *TrafficAll* y la *IP* y puerto de la máquina de la que se va a recibir la información.

  Parámetros

  - **TrafficAll tm** - Objeto de tipo *TrafficAll* sobre el que se va a trabajar.
  - **String host** - La dirección *IP* de la máquina a la que se va a realizar la conexión.
  - **int port** - Puerto al que se va a realizar la conexión.

  Excepciones

  - **IOException** - Se lanza cuando hay algún problema relacionado con la conexión a la máquina de la que se quiere recibir información o cuando hay un error al leer la información del flujo de datos transmitido.

8.1.5.3. Atributos

- **private Socket phone** - *Socket* que se crea para escuchar la información que llega.
- **private DataInputStream dis** - Flujo de datos de tipo primitivo (Ver 5.3.5.2) generado a partir del flujo transmitido por el *socket*.
- **private byte[] initial_bytes** - *Array* de *bytes* correspondiente al identificador de inicio de un mensaje *Beast* (para entender la estructura de un mensaje *Beast*, ver 5.1.1).
- **private byte[] type_byte** - *Array* de *bytes* correspondiente al identificador de tipo de un mensaje *Beast*.
- **timestamp_bytes** - *Array* de *bytes* correspondiente al *timestamp* contenido en el mensaje *Beast*.
- **short_message_bytes** - *Array* de *bytes* correspondiente a un mensaje Modo S corto.
- **long_message_bytes** - *Array* de *bytes* correspondiente a un mensaje Modo S largo.
- **private ByteBuffer bb_timestamp** - *Buffer*[1] de *bytes* en el que se almacena el *timestamp* de un mensaje, que permite la extracción del *timestamp* en formato de un atributo de tipo *long*.
- **private ByteBuffer bb_message_short** - *Buffer* de *bytes* en el que se almacena un mensaje Modo S corto.
- **private ByteBuffer bb_message_long** - *Buffer* de *bytes* en el que se almacena un mensaje Modo S largo.
- **private long timestamp** - Marca de tiempo contenida en el mensaje *Beast*.

[1] Espacio para el almacenamiento temporal de información.

- **private long ns_per_tick** - Factor de conversión para pasar el timestamp en *Beast* a nanosegundos (Ver 5.1.1.0.2).
- **private String short_message, long_message** - Mensajes Modo S corto y largo como atributos de tipo *String* (caracteres *ASCII*).
- **private String host** - *IP* de la máquina a la que se realiza la conexión como atributo de tipo *String*.
- **public TrafficAll tm** - Objeto de tipo *TrafficAll* sobre el que se va a trabajar.

8.1.5.4. Métodos

- **public void run()** - Método que se ejecuta mientras *TrafficDecoder.running = true* (Ver 8.1.9.3) y se encarga de leer *byte* a *byte* la información que llega en el *input stream*. Identifica las distintas partes del mensaje *Beast* para detectar cuándo se ha recibido un mensaje válido y se encarga de ejecutar, en tal caso, el método *public void updateTrafficMap(long timestamp, String raw_message)* de la clase *TrafficAll* (de manera sincronizada). Se captura la excepción *IOException*, en cuyo caso se cierra la conexión (el *socket*) y el sistema deja de funcionar, es decir, *TrafficDecoder.running = false*. Por otro lado, también se capturan las excepciones de error de formato, pero en esos casos el programa sigue funcionando, pues el hecho de que se haya recibido un mensaje inválido no debería ser razón para detener la aplicación.

8.1.6. Clase CleanerThread

Clase que ejecuta el hilo responsable de eliminar de las listas de tráfico las aeronaves que pasan a encontrarse fuera de la cobertura del sistema.

8.1.6.1. Declaración

public class CleanerThread extends Thread

8.1.6.2. Constructores

- **public CleanerThread(TrafficAll tm)** - Crea y ejecuta un hilo a partir de un objeto de tipo *TrafficAll*.

  Parámetros

  - **TrafficAll tm** - Parámetro de tipo *TrafficAll*.

8.1.6.3. Atributos

- **public TrafficAll tm** - Objeto de tipo *TrafficAll* sobre el que se va a trabajar.

8.1.6.4. Métodos

- **public void run()** - Mientras se cumpla que *TrafficDecoder.running = true*, cada segundo, ejecuta el método *public void clean_lost_flights()* (de manera sincronizada). Aquí se captura la excepción *InterruptedException*, que se lanza cuando se interrumpe la espera de 1 segundo entre ejecuciones del método anteriormente mencionado. Al capturar esta excepción, el programa sigue funcionando.
- **public void clean_lost_flights()** - Se encarga de eliminar las aeronaves que pasan a estar fuera de cobertura de las listas de tráfico. Para entender el funcionamiento del limpiador de tráfico, ver 5.3.4.

8.1.7. Clase RadarServer

Clase cuya instancia escucha peticiones de conexión al servidor del sistema y se encarga de instanciar la clase que crea el hilo para enviar la información al cliente.

8.1.7.1. Declaración

public class RadarServer

8.1.7.2. Constructores

- **public RadarServer(int port, TrafficAll ta)** - Crea el *ServerSocket* (Ver 5.3.5.1).

  Parámetros

  - **int port** - Puerto por el que el servidor envía la información a los clientes.
  - **TrafficAll ta** - Parámetro de tipo *TrafficAll*.

8.1.7.3. Atributos

- **private ServerSocket serverSocket** - Atributo de tipo *ServerSocket* que se crea sobre el puerto del servidor para escuchar peticiones de conexión.
- **private int port** - Puerto sobre el que se crea el *ServerSocket*.
- **private Socket s** - *Socket* que se crea cuando se acepta una petición de conexión, para enviar la información por el mismo.
- **public TrafficAll ta** - Objeto de la clase *TrafficAll* sobre el que se va a trabajar.

8.1.7.4. Métodos

- **public void listen() throws IOException** - Método con el que se escuchan peticiones mientras *TrafficDecoder.running = true*. Cuando se acepta una conexión, se crea un hilo que envía la información. Cuando el método deja de escuchar, el *ServerSocket* se cierra.

  Excepciones

  - **IOException** - Se lanza cuando se intenta cerrar el *ServerSocket* cuando el método ha dejado de funcionar, pero ha ocurrido un problema al cerrarlo (por ejemplo, porque el objeto en cuestión nunca se ha creado).

8.1.8. Clase ServerThread

Clase que crea el hilo que se va a encargar de enviar la información a los clientes.

8.1.8.1. Declaración

public class ServerThread extends Thread

8.1.8.2. Constructores

- **public ServerThread(Socket s, TrafficAll ta) throws IOException** - Crea el flujo de datos a transmitir por el *socket* creado sobre el puerto del servidor. El flujo en cuestión permite escribir sobre él tipos de datos primitivos.

  Parámetros

  - **Socket s** - *Socket* que se crea sobre el puerto del servidor.
  - **TrafficAll ta** - Parámetro de tipo *TrafficAll*.

  Exceptions

  - **IOException** - Se lanza cuando no se ha podido obtener el *output stream* del *socket* creado.

8.1.8.3. Atributos

- **private Socket s** - *Socket* creado sobre el puerto del servidor.
- **private DataOutputStream dos** - *Output stream* sobre el que se escribe la información enviada al cliente.
- **public TrafficAll ta** - Objeto de tipo *TrafficAll* sobre el que se va a trabajar.
- **private byte[] byte_array** - *Array* de *bytes* que se escribe sobre el *output stream*.

8.1.8.4. Métodos

- **public void run()** - Se ejecuta mientras *TrafficDecoder.running = true* y se encarga de obtener el *array* de *bytes* de información y escribirlo en el *output stream* que va a ser enviado al cliente.

8.1.9. Clase TrafficDecoder

Clase que se encarga de crear todas las instancias y ejecutar los métodos necesarios para el funcionamiento de la aplicación integradora completa, junto al servidor.

8.1.9.1. Declaración

public class TrafficDecoder(String[] host, int port)

8.1.9.2. Constructores

- **public TrafficDecoder(String[] host, int[] port) throws IOException** - Pone *TrafficDecoder.running = true*, crea un hilo por cada antena receptora, genera el limpiador de tráfico y el servidor.

  Parámetros

  - **String[] host** - *Array* que contiene las direcciones *IP* de los equipos encargados de recibir y transmitir la información que reciben las antenas.
  - **int[] port** - *Array* que contiene los puertos por los que los equipos encargados de recibir y transmitir la información que reciben las antenas.

  Excepciones

  - **IOException** - Ya se han visto los motivos por los que se puede lanzar esta excepción en cada una de las partes de la aplicación.

8.1.9.3. Atributos

- **TrafficAll trafficAll** - Objeto de tipo *TrafficAll* sobre el que el sistema va a trabajar.
- **RadarServer radarServer** - Objeto de tipo *RadarServer*, crea el servidor.
- **public static boolean running** - *Flag* que indica si el sistema se está ejecutando. Condiciona el funcionamiento de todos los hilos del sistema.

8.1.9.4. Métodos

- **public static void main(String[] args) throws IOException** - Crea una instancia de la clase *TrafficDecoder*.

  Excepciones

  - **IOException** - Ya hemos visto los motivos por los que se puede lanzar esta excepción. En caso de lanzarse, se captura y se hace *TrafficDecoder.running = false*.

8.2. Paquete TrafficDisplay

8.2.1. Clase TrafficMapDisplay

Clase que genera la representación de la información recibida desde el servidor.

8.2.1.1. Declaración

public class TrafficMapDisplay extends JFrame implements Runnable

8.2.1.2. Constructores

- **public TrafficMapDisplay(InetAddress host, int port) throws IOException** - Crea el *socket* para recibir la información del servidor y ejecuta el método que la lee línea a línea y actualiza la tabla del *display*. Genera también la interfaz (instancia de la clase *JFrame*) sobre la que se va a colocar la tabla. Como *TrafficMapDisplay* es de tipo *runnable*[2], se debe crear un hilo dentro de este constructor para después ejecutarlo.

  Parámetros

  - **InetAddress host** - Dirección *IP* del servidor, en forma de instancia de la clase *InetAddress*.
  - **int port** - Puerto del servidor por el que éste envía la información.

  Excepciones

  - **IOException** - Se lanza cuando hay un problema relacionado con la conexión con el servidor. Si se captura, el *display* deja de funcionar.

8.2.1.3. Atributos

- **private Socket s** - *Socket* que se crea en el servidor para recibir la información de éste.
- **private BufferedReader dis** - Objeto que permite leer la información que llega en el flujo de datos del servidor línea a línea.
- **private InetAddress host** - Atributo que corresponde a la *IP* del servidor.
- **private int port** - Atributo que corresponde al puerto del servidor.
- **private TrafficTable att** - Tabla que crea el *display* y que se va actualizando conforme se lee nueva información.
- **private String message_line** - Atributo al que se asigna cada línea de mensaje leída.
- **private String nAirString** - Atributo que representa el número de aeronaves contenidas en el bloque de información enviado por el servidor.
- **private String[] icao24, callsign, squawk ...** - Arrays que contienen cada uno de los datos de las aeronaves contenidas en el bloque de información enviado por el servidor.

8.2.1.4. Métodos

- **public void run()** - Método que se encarga de leer la información del flujo de datos entrante línea de línea, actualizar los atributos correspondientes y ejecutar el método que actualiza la información que se pasa a la tabla.
- **public static void main(String[] args** - Crea una instancia de la clase *TrafficMapDisplay*.

[2]Interfaz que da a la clase la capacidad de ser ejecutable.

8.2.2. Clase TrafficTable

Clase que permite generar una tabla con información y actualizar ésta de forma continua.

8.2.2.1. Declaración

public class TrafficTable extends AbstractTableModel

8.2.2.2. Constructores

- **public TrafficTable()** - Constructor por defecto, que corresponde al de la clae *AbstractTableModel*. Crea una tabla genérica y se deben sobreescribir una serie de métodos para indicar la información que ésta debe contener.

8.2.2.3. Atributos

- **private static final String[] columnNames** - *Array* que contiene los nombres de las columnas de la tabla.
- **private static final Class[] columnsClasses** - *Array* que contiene las clases de los objetos contenidos en cada columna de la tabla.
- **private String [] icao24, callsing, squawk ...** - *Arrays* que contienen la información para cada columna de la tabla.
- **private String nAirString** - Número de aeronaves representadas en la tabla o lo que es lo mismo, número de filas de la misma.

8.2.2.4. Métodos

- **public void updateCurrentTraffic(String nAirString, String[] icao24, String[] squawk, String[] callsign ...)** - Método que actualiza los atributos que corresponden a los valores contenidos en la tabla.

  Parámetros

  - **String nAirString, String[] icao24, String[] squawk, String[] callsign ...** - Parámetros para actualizar los valores contenidos en la tabla.

- **public int getRowCount()** - Método que devuelve el número de filas de la tabla.

  Salida

  - **int nAirInt** - Número de filas de la tabla.

- **public int getColumnCount()** - Método que devuelve el número de columnas de la tabla.

  Salida

  - **int columnNames.length** - Número de columnas de la tabla.

- **public String getColumnName(int col)** - Método que devuelve el título de una columna concreta.

  Parámetros

  - **int col** - Número de la columna cuyo título se quiere obtener.

  Salida

  - **String columnNames[col]** - Nombre de la columna.

- **public Class getColumnClass(int col)** - Método que devuelve la clase de los objetos contenidos en una columna determinada.

  Parámetros

  - **int col** - Número de la columna cuya clase se quiere obtener.

  Salida

  - **Class columnNames[col]** - Clase de la columna.

- **public Object getValueAt(int row, int col** - Método que devuelve el valor de la tabla localizado en una celda concreta.

  Parámetros

  - **int row** - Número de la fila cuya información se quiere obtener.
  - **int col** - Número de la columna cuya información se quiere obtener.

  Salida

  - **String[] icao24[row], callsign[row], squawk[row] ...** - Valor en la celda correspondiente.

8.3. Paquetes OpenSky

Estos paquetes contienen las clases que permiten la decodificación de mensajes. Es decir, contienen los tipos de mensajes Modo S definidos como clases, el algoritmo de decodificación de posición y las excepciones que pueden ser lanzadas en la decodificación. Se trata de unos paquetes de uso libre, bajo los términos establecidos por *GNU Public License*. El autor del código contenido en estos paquetes es *Matthias Schäfer (schaefer@opensky-network.org)*. Para este proyecto, se ha obtenido la última versión de estos paquetes, pero es probable que vayan apareciendo nuevas versiones, sobre todo para aportar robustez a la aplicación e incluir los mensajes que, de momento, no son completamente decodificables (como los *Comm-B* y *Comm-D*).

8.3.1. Paquete org.opensky.libadsb.msgs

Este paquete contiene definidas las clases correspondientes con cada uno de los tipos de mensajes codificables. Es decir, tenemos una clase (mensaje) que tiene unos atributos (contenido del mensaje) y unos métodos (*getters*, *setters* u otros métodos para calcular datos a partir de los atributos). Se vio en 5.3.3 que el proceso de decodificación se realiza en varias fases. Es decir, primero se crea un objeto de tipo mensaje Modo S general y después se van instanciando clases más específicas que heredan de esta clase general, correspondientes a los tipos de mensajes Modo S. Para *ES* hay un nivel de decodificación adicional. Vamos a analizar esta clase Modo S general (*ModeSReply*). Las clases correspondientes a los diferentes tipos de mensajes Modo S siguen una estructura similar, conforme a lo establecido en 3.4, por lo que no es necesario explicarlas.

8.3.1.1. Clase ModeSReply

Clase que se instancia cuando se recibe un mensaje Modo S válido.

8.3.1.1.1 Declaración

public class ModeSReply implements Serializable[3]

8.3.1.1.2 Constructores

- **public ModeSReply()** - Constructor por defecto.
- **public ModeSReply(String raw_message) throws BadFormatException** - Define los atributos generales de un mensaje Modo S (Tabla 3.3). Obtiene la dirección *ICAO* de manera diferente dependiendo del tipo de mensaje (3.4.1.3).

  Parámetros

  - **String raw_message** - Mensaje Modo S en forma de cadena de caracteres *ASCII*.

  Excepciones

  - **BadFormatException** - Se lanza cuando el mensaje Modo S tiene una longitud distinta a la propia de los mensajes Modo S (14 o 28 caracteres) o cuando no tiene la longitud esperada para el *Downlink Format* contenido en el mensaje.
- **public ModeSReply(ModeSReply reply)** - Constructor que copia una instancia de *ModeSReply* creada.

  Parámetros

  - **ModeSReply reply** - Instancia de la clase *ModeSReply*.

8.3.1.1.3 Atributos

- **private byte downlink_format** - *DF* del mensaje como atributo de tipo *byte*.
- **private byte first_field** - Campo que contiene información diferente dependiendo del tipo de mensaje.
- **private byte[] icao24** - Dirección *ICAO*.
- **private byte[] payload** - Carga de pago del mensaje, variable dependiendo del tipo de mensaje.
- **private byte[] parity** - Campo de paridad del mensaje.
- **public static enum subtype** - Lista enumerada de atributos, cada uno correspondiente a un tipo de mensaje, para indicar qué mensaje es el que está codificado.
- **private subtype type** - Indicador de tipo de mensaje.
- **public static final byte[] CRC_polynomial** - Polinomio usado para el cálculo de paridad.

[3]Interfaz que permiten a los objetos de la clase ser convertidos a bytes, para, por ejemplo, ser transmitidos por la red, y después ser recuperados y reconstruidos al otro lado de la red.

8.3.1.1.4 Métodos

- **public static byte[] calcParity(byte[] msg)** - Método que calcula la paridad del mensaje recibido.

  Parámetros

  - **byte[] msg** - Parte del mensaje recibido usada para el cálculo.

  Salida

  - **byte[] Arrays.copyOf(pi, CRC_polynomial.length)** - La paridad calculada.

- **public static int getExpectedLength(byte downlink_format)** - Método que devuelve la longitud esperada para el *DF* del mensaje.

  Parámetros

  - **byte downlink_format** - *DF* del mensaje.

  Salida

  - **int length** - Longitud (7 o 14) dependiendo del *DF*.

- **public subtype getType()** - Método que devuelve el atributo de tipo de mensaje, uno de los de la lista enumerada.

  Salida

  - **subtype type** - Tipo de mensaje.

- **protected void setType(subtype subtype)** - *Setter* del atributo de tipo de mensaje.

- **public byte getDownlinkFormat(), protected byte getFirstField(), public byte[] getIcao24(), public byte getPayload(), public byte[] getParity()** - *Getters* de la clase.

- **public byte[] calcParity()** - Método que llama a *public static byte[] calcParity(byte[] msg)* para calcular la paridad.

  Salida

  - **byte[] calcParity(byte[] message)** - La paridad calculada.

- **public boolean checkParity()** - Método que hace la comprobación de paridad necesaria para determinar la validez de ciertos mensajes.

  Salida

  - **boolean tools.areEqual(calcParity(), getParity())** - Booleano que indica si la paridad calculada y el campo de paridad del mensaje coinciden.

- **public String toString()** - Método que devuelve la información contenida en el mensaje como cadena de caracteres.

  Salida

  - **String "Mode S Reply:\n " + " \t Downlink format:\t " + getDownlinkFormat() + ...** - Cadena de caracteres con la información del mensaje.

- **public boolean equals(Object o)** - Se sobreescribe el método (de la clase *Object*) que compara dos instancias de esta clase y devuelve un booleano que indica si los objetos son iguales.

  Parámetros

  - **Object o** - Instancia a comparar con el objeto actual.

  Salida

  - **boolean result** - Booleano que indica si los objetos comparados son iguales o no.

- **public int hashCode()** - Se sobreescribe el método que devuelve el *Hash Code*[4] asignado a una instancia de esta clase.

  Salida

  - **int sum** - *Hash Code* de la instancia.

[4]Se trata de un identificador de 32 bits que cada instancia de una clase tiene asignado. La clase Object asigna un Hash Code por defecto.

8.3.2. Paquete org.opensky.libadsb.exceptions

Este paquete contiene definidas las excepciones que puede lanzar el decodificador (las que no están definidas por defecto en *Java*).

8.3.2.1. Clase BadFormatException

Excepción relacionada con un formato de mensaje incorrecto. Se lanza, por ejemplo, cuando se intenta decodificar un mensaje Modo S que no tiene una longitud adecuada.

8.3.2.1.1 Declaración

public class BadFormatException extends Exception

8.3.2.1.2 Constructores

- **public BadFormatException(String reason, String message)** - Crea la excepción a partir de un mensaje y el motivo por el que ésta se ha creado.

  Parámetros

  - **String reason** - Razón de generación de la excepción.
  - **String message** - Mensaje por el que se ha generado la excepción.

- **public BadFormatException(String reason)** - Crea la excepción a partir de una razón, pero no un mensaje.

  Parámetros

  - **String reason** - Razón de generación de la excepción.

8.3.2.1.3 Atributos

- **private String msg** - Mensaje Modo S en forma de cadena de caracteres *ASCII*.
- **private String reason** - Atributo asignado al motivo por el que se ha generado una excepción, en forma de cadena de caracteres.

8.3.2.1.4 Métodos

- **public String getMessage()** - Método sobreescrito que devuelve la descripción de la excepción.

  Salida

  - **String "Message " + this.msg + " has an illegal format: " + this.reason** - Descripción de la excepción.

8.3.2.2. Clase MissingInformationException

Excepción que se lanza cuando se llama al *getter* de la clase correspondiente a un tipo de mensaje pero la información que devuelve el *getter* no está disponible.

8.3.2.2.1 Declaración

public class MissingInformationException extends Exception

8.3.2.2.2 Constructores

- **public MissingInformationException(String reason)** - Genera la excepción conociendo la razón por la que ésta se debe generar. **Parámetros**
  - **String reason** - Motivo por el que se ha generado la excepción en forma de cadena de caracteres.

8.3.2.3. Clase PositionStraddleError

Excepción que se lanza cuando ocurre algún tipo de incompatibilidad a la hora de calcular la posición por el algoritmo *CPR*.

8.3.2.3.1 Declaración

public class PositionStraddleError extends Exception

8.3.2.3.2 Constructores

- **public PositionStraddleError(String reason)** - Genera la excepción conociendo la razón por la que ésta se debe generar. **Parámetros**
  - **String reason** - Motivo por el que se ha generado la excepción en forma de cadena de caracteres.

8.3.2.4. Clase UnspecifiedFormatError

Excepción que se lanza cuando un mensaje Modo S se pasa al decodificador equivocado.

8.3.2.4.1 Declaración

public class UnspecifiedFormatError extends Exception

8.3.2.4.2 Constructores

- **public UnspecifiedFormatError(String reason)** - Genera la excepción conociendo la razón por la que ésta se debe generar. **Parámetros**
  - **String reason** - Motivo por el que se ha generado la excepción en forma de cadena de caracteres.

8.3.3. Paquete org.opensky.libadsb (general)

8.3.3.1. Clase Position

Clase que representa la posición de una aeronave.

8.3.3.1.1 Declaración

public class Position implements Serializable

8.3.3.1.2 Constructores

- **public Position()** - Constructor por defecto, no define los atributos de posición (o los define como inválidos[5]).
- **public Position(Double lon, Double lat, Double alt)** - Constructor que define los atributos de posición.

8.3.3.1.3 Atributos

- **private Double longitude** - Longitud.
- **private Double latitude** - Latitud.
- **private Double altitude** - Altitud.
- **private boolean reasonable** - *Flag* que indica si el decodificador considera que la posición decodificada es razonable.

8.3.3.1.4 Métodos

- **public Double getLongitude(), public void setLongitude(Double longitude) ...** - *Setters* y *getters* de la clase.
- **public String toString()** - Método sobreescrito que pasa a cadena de caracteres la información de la instancia.

  Parámetros

  - **String ReflectionToStringBuilder.toString(this)** - Información de la instancia en forma de cadena de caracteres.

- **public boolean equals(Object o)** - Se sobreescribe el método (de la clase *Object*) que compara dos instancias de esta clase y devuelve un booleano que indica si los objetos son iguales.

  Parámetros

  - **Object o** - Instancia a comparar con el objeto actual.

  Salida

  - **boolean result** - Booleano que indica si los objetos comparados son iguales o no.

- **public int hashCode()** - Se sobreescribe el método que devuelve el *Hash Code* asignado a una instancia de esta clase.

  Salida

  - **int sum** - *Hash Code* de la instancia.

[5] Esto quiere decir que los define como "*null*".

- **public Double distanceTo(Position other** - Método que calcula la distancia ortodrómica entre dos puntos sobre la superficie terrestre.

  Parámetros

  - **Position other** - Posición a la que se calcula la distancia.

  Salida

  - **Double distance** - Distancia entre los dos puntos.

8.3.3.2. Clase PositionDecoder

Clase para decodificación de la posición, cuyas instancias están contenidas en un *hash map* (para entender el funcionamiento del *hash map* en cuestión, ver 5.3.1).

8.3.3.2.1 Declaración

public class PositionDecoder

8.3.3.2.2 Constructores

- **public PositionDecoder()** - Constructor por defecto, no define los atributos (o los define como inválidos).

8.3.3.2.3 Atributos

- **private AirbornePositionMsg last_even_airborne** - Atributo correspondiente al último mensaje *Airborne Position* de codificación *CPR* par.
- **private AirbornePositionMsg last_odd_airborne** - Atributo correspondiente al último mensaje *Airborne Position* de codificación *CPR* impar.
- **private SurfacePositionMsg last_even_surface** - Atributo correspondiente al último mensaje *Surface Position* de codificación *CPR* par.
- **private SurfacePositionMsg last_odd_surface** - Atributo correspondiente al último mensaje *Surface Position* de codificación *CPR* impar.
- **private Position last_pos** - Atributo correspondiente a la última posición válida calculada.
- **private double last_even_airborne_time, private double last_odd_airborne_time ...** - Marcas de tiempo de todos los atributos anteriores.
- **private boolean supplA, supplC** - Atributos correspondientes a los suplementos *NIC*, contenidos en el mensaje *Operational Status ES* (3.4.7.12).
- **num_reasonable** - Número de mensajes "razonables"sucesivos, es decir, aquellos que hacen *reasonable = true* en en el objeto de tipo *Position* correspondiente a la posición de la aeronave.
- **private static final int MAX_DIST_TO_SENDER** - Distancia máxima entre un emisor de mensajes (aeronave) y el receptor (se establece en 700km).

8.3.3.2.4 Métodos

- **static boolean withinThreshold (double timeDifference, double distance, boolean surface)** - Método que se utiliza para determinar si la diferencia de tiempo entre recepción de mensajes sucesivos es realista para la distancia que la aeronave ha recorrido.

  Parámetros

  - **double timeDifference** - Diferencia de tiempo entre la recepción de mensajes.
  - **double distance** - Distancia recorrida.
  - **boolean surface** - *Flag* que indica si la aeronave está o no en tierra.

  Salida

  - **boolean result** - Booleano que indica si la distancia es realista.

- **public static boolean withinReasonableRange(Position receiver, Position sender)** - Método que comprueba si la distancia entre la aeronave que ha enviado un mensaje y el receptor del mensaje es razonable.

  Parámetros

  - **Position receiver** - Posición del receptor.
  - **Position sender** - Posición del emisor.

  Salida

  - **receiver.distanceTo(sender)⟨=MAX_DIST_TO_SENDER** - Booleano que indica si la distancia es razonable.

- **public Position decodePosition(double time, AirbornePositionMsg msg)** - Método que calcula la posición de la aeronave a partir de un mensaje *Airborne Position ES*.

  Parámetros

  - **double time** - Marca de tiempo del mensaje.
  - **AirbornePositionMsg msg** - Mensaje recibido.

  Salida

  - **Position ret** - Posición calculada. Puede ser calculada de forma global o local. Puede ser inválida si no se cumplen las condiciones necesarias, como ya se vio.

- **public Position decodePosition(AirbornePositionMsg msg)** - Método similar al anterior pero sin una marca de tiempo para el mensaje, sino que se utiliza el tiempo actual del sistema donde se ejecuta la aplicación.

  Parámetros

  - **AirbornePositionMsg msg** - Mensaje recibido.

  Salida

  - **Position ret** - Posición calculada. Puede ser calculada de forma global o local. Puede ser inválida si no se cumplen las condiciones necesarias, como ya se vio.

- **public Position decodePosition(double time, Position receiver, AirbornePositionMsg msg)** - Método que usa el método anterior para decodificar la posición y comprueba si la nueva posición es razonable en lo que a distancia al receptor se refiere.

  Parámetros

  - **double time** - Marca de tiempo del mensaje.
  - **Position receiver** - Posición del receptor.
  - **AirbornePositionMsg msg** - Mensaje recibido.

  Salida

  - **Position ret** - Posición calculada y comprobada en lo que a distancia al receptor se refiere.

- **public Position decodePosition(Position receiver, AirbornePositionMsg msg)** - Método similar al anterior, pero que no tiene una marca de tiempo para el mensaje, sino que utiliza el tiempo actual del sistema donde se ejecuta la aplicación.

  Parámetros

- **Position receiver** - Posición del receptor.
- **AirbornePositionMsg msg** - Mensaje recibido.

Salida

- **Position ret** - Posición calculada y comprobada en lo que a distancia al receptor se refiere.

- **public Position decodePosition(double time, SurfacePositionMsg msg, Position ref)** - Método que decodifica la posición a partir de un mensaje *Surface Position ES*.

Parámetros

- **double time** - Marca de tiempo del mensaje.
- **SurfacePositionMsg msg** - Mensaje recibido.
- **Position ref** - Posición de referencia para resolver ambiguedades en el cálculo de posición. Si no hay ninguna posición como tal, el decodificador utilizará la última posición calculada.

Salida

- **Position ret** - Posición calculada, que puede ser calculada de forma global o local. También puede ser inválida si no se cumplen las condiciones necesarias.

- **public Position decodePosition(double time, SurfacePositionMsg msg)** - Método similar al anterior, pero que no utiliza ninguna posición de referencia, sino la última posición válida.

Parámetros

- **double time** - Marca de tiempo del mensaje.
- **SurfacePositionMsg msg** - Mensaje recibido.

Salida

- **Position ret** - Posición calculada.

- **public Position decodePosition(SurfacePositionMsg msg, Position reference)** - Método similar a los anteriores, pero que no usa la marca de tiempo del mensaje sino el tiempo actual del sistema donde se ejecuta la aplicación.
Parámetros

- **SurfacePositionMsg msg** - Mensaje recibido.
- **Position reference** - Posición de referencia.

Salida

- **Position ret** - Posición calculada.

- **public Position decodePosition(double time, Position receiver, SurfacePositionMsg msg, Position reference)** - Método similar a los anteriores. Utiliza la marca de tiempo del mensaje y además realiza la comprobación de razonabilidad en lo que a distancia al receptor se refiere.
Parámetros

- **double time** - Marca de tiempo del mensaje.
- **Position receiver** - Posición del receptor.
- **SurfacePositionMsg msg** - Mensaje recibido.
- **Position reference** - Posición de referencia.

Salida

  - **Position ret** - Posición calculada y con razonabilidad comprobada.

- **getNICSupplementA(), setNICSupplementA() ...** - *Getters* y *setters* de la clase.

8.3.3.3. Clase tools

Clase que cuenta con algunos métodos utilizados en la decodificación de mensajes.

8.3.3.3.1 Declaración

public class tools

8.3.3.3.2 Constructores

- **public tools()** - Constructor por defecto. Nunca se usa para esta clase, pues todos sus atributos y métodos son estáticos.

8.3.3.3.3 Atributos

- **private static final char[] hexDigits** - Digitos hexadecimales definidos en un *array* de caracteres.

8.3.3.3.4 Métodos

- **public static String toHexString(byte b)** - Método que convierte un *byte* en un objeto de tipo *String*.

  Parámetros

  - **byte b** - *Byte* a convertir.

  Salida

  - **String string** - Objeto de tipo *String*.

- **public static String toHexString(byte[] bytes)** - Método que convierte un *array* de *bytes* en un objeto de tipo *String*.

  Parámetros

  - **byte[] b** - *Array* de *bytes* a convertir.

  Salida

  - **String string** - Objeto de tipo *String*.

- **public static boolean areEqual(byte[] array1, byte[] array2)** - Método que compara dos *arrays* de *bytes* elemento a elemento.

  Parámetros

  - **byte[] array1** - *Array* de *bytes* 1.
  - **byte[] array2** - *Array* de *bytes* 2.

  Salida

  - **boolean result** - Booleano que indica si los *arrays* son iguales.

- **public static boolean areEqual(char[] array1, char[] array2)** - Método que compara dos *arrays* de *char* (caracteres) elemento a elemento.

  Parámetros

  - **byte[] array1** - *Array* de *bytes* 1.

  - **byte[] array2** - *Array* de *bytes* 2.

  Salida

  - **boolean result** - Booleano que indica si los *arrays* son iguales.

- **public static byte xor(byte byte1, byte byte2)** - Método que hace de puerta lógica *XOR* para dos *bytes*. Tanto este método como el siguiente se utilizan en la comprobación de paridad.

  Parámetros

  - **byte byte1** - *Byte* 1.
  - **byte byte2** - *Byte* 2.

  Salida

  - **byte result** - Resultado de la operación *XOR*.

- **public static byte xor(byte[] array1, byte[] array2)** - Método que hace de puerta lógica *XOR* para dos *arrays* de *bytes*

  Parámetros

  - **byte[] array1** - *Array* de *bytes* 1.
  - **byte[] array2** - *Array* de *bytes* 2.

  Salida

  - **byte[] res** - Resultado de la operación *XOR*.

- **public static boolean isZero(byte[] bytes)** - Método que comprueba si todos los elementos de un *array* de *bytes* son nulos.

  Parámetros

  - **byte[] bytes** - *Array* de *bytes*.

  Salida

  - **boolean result** - Booleano que indica si todos los elementos del *array* son nulos.

8.3.4. Otras clases

Los paquetes de *OpenSky* cuentan también con una clase llamada *Decoder*, en la cual está basado el decodificador explicado en 8.1.4. Por otro lado, también tiene una aplicación llamada *ExampleDecoder* que permite escribir en consola un mensaje Modo S en forma de cadena de caracteres *ASCII* y decodificarlo, mostrando en pantalla la información contenida en el mensaje. Esta aplicación se ha utilizado únicamente para la comprobación del correcto funcionamiento de nuestro decodificador.

Parte IV

Conclusiones y vista al futuro

A lo largo de esta memoria se ha presentado y descrito la normativa aeronáutica que es necesario estudiar y aplicar para que el sistema desarrollado funcione correctamente; se ha establecido el diseño del mismo, tanto a nivel de hardware como de software, para alcanzar los objetivos y cumplir los requisitos establecidos; y, por último, se ha presentado una primera versión de la implementación del sistema que, como se ha recalcado, será actualizada y mejorada en versiones futuras del proyecto.

Todo lo expuesto aquí es el resultado de un proceso de desarrollo largo y complejo, marcado por la adquisión de conocimiento en las distintas materias que abarca el proyecto y la obtención de experiencia en éstas a la hora de resolver problemas de diversos tipos y tomar decisiones relativas al diseño y la implementación. Así, a continuación se procederá a describir las conclusiones exraídas de todo este camino recorrido, junto con las mejoras propuestas para futuras versiones del sistema, además de algunas de las dificultades surgidas que se han tenido que superar y que han contribuido, finalmente, a un aprendizaje continuo.

- **El Modo S y la normativa ICAO**

  En lo referente al Modo S de vigilancia, es destacable la enorme cantidad de información que éste puede proporcionar en los distintos tipos de mensajes. Es cierto que, como se ha visto, una parte importante de la información que se transmite en Modo S lo hace por un enlace de datos entre una estación de tierra y la aeronave (*Comm-B/Comm-D*), información que no es posible recibir en nuestro sistema. Sin embargo, la existencia del protocolo *Comm-B Broadcast* hace que, en algunos casos, sí sea posible obtenerla.

  Hay que mencionar que la normativa de *Eurocontrol* relativa al Modo S requiere que todas las aeronaves que vuelen en territorio europeo bajo normas de vuelo instrumental (*IFR*) estén equipadas con un transpondedor Modo S, por lo que, en el territorio español, que es el que se presente cubrir con nuestro sistema, se pueden llegar a registrar todas las aeronaves con *IFR* cuando se lleve a cabo la instalación de las antenas necesarias para alcanzar los objetivos de cobertura.

  Por otro lado, se puede intuir que la normativa relativa al transpondedor y los mensajes Modo S será ampliada en el futuro. La última edición del Anexo 10 de la *ICAO* se lanzó en 2014 y sólo la primera edición del *9871*, en 2008. Esto quiere decir que si se pretende mantener y actualizar nuestro sistema, probablemente se deberá volver a estudiar la normativa establecida en las futuras versiones de estos documentos y otros que puedan surgir. Por ejemplo, el sistema *ACAS* se encuentra en continuo desarrollo y lo más probable es que la normativa relativa a los mensajes que dan soporte a este sistema vaya cambiando.

  Sin embargo, toda esta normativa ya está en un avanzado estado de desarrollo y que, además, parece que la *ICAO* trata cuidar la compatibilidad entre versiones (véase el caso de *Extended Squitter* Versión 0 y Versión 1). Esto significa que, en principio, no deberían de surgir problemas importantes a la hora de adaptar el sistema a la nueva normativa.

- **Diseño hardware y dump1090**

  El diseño general del sistema fue algo que se estableció al principio del proceso, junto con los requerimientos. A partir de aquí, se ha ido detallando para cumplir con estos requisitos.

  La dificultad más destacable que ha surgido en relación a la configuración hardware y software receptores ha sido la decisión sobre protocolo o el modo por el que se iba a configurar la *RPi* (como se ha visto, se eligió realizar la configuración en *Windows* por *SSH*). Por otro lado, la instalación de *dump1090* ha aportado algunos conocimientos básicos relativos a las órdenes presentes en *Linux* y a la instalación de aplicaciones en este *SO*. Cabe recalcar la multitud de funciones que se han descubierto en relación a *dump1090*, con un conjunto bastante completo

de opciones de configuración.

También, en su momento, se realizó la configuración del *NTP (Network Time Protocol)*[6] en la placa, con el objetivo de obtener una señal de reloj precisa. De momento, esta configuración no ha tenido ninguna utilidad, pero quizá sirva en un futuro para sincronizar los sistemas receptores con objetivos de integración (Ver 5.3.6).

El proceso de elección del formato de mensaje a transmitir a la aplicación integradora ha tenido una gran importancia. Se estudiaron los distintos datos proporcionados por ambos formatos y las ventajas de trabajar con cada uno de ellos. Sin embargo, finalmente el formato *Beast* se impuso a *BaseStation* claramente, por la gran diferencia en la cantidad de información y la importancia de ésta.

Por último, la necesidad de establecer un origen de tiempos común para el sistema probablemente traerá cambios profundos en el diseño hardware. Ya sea mediante un dispositivo GPS o algún sistema diferente, se deberá solucionar esta cuestión de una forma apropiada.

- **Diseño e implementación Java**

  Esta parte del proyecto ha sido claramente la más compleja y complicada, pero también ha permitido adquirir mucha experiencia en el campo de la programación orientada a objetos. Concretamente, se han descubierto y puesto en práctica la enorme cantidad de herramientras (clases) que proporciona *Java* para un proyecto de este tipo. Vamos a mencionar, por puntos, las partes más relevantes de este proceso de desarrollo, cómo se ha llevado a cabo este proceso en algunas de estas partes y las conclusiones que se han extraído.

  – Por un lado, en lo que a recepción de información se refiere, en un primer momento se pretendía recibir los mensajes Modo S en forma de caracteres *ASCII* directamente, pues éstos son proporcionados por el puerto 30003 de la *RPi*. El problema es que este formato solamente contiene los mensajes Modo S como tal, sin el *timestamp* del formato *Beast* completo. Por ello, en un principio el hilo encargado de recibir los mensajes y llamar al decodificador era bastante sencillo, pues simplemente leía línea a línea los mensajes (pues así es como éstos se envían por el puerto 30003, un mensaje por línea). Sin embargo, al cambiar al puerto 30005, donde los mensajes se mandan *byte* a *byte* de forma seguida, sin cambiar de línea, ha sido necesario añadir complejidad al algoritmo.

    Por ejemplo, en un principio se usaba lo que se conoce como un *Buffered Reader* (clase *BufferedReader*) en lugar de una instancia de *DataInputStream* para leer la información. La clase en cuestión permite leer texto de un *stream* de caracteres, que era lo que se recibía en aquel caso. Además, realmente no era necesario realizar ninguna comprobación en la recepción de mensajes, simplemente las líneas se pasaban al decodificador y éste detectaba si el mensaje era válido o no. El diseño actual es más complicado porque requiere un manejo de excepciones más cuidado dentro del hilo del receptor, al realizar comprobaciones *byte* a *byte*. Se ha tratado de aportar a la aplicación la robustez necesaria para que ésta pueda seguir funcionando aunque se reciban mensajes no válidos, pero siendo capaz a su vez de detectar cuándo es la propia aplicación la que está fallando a la hora de leer los mensajes recibidos.

    Cabe mencionar que, cuando se realizó este cambio en el diseño del sistema relacionado con el formato de recepción (de caracteres *ASCII* a *Beast* puro), se detectó que se recibían cada cierto tiempo mensajes con una dirección *ICAO* nula (000000) sin ningún tipo de información contenida en ellos. Al no aportar absolutamente ningún dato útil, el decodificador simplemente los descarta. Esto es un ejemplo de caso que es fundamental abordar, aunque

[6] Se trata de un protocolo para sincronizar el reloj del sistema con una referencia externa.

no esté claro el origen de estos mensajes, pero no es el único. En el futuro se debería tratar de encontrar respuestas a la cuestión de por qué se reciben estos mensajes exactamente.

Por último, hay que añadir que la aplicación se ha diseñado con el objetivo de alcanzar la escalabilidad, uno de los requisitos fundamentales establecidos. El hecho de que simplemente se cree un hilo por cada antena receptora conectada al sistema hace que la aplicación no requiera de grandes cambios al incluir nuevos receptores.

- Otra característica del sistema es el decodificador y su interacción con los mapas de tráfico. Como ya se explicó, se realiza en primer lugar la actualización del mapa de mensajes y, a partir de éste, la del mapa de datos, pues así se asegura que toda la información que va a ser transmitida a los clientes es justamente la que está almacenada en el mapa de mensajes. Este último es una especie de base de datos interna de información de tráfico de la que siempre se debe extraer cualquier dato transmitido. Además, como se vio, una gran parte de la información contenida en los mensajes no se transmite a los clientes, fundamentalmente la relacionada con *ACAS* y con la precisión e integridad de la posición y la velocidad, pero quizá, por exigencias del proyecto, en un futuro se empiecen a transmitir también estos datos.

- La extrapolación de la posición que se realiza en nuestro sistema es bastante básica. Sólo se utilizan algunos datos para hacerla, pero se puede añadir más complejidad al algoritmo si se tienen en cuenta también el *heading*, la *IAS* y la *TAS*. Además, se utiliza el algoritmo de la loxodrómica, cuando se podría utilizar un proceso más complejo para realizar la estimación. En futuras versiones se podrían considerar estas posibilidades, aunque bien es cierto que se ha visto que, generalmente, cuando una aeronave se encuentra en cobertura, ésta envía mensajes de posición de una manera bastante continua, lo que en resta importancia a la necesidad de estimación. Se debería estudiar esta importancia y decidir si una extrapolación compleja es verdaderamente necesaria.

- Se ha explicado en la memoria el concepto de sincronización de objetos y los monitores, y cómo se aplica éste en el diseño de la aplicación. A la hora de realizar la implementación, se ha visto que, claramente, la tasa de actualización de los mapas se hace más lenta al sincronizar éstos, aunque es necesario hacerlo para asegurar un correcto funcionamiento.

  El efecto de la falta de sincronización se ha podido observar, por ejemplo, a la hora de implementar el método *getByteArray()*, que obtiene la información del mapa de datos para ser enviada a los clientes. Si no se sincroniza el mapa de datos mientras se está escribiendo la información del mismo en el flujo de datos enviado por el servidor, esta información escrita no coincide con la contenida en los mapas en todo momento.

  También, por ejemplo, cuando se ha tratado de escribir en el flujo solamente la información de las aeronaves que tienen información de posición, se ha realizado una comprobación antes de escribir los datos (con un iterador) para averiguar el número de aeronaves con posición. Si esta comprobación no se sincroniza también, entonces el número de aeronaves presente al principio del bloque de datos transmitido por el servidor (el que va seguido de una *N*, ver 5.3.5.3) no coincide con el número real de aeronaves contenidas en el bloque.

  Por último, los problemas al no sincronizar el limpiador de tráfico también son evidentes, al surgir conflictos a la hora de tratar de añadir y eliminar tráfico de manera simultánea.

- La importancia de los *timestamps* también ha estado muy presente en el proyecto. En un principio, se pretendía que todas las marcas de tiempo tuvieran un mismo origen, la medianoche *UTC* del 1 de Enero de 1970. Esto se decidió porque existe una orden que permite

obtener la fecha y hora con este origen en la *RPi* y, a partir de ésta, se quería corregir el *timestamp* del mensaje (calculando el tiempo de transmisión de la orden por *SSH* a la placa, el tiempo de procesamiento y el tiempo de vuelta). Sin embargo, resultó que este método tenía un error de, al menos, dos segundos. Esto es, el *timestamp* calculado tenía unos dos segundos de diferencia respecto al real. Por ello, se optó por no utilizarlo y tratar de buscar de otra forma la uniformidad en el origen de tiempos. Como se mencionó, de momento no se ha ideado una solución al problema, aunque lo más probable es que se necesite alguna fuente externa de reloj para el sistema.

– En lo que a integración de la información se refiere, como se ha visto, aunque no se ha desarrollado este aspecto del sistema en profundidad, sí han aparecido algunas ideas que se podría implementar en futuras versiones del proyecto.

  Una forma de conseguir mejorar la integración sería ampliando el número de mapas de tráfico, creando un tipo de mapa por cada una de las antenas receptoras del sistema. Cada uno de estos mapas tendría algún identificador que permitiría saber qué zona cubre la antena correspondiente. Así, a la hora de recibir un mensaje, no se tendría que iterar una extensísima lista de todas las aeronaves del sistema, sino sólo aquella/s correspondiente/s a la zona en la que se encuentra la aeronave.

  Además, esto es una mejora fundamental si se pretende ofrecer un servicio selectivo, es decir, uno que permita la selección de un área determinada de toda la zona de cobertura, para obtener únicamente la información de la misma. Al igual que en el caso de la integración, la importancia de esta ampliación de mapas en el sistema selectivo se debe a que es muy ineficiente y complicado comprobar qué aeronaves se encuentran en la zona de interés del cliente si todas ellas se encuentran en el mismo mapa.

  Por último, la integración es la característica por la que se debe elegir un origen para las marcas de tiempo de los mensajes, pues permite resolver las ambigüedades que surgen en la recepción de mensajes idénticos por parte de estaciones distintas.

– A raiz del proceso de desarrollo del *display* que muestra la información que envía el servidor, también se han podido conocer algunas funcionalidades que ofrece *Java* en lo que a creación de interfaces gráficas se refiere. Éstas permiten la creación de estas interfaces (como la tabla de contenido dinámico que se ha diseñado) de manera sencilla e intuitiva, si se comprende el funcionamiento general de las clases utilizadas en el proceso. En versiones futuras del proyecto se pretende crear una interfaz más compleja, al estilo *flightradar24.com*, lo que requerirá un conocimiento más avanzado acerca de esta parte del lenguaje.

Bibliografía

International Civil Aviation Organization (2014). *Annex 10, Aeronautical Communications - Volume IV.* (5^a ed.)

International Civil Aviation Organization (2008). *Doc 9871, Technical Provisions for Mode S Services and Extended Squitter.* (1^a ed.)

International Civil Aviation Organization (2004). *Manual on the Secondary Surveillance Radar (SSR) Systems.* (3^a ed.)

Eckel, B. (2008). *Thinking in Java.* (4^a ed.)

Oracle Corporation. *Java Documentation.* `https://docs.oracle.com/en/java/`

modesbeast.com. *Mode-S Beast:Data Output Formats.* `http://wiki.modesbeast.com/`

Bones Aviation Page. *SBS Basestation.* `http://woodair.net/sbs/Article/Barebones42\_Socket\_Data.htm`

OpenSky Network. *Mode S and ADS-B Decoder for Java.* `https://opensky-network.org/social/projects/20-java-adsb`

David & Cecilia Taylor's Web Pages. *ADS-B dump1090.* `http://www.satsignal.eu/raspberry-pi/dump1090.html`

FlightAware. *FlightAware.* `http://flightaware.com/`

Raspberry Pi - Teach, Learn and Make with Raspberry Pi. *Raspbian.* `https://www.raspberrypi.org/downloads/raspbian/`