Universitat Politècnica de València

Escuela Técnica Superior de Ingeniería del Diseño
Grado en Ingeniería Aeroespacial

TFG:

Integración de tráficos en entorno de simulación

Maximiliano Barrios García

Tutor:
Pedro Yuste Pérez
Cotutor:
Juan Antonio Vila Carbó

Índice general

CAPÍTULO 1

Introducción

1.1. Motivación

Este proyecto presenta un objetivo principal, la integración del tráfico real recogido por la antena SACTA en entorno de simulación de navegación aérea presente en el laboratorio de Aeronavegación de la UPV. Los dos software de simulación escogidos son el *EuroScope* y el *XPlane*, ya que son los empleados en las prácticas de la asignatura Gestión del Espacio Aéreo II de la modalidad de Aeronavegación en el grado de Ingeniería Aeroespacial. En ellas, el alumnado se dividirá en controladores aéreos en puestos ATC y en pilotos. De esta forma se consigue emular una situación real en la cual los alumnos puedan probar sus habilidades para prevenir colisiones y ordenar el flujo de tráfico. De esta forma, gracias a nuestro proyecto, las aeronaves que visualizarán los alumnos en las diferentes herramientas de simulación elegidas serán aviones reales que sobrevuelen Valencia en ese mismo momento, además de los posibles simulados por el mismo programa.

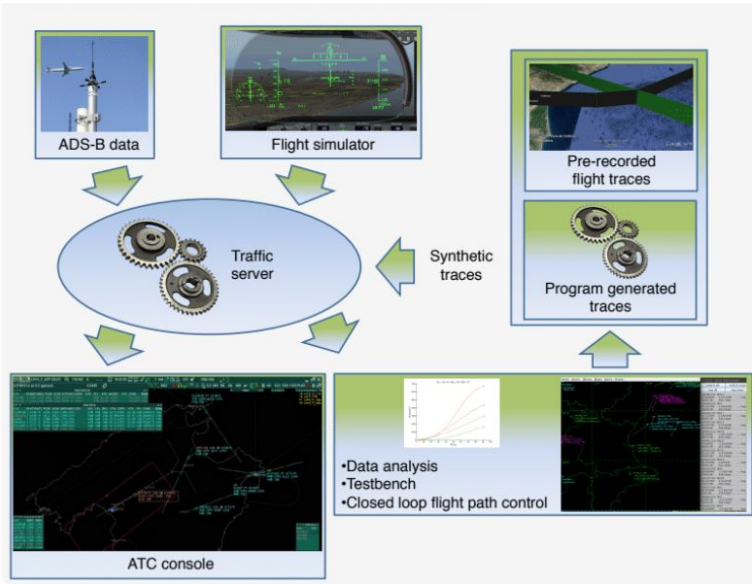


Figura 1.1: Estructura del entorno de aeronavegación

El proyecto se divide en dos programas independientes y que pueden ser ejecutados a la vez. El primero de ellos será el encargado de recoger el tráfico de la antena, filtrarlo y enviarlo al servidor FSD, recipiente del cual representará vuelos el *EuroScope* en los diferentes puestos de ATC que se estén simulando en ese instante. En cuanto al segundo programa, su función será obtener los tráficos almacenados dentro del FSD, clasificarlos por cercanía y enviarlos al *XPlane*. De esta forma, los alumnos que ejerzan de pilotos en las prácticas podrán visualizar desde la ventanilla de su cabina el resto de vuelos a tiempo real que se mueven por el espacio aéreo de Valencia. Hemos optado por hacer el proyecto en Java debido a su portabilidad y a la posibilidad de ser ejecutado en cualquier máquina, independientemente del sistema operativo que tenga.

1.2. Antecedentes

Previamente se han realizado trabajos de fin de grado que consistían en implementar tráfico en diferentes simuladores. Nuestro caso es ir un pasito más de lo que se ha hecho anteriormente e implementar tráfico real de la antena. A pesar de que este tráfico presenta algunas irregularidades, gracias al filtrado y posterior depuración dentro de nuestro proyecto, hemos obtenido que los vuelos cumplan unas trazas que se asemejen bastante a las reales. Para la implementación del primero de los dos programas que componen este proyecto hemos reutilizado varios objetos del ATSIM (**TFG Herramienta para la simulación del sistema de control de tráfico aéreo** [1]) ya que guarda bastantes similitudes con nuestro proyecto. En el caso del segundo programa, partiremos de la información presente en el manual de *XPlane* para enviar y recibir datos mediante mensajes UDP. La usaremos como guía para diseñar el programa de Java encargado de esta función.

1.3. Resumen

Nuestro proyecto, como ya se ha indicado, consta de dos partes. El primer programa, el ITS, es el encargado de enviar los vuelos al FSD. Para ello, se obtendrán los mensajes recibidos por la antena mediante *sockets* TCP de Java. A continuación se almacenarán en un objeto específico de Java que explicaremos con mayor detenimiento en el capítulo 3. Acto seguido, se procede a enviar estos mensajes al servidor, para lo cual hay que seguir un procedimiento concreto de mensajes, de forma que el FSD interprete y almacene la información para cada avión. En caso de que se desconozca algún término necesario, el programa optará por rellenarlo como ZZZZ, siguiendo la práctica habitual en aviación. La forma que tenemos de que usuario y programa interactúen es una serie de interfaces gráficas mediante las cuales pedimos al usuario la dirección IP del servidor, los datos de usuario y contraseña para conectarse a él, así como su el nombre del cliente. El programa, a su vez, tiene una clase ejecutándose constantemente que permite limpiar los tráficos obsoletos o con información espuria. Por último, se ha optado por crear una sencilla interfaz donde el usuario pueda ver todos los tráficos almacenados y contrastar con la información real de los aviones.

[1] TFG hecho por Davinia González Morello, Ing. Aeroespacial, 2017

El segundo programa, el FSD2XP, se encargará de enviar el tráfico almacenados en el FSD, ya sea mediante el programa anterior o por algunos programas externos como pueden ser el ATSIM, al *XPlane*. Comenzará instanciando una serie de interfaces gráficas en las que se pedirán los parámetros necesarios para conectarse al FSD y al *XPlane*. Leeremos la posición actual del avión controlado por el piloto dentro del simulador de vuelo. Después, enviaremos esta información mediante paquetes TCP[2] al FSD. De manera automática, el servidor nos devolverá las posiciones de todos los aviones dentro de un radio que podemos configurar dentro de los archivos de texto del programa. Estos aviones serán filtrados y almacenados dentro el programa. Con esto hecho, simplemente se filtrarán los aviones más cercanos para ser enviados de vuelta al *XPlane*, donde el usuario podrá verlos a tiempo real.

[2]Explicado con detalle en el Capítulo.4

CAPÍTULO 2

Introdución teórica

2.1. Sistemas de Control de Tráfico Aéreo

Estos sistemas son los encargados de proporcionar un servicio de guiado a las aeronaves en espacios aéreos controlados y de ofrecer apoyo a los pilotos en el caso de espacios no controlados. Su objetivo principal es proporcionar seguridad, orden y eficiencia al tráfico aéreo. En este proyecto emplearemos una reproducción a menor escala del SACTA de Indra, que es el sistema más conocido y el empleado por Enaire en España. El SACTA es un sistema que tiene como finalidad asegurar la coordinación del movimiento del tráfico aéreo, a la vez que se gestionan incidencias y tanto aterrizajes como despegues de una forma eficiente sin que se produzca ningún tipo de retraso. Este control del tráfico se efectúa desde los Centros de Control de Tránsito Aéreo (ACC) donde podemos encontrar dos radares, uno primario y uno secundario. Este centro se pretende reproducir en la ETSID donde, con una antena situada en la azotea, actuando a modo de radar, podemos obtener mensajes ADS-B emitidos por los transpondedores de los distintos aviones que vuelen en un espacio aéreo de unas 150 millas de radio. La antena recibe mensajes de diferentes tipos como pueden ser mensajes de identificación, mensajes de posición,etc. Mediante un programa en Java podremos conectarnos a esa antena y filtrar la información obtenida de cada vuelo, centrándonos exclusivamente en la información necesaria. Además, de ir actualizando la información a medida que esta es recibida.La información obtenida atraviesa un filtrado específico adicional dentro del programa llegando a un mensaje con la siguiente estructura:

MSG,1,163,21395,4CA33E,3456843,2016/09/21,12:46:16.693,2016/09/21,12:46:17.083,RYR98ZY,,,,,,,,,,

Cada término representado entre comas se corresponde a un parámetro distinto del avión(latitud, *squawk*, código hexadecimal,etc). De esta forma es más sencillo almacenar los términos que a partir de la información obtenida directamente de la señal en bruto emitida por el transpondedor. Los mensajes espurios se diferencian de los mensajes correctos principalmente por el primer término del mensaje. El filtrado del programa detecta cuando el mensaje recibido presenta anomalías. Si no las presenta, lo identifica como MSG, SEL, ID o AIR. A partir de estos datos ya podemos proceder a la conexión con el servidor FSD.

2.2. FSD

El FSD o *Flight Simulator Daemon* es un tipo específico de servidor empleado en el sector de la simulación de tráfico aéreo. Su principal función es almacenar el tráfico recibido de tal forma que un programa externo pueda obtener los datos de los aviones almacenados dentro. El servidor FSD empleado en este trabajo es una versión beta del FSFDT Windows FSD server.

El funcionamiento del FSD consta de tres pasos: conexión de la aeronave, envío del plan de vuelo y envío de la posición. Para ello, se tendrá que enviar tres mensajes, mínimo, para que el avión se almacene dentro del servidor. Estos mensajes son:

1. **Mensaje de conexión.** El mensaje de conexión es el encargado de crear un pequeño espacio dentro del servidor reservado para la aeronave del mensaje en concreto. Se envía solo una vez, ya que de lo contrario puede dar lugar a errores dentro del servidor provocando la desconexión de este mismo. Este mensaje tiene la siguiente forma:

   "#AP" + callsign + ":SERVER:" + usuario + ":" + password + ":1:9:11:" + nombre +"\n" donde:

   - *# AP.* Significa que la conexión es como piloto. Existe la posibilidad de conectarse al FSD como controlador pero no se dará el caso dentro de este trabajo.
   - *Callsign.* Es el código diferenciador del avión dentro del servidor. A partir de ahora, toda la información que se envíe para este mismo avión deberá tener el mismo *callsign.*
   - *Usuario y password.* Usuario y contraseña que permiten efectuar la acreditación para acceder al servidor. Se pueden editar dentro del fichero de texto *cert.text* que se encuentra dentro de la carpeta donde se encuentra el FSD.
   - *nombre.* El nombre del piloto.

   El mensaje de conexión no es necesario en algunos FSDs.

2. **Mensaje de plan de vuelo.** El mensaje de plan de vuelo pretende almacenar el plan, previamente diseñado, del avión cuyo mensaje se está enviando. Este mensaje no es obligatorio de enviar, sin embargo algunos programas externos como el *EuroScope* sí que necesitan esta información para operar de una forma más realista. La forma del mensaje de plan de vuelo es la siguiente:

   "$FP"+ callsign + ":*A:"+ Flight_ Rules(V|I) + ":" + Aircraft_ type + ":" + TAS + ":"+ DepAirport + ":" + DepTime + ":" + Actual_ Time + ":" + CruiseAltitude + ":" + ArrAirport + ":1:40:00:00:"+AltnAirport+":"+remarks+": " + DepAirport + " " + Route + " " + ArrAirport + "\n" donde:

   - *$FP.* Es el identificador de que el mensaje es un mensaje de plan de vuelo.
   - *Callsign.* El *callsign* del avión.

- *Flight_ Rules(V|I).* En este apartado tendremos que elegir si el avión vuela con reglas de vuelo visuales o instrumentales. En este trabajo todos los vuelos serán, por defecto, instrumentales.
- *Aircraft_ type.* Modelo de la aeronave.
- *TAS.True AirSpeed.* Velocidad del avión.
- *DepAirport.* Aeropuerto de salida.
- *DepTime.* Hora de salida
- *Actual_ Time.* Hora real.
- *CruiseAltitude* Altitud de crucero.
- *ArrAirport.* Aeropuerto de llegada.
- *AltnAirport.* Aeropuerto alternativo.
- Por último, los apartados que faltan son los diferentes *WayPoints* que componen la ruta del vuelo.

3. **Mensaje de posición.** El mensaje de posición tiene como función enviar al FSD los diferentes parámetros necesarios para situar el avión en el espacio. Este mensaje será el que se esté enviando continuamente. De esta forma, la posición del avión se sobrescribirá dentro del servidor, actualizándose a medida que el avión real se mueve.Tiene la siguiente forma:

   "@N:" + callsign + ":" + squawk + ":1:" +latitud+ ":" + longitud + ":" + altitud + ":" + TAS + ":" + Rumbo + ":0" + "\n"
   donde:

   - *@N.* Modo del transpondedor. Puede ser N (modo radar primario + modo A/C radar secundario), S(modo *StandBy*) o I(modo IDENT). Por defecto es N.
   - *callsign.* El *callsign* del avión.
   - *squawk.* El código del transpondedor.
   - *latitud.* La latitud del avión (en grados).
   - *longitud.* La longitud del avión (en grados).
   - *altitud.* La altitud del avión (en pies).
   - *TAS.True AirSpeed.* Velocidad del avión.
   - *rumbo.* El rumbo del avión (en grados).

El método empleado para enviar estos mensajes al FSD será el protocolo TCP (*Transmission Control Protocol*). La forma de operar de este protocolo es mediante la creación de un canal exclusivo entre los dos equipos entre los cuales se pretende establecer la comunicación. Este canal garantiza que toda la información enviada llegue al otro extremo con éxito y en el orden que ha sido enviada. Por lo tanto, se trata de un protocolo orientado a la conexión. La forma de comunicación empleada por este protocolo son los *sockets* TCP ya que proporcionan un punto de comunicación entre procesos ejecutándose en máquinas distintas. Estos *sockets* permiten el envío y recepción de datos entre procesos. Existen

sockets servidor y *sockets* clientes. En nuestro caso, el *socket* TCP servidor será el FSD y los *sockets* cliente serán creados por nuestro programa para conectarse al servidor cuyo puerto será 6809 siempre.

Al pasar un tiempo sin recibir información, el FSD dejará de almacenar datos para el *callsign* en concreto y borrará los residuales dentro del servidor. El funcionamiento del programa de Java encargado de enviar estos mensajes se explicará con más detalle en el Capítulo 3.

2.3. *EuroScope*

Uno de los programas se empleará para leer directamente los datos almacenados en el FSD es el *EuroScope*. *EuroScope* es un programa orientado a la simulación a tiempo real de un cliente ATC. Permite al usuario ejercer el papel de controlador aéreo, con las funciones que esto conlleva (prevenir el las colisiones entre aeronaves y mantener ordenadamente el flujo de aeronaves). Para ello se recrea una presentación radar que recoge los aviones que sobrevuelan el espacio aéreo cercano al escenario seleccionado. Este tráfico puede ser simulado por el mismo programa a partir de modelos dinámicos integrados en el mismo programa o tráfico que obtiene directamente del FSD, como es el caso de la primera parte de este trabajo. A partir de los datos obtenidos del servidor, el programa es capaz de posicionar las aeronaves en el espacio aéreo correspondiente al escenario deseado. Además, nos permite visualizar la ficha de progresión de vuelo correspondiente a cada avión, entre la que se incluye información del plan de vuelo.

2.4. *XPlane*

Por otra parte, en la segunda parte de este trabajo se empleará el simulador de vuelo *XPlane* 10. Es un simulador increíblemente detallado, donde el usuario puede ejercer de piloto con un realismo muy aproximado al pilotaje real. Este simulador permite la simulación de hasta 19 aeronaves, incluyendo al usuario, dentro del escenario que elijamos. El objetivo de la segunda parte de este proyecto, como ya hemos dicho, es la visualización del tráfico real más cercano, además de el resto de aviones de los demás alumnos. Para efectuar esta conexión es necesario conocer la forma de comunicación que tiene el XPlane con programas externos. Existen dos formas: mediante *plugins* o mediante el protocolo de comunicación UDP (*User Datagram Protocol*. Nosotros optaremos por emplear la segunda opción ya que es más flexible a la hora de llevarla a cabo. En primer lugar debemos saber que, al contrario que el protocolo TCP, UDP no está orientado a la conexión. Esto es debido a que la forma de transferir los datos no garantiza que la recepción de estos sea exitosa. Además, es posible que la integridad de los datos al llegar no sea la deseada ya que no se emplea un conducto exclusivo como en TCP sino datagramas. Estos paquetes son enviados por la red a espera de que el servidor de llegada los recoja, los ordene y los reconstruya. A pesar de estas desventajas, el protocolo UDP nos permite la conexión entre dos máquinas sin una sincronización previa entre el origen y el destino y tolera el envío de paquetes de datos y no solo *bytes* como el caso de TCP, lo cual reduce la cantidad de información necesaria permitiendo envíos más rápidos. Además, al buscarse la

comunicación en tiempo real, el protoclo UDP, al reducir la cantidad de datos, disminuye la latencia y se aumenta la cantidad de datos que se pueden enviar al contrario que en TCP que llegan con un cierto retardo.

Para establecer esta comunicación, en primer lugar, tendremos que configurar la salida y entrada de datos desde dentro del programa XPlane. Debemos seleccionar desde la pestaña Conexiones de Redes los puertos a través de los cuales se enviarán y recibirán los datos.

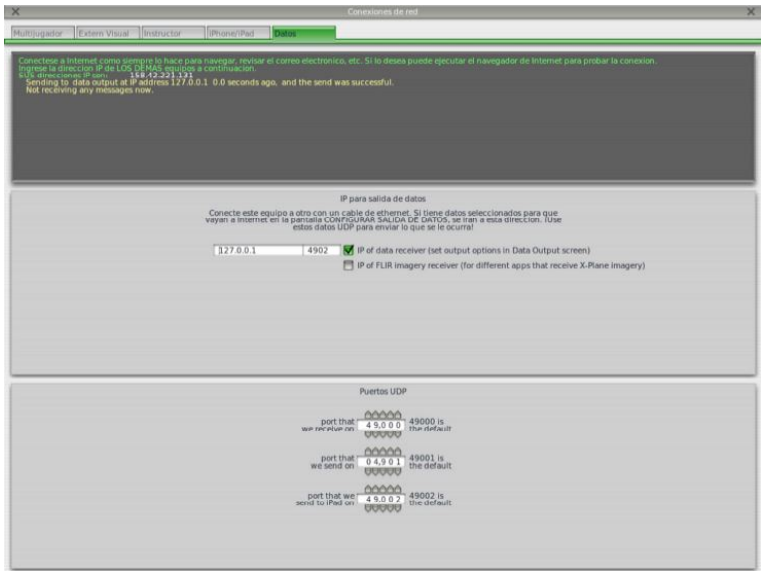


Figura 2.1: Configuración de Conexión de Red. XPlane

En la Figura 2.1 observamos la ventana que debemos configurar. En primer lugar, tendremos que seleccionar la dirección IP a la cual deseamos enviar los datos. Esta será por defecto la dirección *loopback* 127.0.0.1 ya que nos interesa que nuestro programa reciba la posición del avión controlado por el usuario. El puerto de envío de datos seleccionado será el 4902. Por otra parte, en la parte de abajo observamos los puertos UDP por los cuales recibirá XPlane información. Nosotros editamos el primero de ellos a 49.000 ya que será el que utilizaremos para hacerle llegar información al simulador.

Nuestro objetivo, mediante un programa en Java, será enviar los datos de los aviones más cercanos (filtrados en el mismo programa) almacenados previamente en el FSD. Este funcionamiento se explicará con mayor detalle en el Capítulo 4.

2.5. Diseño del *software*

2.5.1. Lenguaje de programación

Este trabajo está enteramente diseñado en el lenguaje de programación Java, en el cual se empleará un estilo de programación orientada completamente a objetos. Se ha optado por utilizar este lenguaje debido a que es portable y se puede ejecutar en cualquier

dispositivo, independientemente del sistema operativo que tenga. La plataforma empleada es Java SE 8 y el software que provee las herramientas de desarrollo para crear los programas en Java es JDK 8.

El único requisito necesario es que el ordenador donde se vaya a ejecutar el programa tenga instalado y actualizado *Java Runtime Environment* (JRE) compuesto por la máquina virtual de Java, como por las diferentes bibliotecas y componentes mediante los cuales se ejecutan las aplicaciones.

2.5.2. Entorno de desarrollo

El entorno empleado para diseñar el programa es el entorno de desarrollo *NetBeans*. Se ha optado por este entorno debido a que es gratutio y a que es de código abierto. La versión empleada en este proyecto será *NetBeans* 8.2.

2.5.3. Estructura del programa

Como hemos dicho anteriormente, Java es un lenguaje de programación orientado a objetos. Esto significa que se ha de definir una serie de clases a la hora de programar el código, que definen las plantillas para la creación de dichos objetos. Mediante este tipo de programación se consiguen una serie de ventajas que en otro tipo de programación no serían posibles:

- **Reusabilidad.** Las clases que definen a los objetos se pueden emplear ya sea dentro de otras clases dentro del mismo programa, o en futuros proyectos si tener que modificar sustancialmente el código.
- **Modificabilidad.** Al ser muy sencillo añadir, suprimir o modificar objetos, se pueden realizar modificaciones del código de una manera bastante sencilla. Esto favorece a su vez la reusabilidad del código en trabajos futuros.
- **Fiabilidad.** Como el código está dividido en objetos, que son partes de código relativamente pequeñas, podemos probarlas de manera independiente y sencilla facilitando el aislamiento y reconocimiento de errores.

Sin embargo, la programación orientada a objetos presenta desventajas como pueden ser el cambio de mentalidad a la hora de programar, ya que se trata de una forma de programar bastante más abstracta que la programación tradicional.

Dicho esto, las clases de este proyecto se pueden dividir en clases de cuatro tipos:

- **Clase principal.** Es la columna vertebral del proyecto ya que es la clase que contiene el *Main* del mismo. Al iniciar el programa, el primer método en ejecutarse es el *Main* de esta clase. En él se instancian todas las demás clases del proyecto en el orden que se desea ejecutarlas.
- **Clases para definir las interfaces de usuario.** Su función principal es definir una serie de interfaces que el usuario utiliza para interactuar con el programa. Heredan de la clase *JFrame*.

- **Clases para definir objetos.** Aquí se engloban todas las clases que no sean interfaces de usuario. Con estas clases el usuario no podrá interactuar durante la ejecución del programa.

En los dos capítulos siguientes se explicará el código de los dos programas que componen este proyecto utilizando la estructura explicada anteriormente.

CAPÍTULO 3

Implementación del Tráfico del SACTA en FSD

En este capítulo se explicará en profundidad el funcionamiento del código del ITS del proyecto, de cada una de las clases que lo componen. La finalidad de este proyecto es recoger el tráfico recibido por la antena SACTA y transmitirlo al servidor FSD. De esta forma podrá ser utilizado por otros programas externos como *EuroScope*.

3.1. Clase Principal

La clase principal de este programa se llama MainFinal y simplemente contiene el método *main* del programa donde se instancian las demás clases en el orden que se precisa. En primer lugar ejecutamos la clase MensajeConexiónES, la cual solicita al usuario, mediante una interfaz gráfica, la dirección IP del servidor FSD al cual nos queremos conectar. De no ser posible la conexión a la dirección IP introducida por el usuario, el mismo programa lanzará una ventana de aviso y detendrá la ejecución del mismo. Por defecto estará la dirección IP de *loopback*, es decir, una dirección especial empleada para redirigir el tráfico de datos hacia uno mismo. Esta dirección tiene como valor predeterminado 127.0.0.1. En el caso de querer utilizarse en el laboratorio Pedro Duque de la UPV, se tendrá que introducir la dirección IP del ordenador del profesor, que cuenta con el servidor FSD al cual se conectarán el resto de máquinas. Si la dirección IP es válida, se procede a instanciar la clase *Traffic*. Esta es la clase más importante de nuestro proyecto. En ella se ejecuta el objeto *AntennaReceiver* cuya función es leer los mensajes enviados por los transpondedores de los aviones que la antena recibe. Si la antena no se encuentra operativa, esta clase cerrará todos los procesos activos y detendrá el programa. La clase *Traffic*, a su vez, hereda de *ConcurrentHashMap* lo cual permite almacenar los tráficos dentro. Un *HashMap* es una colección de objetos identificados por una *key* o identificador, que puede ser desde una variable *double* hasta un objeto en sí. En nuestro caso, esto nos permite archivar los distintos aviones por su *callsign* y sobrescribir los parámetros recibidos por la antena, de manera que solo se almacenan los valores actuales dentro del mapa. Los valores estarán almacenados en una clase aparte llamada *FlightObject* que actúa como un baúl gigante donde se van guardando y actualizando las distintas variables. El método *message* instanciado dentro de la clase *AntennaReceiver* es el encargado de guardar estos valores dentro del *FlightObject*. También contamos con el método *EnviarES*, método genérico de envío de datos mediante mensajes TCP que emplearemos para establecer la conexión con el servidor FSD. Es el método mas importante dentro de la parte de envío

de datos al servidor y será empleado repetidas veces dentro de este proyecto, tanto en el primer programa como en el segundo.

Despúes de instanciar el método *Traffic*, mediante la clase MensajeUsuarioES se pedirá al usuario que introduzca los datos necesarios para la conexión con FSD, es decir, el usuario, la contraseña y el nombre de la persona que interactúa con el programa. Con todos los datos, el programa ya es capaz de establecer una conexión con el servidor y empezar a enviar información. Por lo tanto, se abrirá lo será la principal interfaz del programa, la clase Interfaz. En ella observaremos una tabla con columnas en las cuales se mostrarán los parámetros que describen la posición de cada avión, identificado con su correspondiente *HexIdent*. Además, se podrán ver los datos introducidos por el usuario de forma que, de haber algún error, este sea capaz de detectarlo. Esta tabla de datos se editará con la clase EnviarInterfaz, clase encargada de recorrer tanto el mapa generado dentro de *Traffic* como la tabla que observamos en la interfaz. Si no encuentra coincidencia de *HexIdent*, añadirá una nueva fila a la tabla. De encontrarla, simplemente sustituirá los valores correspondientes que hayan variado.

El programa, a su vez, cuenta con una clase encargada de limpiar los tráficos de los cuales no se haya recibido información en una determinado espacio de tiempo. Esta clase es *TrafficCleaner* y se encarga de eliminar del mapa *Tráffic* los tráficos deseados y de editar la tabla anterior, suprimiendo la fila con el tráfico que ya no existe. Simultáneamente se está ejecutando la clase EnviarEs, encargada de transferir los datos de los aviones almacenados en *Tráffic* al FSD. En ella se abrirá un *socket* TCP por cada avión para evitar la colisión de los datos que puede acabar en que el servidor se bloquee ocasionando un fallo en el programa. Dentro de esta clase se enviarán los tres mensajes explicados en el Capítulo.2.2. Tenemos que tener en cuenta que el mensaje de conexión solo se debe enviar una vez, por lo que este objeto comprobará antes de enviar dicho mensaje que no ha sido enviado previamente. Se ha de tener en cuenta el cambio que experimenta el *callsign* dentro del programa ya que se actualiza a medida que se recibe la información. En una primera instancia es normal no recibir el *callsign*, por lo que el mensaje será enviado al FSD con el código hexadecimal como identificador. En el momento que se ha recibido el *callsign* real, se cierra la conexión con el elemento anterior y se crea una nueva, asegurándose de enviar los tres mensajes necesarios. Como hereda de la clase *Thread*, se estará ejecutando constantemente recorriendo *Traffic key* a *key* enviando los parámetros solicitados por el FSD para cada vuelo.

El programa términará cuando el usuario decida cerrar la interfaz principal, cerrándose toda conexión abierta con la antena y el servidor.

3.2. Clases Interfaces

3.2.1. Clase MensajeConexionES

Esta clase consiste en una sencilla interfaz que solicita la IP del servidor FSD al cual se desea enviar los datos de los tráficos. Por defecto es 127.0.0.1. El usuario puede editarla

y seleccionar el botón "Enviar". Al instanciar el objeto se genera la ventana llamando al método iniciar() en el cual se han definido los componentes de la ventana. Al pulsar el botón "Enviar" se ejecutará el método validarip() que se encargará de comprobar que si se puede abrir un socket de conexión TCP con el el servidor FSD con la dirección IP provista y el puerto 6809 (puerto del cliente del FSD). Si se completa la conexión satisfactoriamente, saltará una ventana informando del éxito y dejará paso a la siguiente interfaz.

Figura 3.1: Interfaz MensajeConexionES

3.2.2. Clase MensajeUsuarioES

Esta clase se inicia inmediatamente después de generarse la clase *Traffic*. En ella, al igual que en la anterior interfaz, se genera una ventana que pide al usuario que introduzca el usuario y la contraseña necesarios para conectarse al FSD. Además, se pide también el nombre de la persona que manipule el programa. Al pulsar el botón "OK", se genera un manejador de eventos que recoge los valores y ejecuta el método validarUsuario() que funciona análogamente al método validarip() de la clase MensajeConexionES. Se comprueba que el usuario y la contraseña son los correctos para establecer conexión con el FSD. Si es válida, se cierra la ventana y se abre la interfaz principal del programa.

3.2.3. Clase Interfaz

Se trata de la interfaz principal del proyecto. Consiste en una tabla, que se edita con en tiempo real, que recoge los parámetros principales de cada tráfico (código hexadecimal, *callsign*, latitud, longitud, altitud, rumbo y TAS). Cada tráfico está identificado por su código hexadecimal ya que este permanece constante a lo largo del vuelo. Se ha optado por ese parámetro como identificador en lugar del por el *callsign* porque los datos que recoge la antena no siempre son completos. Lo único que sabemos con certeza cuando un tráfico es recibido por la antena es su código hexadecimal. El resto de valores pueden variar o simplemente no aparecer en nuestro banco de datos debido a la lejanía de los vuelos de los que recibimos información. Como es obvio, a más cercanía, mayor será la precisión de los datos obtenidos.

En la parte superior de la ventana se crean unas etiquetas que muestran el nombre de usuario, la contraseña y la IP introducidas. De esta forma el usuario podrá comprobar

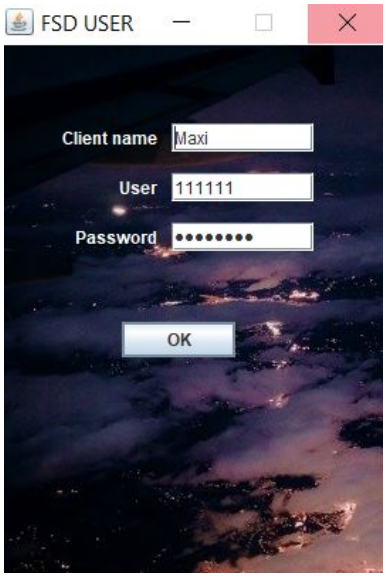


Figura 3.2: Interfaz MensajeUsuarioES

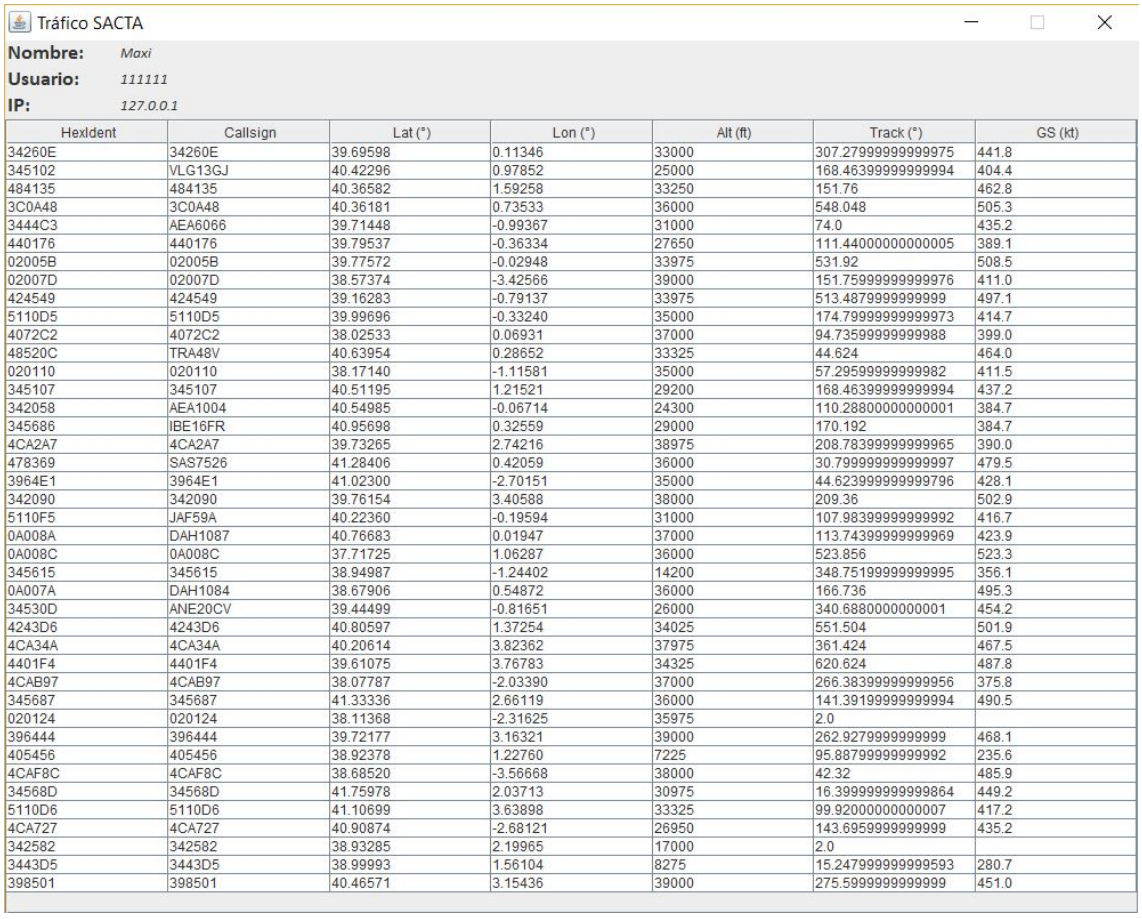
que los datos introducidos son los correctos. El funcionamiento de la tabla se basa en tres métodos definidos dentro de esta misma clase que, sin embargo, son instanciados dentro de la clase EnviarInterfaz que explicaremos con mayor detenimiento más adelante. Los tres métodos son:

- **addRow2Table**. Es un método sencillo que toma como parámetros los diferentes valores mencionados anteriormente y, mediante el método de la clase JTable de Java addRow, son añadidos a la tabla.

- **setValueAt**. Este método tiene como parámetros de entrada los mismos que el método anterior más una variable entera que simboliza el número de la fila que queremos editar. Mediante el método de JTable setValueAt(String valor,int fila,int columna) editamos las diferentes filas de nuestra tabla.

- **removeRow**. Este método toma como entrada una variable entera que representa el número de la fila que deseamos borrar. Ejecuta el método de JTable removeRow que simplemente borra la fila deseada. Este método, al contrario que los otros dos, será instanciado en la clase TrafficCleaner, encargada de limpiar los tráficos de los cuales ya no se recibe ninguna información.

La clase Interfaz implementa Runnable ya que pretendemos que se esté ejecutando constantemente. La forma de que el hilo termine y, a su vez, el programa principal, es mediante un WindowListener. Al detectar que se cierra la ventana, la variable booleana *cerrada* se convertirá en *false* y terminarán todos los procesos activos.



HexIdent	Callsign	Lat (°)	Lon (°)	Alt (ft)	Track (°)	GS (kt)
34260E	34260E	39.69598	0.11346	33000	307.27999999999975	441.8
345102	VLG13GJ	40.42296	0.97852	25000	168.46399999999994	404.4
484135	484135	40.36582	1.59258	33250	151.76	462.8
3C0A48	3C0A48	40.36181	0.73533	36000	548.048	505.3
3444C3	AEA6066	39.71448	-0.99367	31000	74.0	435.2
440176	440176	39.79537	-0.36334	27650	111.44000000000005	389.1
02005B	02005B	39.77572	-0.02948	33975	531.92	508.5
02007D	02007D	38.57374	-3.42566	39000	151.75999999999976	411.0
424549	424549	39.16283	-0.79137	33975	513.4879999999999	497.1
5110D5	5110D5	39.99696	-0.33240	35000	174.79999999999973	414.7
4072C2	4072C2	38.02533	0.06931	37000	94.73599999999988	399.0
48520C	TRA48V	40.63954	0.28652	33325	44.624	464.0
020110	020110	38.17140	-1.11581	35000	57.29599999999982	411.5
345107	345107	40.51195	1.21521	29200	168.46399999999994	437.2
342058	AEA1004	40.54985	-0.06714	24300	110.28800000000001	384.7
345686	IBE16FR	40.95698	0.32559	29000	170.192	384.7
4CA2A7	4CA2A7	39.73265	2.74216	38975	208.78399999999965	390.0
478369	SAS7526	41.28406	0.42059	36000	30.799999999999997	479.5
3964E1	3964E1	41.02300	-2.70151	35000	44.623999999999796	428.1
342090	342090	39.76154	3.40588	38000	209.36	502.9
5110F5	JAF59A	40.22360	-0.19594	31000	107.98399999999992	416.7
0A008A	DAH1087	40.76683	0.01947	37000	113.74399999999969	423.9
0A008C	0A008C	37.71725	1.06287	36000	523.856	523.3
345615	345615	38.94987	-1.24402	14200	348.75199999999995	356.1
0A007A	DAH1084	38.67906	0.54872	36000	166.736	495.3
34530D	ANE20CV	39.44499	-0.81651	26000	340.6880000000001	454.2
4243D6	4243D6	40.80597	1.37254	34025	551.504	501.9
4CA34A	4CA34A	40.20614	3.82362	37975	361.424	467.5
4401F4	4401F4	39.61075	3.76783	34325	620.624	487.8
4CAB97	4CAB97	38.07787	-2.03390	37000	266.38399999999956	375.8
345687	345687	41.33336	2.66119	36000	141.39199999999994	490.5
020124	020124	38.11368	-2.31625	35975	2.0	
396444	396444	39.72177	3.16321	39000	262.9279999999999	468.1
405456	405456	38.92378	1.22760	7225	95.88799999999992	235.6
4CAF8C	4CAF8C	38.68520	-3.56668	38000	42.32	485.9
34568D	34568D	41.75978	2.03713	30975	16.399999999999864	449.2
5110D6	5110D6	41.10699	3.63898	33325	99.92000000000007	417.2
4CA727	4CA727	40.90874	-2.68121	26950	143.6959999999999	435.2
342582	342582	38.93285	2.19965	17000	2.0	
3443D5	3443D5	38.99993	1.56104	8275	15.247999999999593	280.7
398501	398501	40.46571	3.15436	39000	275.5999999999999	451.0

Figura 3.3: Interfaz Principal

3.3. Clases para definir objetos

3.3.1. FlightObject

Esta clase auxiliar es la encargada de almacenar todas las distintas variables que podemos obtener de la antena. En un principio, al ejecutarse el constructor de esta clase, todos los valores desconocidos son iniciados a como ZZZZ siguiendo la práctica habitual en aviación. De esta forma, el controlador puede saber que este dato no es conocido por el programa y editarlo si lo conoce o, simplemente, esperar a que la antena lo reciba. Además, cuenta con una serie de métodos para actualizar la antena dependiendo del número de valores que esta nos aporta. Estos métodos son UpdateCallsign (empleado cuando solo recibimos el *callsign*) y los métodos UpdateMSG1 hasta UpdateMSG8 que, dependiendo del número de valores recibidos, ejecutaremos uno específico. Estos métodos cuentan como parámetro de entrada un vector de cadenas de caracteres, resultado de descomponer el mensaje de la antena en términos pequeños. Tomamos el término que corresponde a cada parámetro y le damos ese valor. Esta clase cuenta también con una lista de *setters* para ayudarnos a definir una serie de variables booleanas que emplearemos como indicadores de que una acción no debe ejecutarse si esta variable es verdadera. Estas son:

- **FP**. Empleada en la clase EnviarES, indica que ya se ha enviado el FP para ese

tráfico por lo que ya no debe enviarse otra vez.

- **conexion**. Empleada en la clase EnviarES, indica que ya se ha enviado el mensaje de conexión para este tráfico. Al actualizarse el *callsign* del tráfico tendremos que volver a enviar el mensaje de conexión por lo que volverá a ser falsa.
- *Añadir*. Empleada en la clase EnviarInterfaz, indica que se debe añadir una fila nueva a la tabla para el tráfico correspondiente. Se inicializa como *true* y solamente se vuelve falsa si ya existe una fila con el código hexadecimal correspondiente al vuelo.
- **Cli**. Es el socket correspondiente al tráfico. Hemos optado por crear un socket particular para cada avión ya que así evitamos la sobresaturación de un mismo socket y no tenemos problemas al enviar los mensajes al FSD. Los sockets los abriremos mediante el método openCli que pedirá la dirección IP y por defecto abrirá el socket en el puerto 6809, puerto del FSD.
- **CallsignValid**. Empleada en la clase EnviarES, se trata de una variable que nos indica que el callsign ha sido actualizado correctamente.

3.3.2. Traffic

El objeto principal de nuestro proyecto. Traffic será la clase encargada de almacenar todos y cada uno de los tráficos que recoja la antena. Para ello hemos empleado una herramienta de Java conocida como HashMap. Los HashMaps de java son una colección de objetos que nos permiten almacenarlos con una etiqueta o *key* que los diferencia unos con otros. Al introducir unos nuevos valores bajo la etiqueta correspondiente, se sobrescribirán los valores dentro del mapa. La clase Traffic hereda de un *ConcurrentHashMap* que almacena objetos FlightObject bajo una *key* que será una *String* del código hexadecimal. Los parámetros solicitados por el constructor para inicializar Traffic son el servidor de la antena de la UPV y el puerto de la antena, "servantena.etsid.upv.es" y 30002 respectivamente. Dentro del constructor se instancia la clase AntennaReceiver, que explicaremos a continuación, con los valores introducidos de entrada.

Dentro de la clase Traffic tenemos dos métodos muy importantes. El método EnviarES y message. El primero es un método genérico de envío de datos mediante sockets TCP. Tomará como parámetros de entrada el socket que queremos emplear y la cadena de caracteres que queremos enviar. Lo que hace este método es convertir en *bytes* la cadena a enviar y, mediante un DataOutputStream generado a partir del socket que introducimos, enviamos el mensaje. De producirse un error, el método comprueba si se ha podido establecer la conexión abriendo un socket auxiliar. Si la variable *exitosa* es falsa significa que no se ha podido crear la comunicación. Este método será empleado tanto en este programa como en el FSD2XP del proyecto al ser muy útil para la comunicación con el FSD.

El segundo método de Traffic es el encargado de transcribir el mensaje en bruto de la antena al objeto FlightObject. El primer paso es convertir el mensaje de la antena en un

vector de *strings* formado por los valores entre las comas del mensaje. Con este vector creado añadimos el FlightObject a traffic con el código hexadecimal (quinto elemento del vector) como *key*. Como explicamos en el Capítulo 2.1, el primer término del mensaje describe el tipo de mensaje que recibimos. Por lo tanto tenemos que filtrar y analizar solo los que nos interesan. Como ya hemos mencionado antes, la antena no siempre muestra el mismo número de valores. El segundo término del vector nos indica que método UpdateMSG tendremos que usar. Solamente nos basaremos en este valor si el tipo del mensaje es MSG. Si, por el contrario, el mensaje es SEL, ID o AIR emplearemos el método UpdateMSG1 siempre. De esta forma se actualizan las variables de cada tráfico.

3.3.3. AntennaReceiver

Esta clase, instanciada dentro del constructor de Traffic, se encarga de recibir los mensajes de la antena y almacenarlos dentro de él. AntennaReceiver hereda de la clase Thread y se ejecuta sin retardo, constantemente. A través del puerto y el host que le introducimos como entrada abre un socket cuya función es, simplemente, leer constantemente lo que haya en el *buffer*. Esta *string* será pasada como parámetro de entrada al método message para que lo identifique, lo filtre y almacene los valores dentro del respectivo FlightObject. Para que la antena funcione, debe estar conectada y funcional. Si la conexión a la antena fallase, esto se debería a que el programa que la inicializa no está funcionado o que simplemente se ha desconectado.

3.3.4. EnviarES

Esta es la clase mas compleja del programa. Se encarga exclusivamente al envío de los tres mensajes solicitados por el FSD y lo hará para cada uno de los vuelos. Al igual que la clase AntennaReceiver, hereda de Thread para poder ejecutarse constantemente con un intervalo de 200 ms. Los parámetros de entrada para instanciar este objeto serán traffic (ya que utiliza el método EnviarES), la dirección IP, el usuario, la contraseña y el nombre. Como tenemos que enviar mensajes para cada vuelo, tendremos que recorrer el mapa Traffic vuelo a vuelo. Para ello utilizaremos una *iteration*. Dentro de cada iteración el proceso sera el mismo. El primer paso es comprobar si el socket existe. De no ser así, se crea mediante el método openCli. Acto seguido filtraremos los mensajes de los cuales se conozca la latitud y la longitud. Hemos optado por este filtrado ya que, al observar los vuelos, aquellos mensajes que contaban con latitud y longitud solían ser bastante completos en términos de información. A partir de aquí entra en juego la variable CallsignValid que es la encargada de distinguir cuando se ha producido un cambio del *callsign* para un mismo tráfico. Si esta variable es falsa, significa que aun no ha existido dicho cambio. En este caso, se tomará este *callsign* como definitivo y se volverán a definir las variables conexion y FP como false. También se cerrará el socket para ese avión y se abrirá otro en su lugar. La variable CallsignValid pasará a ser *true* a partir de este momento. Si el *callsign* es ZZZZ, se tomará el código hexadecimal como tal.

Hecho esto es el turno de enviar los diferentes mensajes. Hay que tener en cuenta que el rumbo debe ser tratado para que el FSD lo interprete correctamente. Para ello

simplemente haremos la siguiente operación: $(track * 2{,}88 + 0{,}5) * 4$. A partir de ahora simplemente comprobaremos si el mensaje de conexión y el mensaje de FP se han enviado ya para este vuelo comprobando las variables FP y conexion del FlightObject correspondiente. Después de enviarlo nos aseguramos de definir las variables como *true*.Finalmente enviaremos el mensaje de posición.

3.3.5. EnviarInterfaz

Esta clase tiene como finalidad el control sobre la interfaz gráfica principal del programa. Mediante los tres métodos descritos en el apartado 3.2.3 se encarga de editar la tabla que el usuario visualiza. Hereda de Thread y presenta un retardo de 100 ms y, al igual que la clase anterior, recorremos el HashMap completo para actualizar la table. El procedimiento tráfico a tráfico es idéntico. Comenzaremos el bucle con un filtrado por latitud y longitud. Si tenemos datos de estos dos parámetros continuamos la evaluación del FlightObject. Aquí tendremos que tener en cuenta una variable guardada dentro del FlightObject denominada *CallsignAsig*, variable que manipulamos dentro de la clase EnviarES. Si esta variable está vacía significará que la antena no habrá recibido información del *callsign*. Si es *null* emplearemos el código hexadecimal al igual que antes. Si no lo es, emplearemos el valor esta variable como *callsign*.

Para empezar a recorrer la tabla, iniciaremos un bucle *for* de 0 al número de filas que tiene nuestra tabla en ese instante. El primer paso es comprobar si el *HexIdent* del tráfico evaluado en ese instante se encuentra dentro de la tabla. Si se lo encontramos, editamos la fila correspondiente con el método setValueAt y asignamos el valor de *false* a la variable *Añadir*. Si el código hexadecimal no se encuentra dentro de la tabla, *Añadir* será *true* por lo que añadiremos una fila con el método addRow2Table. Repetiremos este procedimiento entero para cada tráfico almacenado en Traffic.

3.3.6. TrafficCleaner

Por último, tenemos la clase TrafficCleaner que se encarga de eliminar los tráficos de los cuales ya no se recibe información tanto de Traffic como de la tabla de la interfaz gráfica. Al igual que las anteriores, hereda de Thread y se ejecuta cada 30 segundos. De esta forma damos tiempo a la antena a recibir mensajes ya que hay zonas, sobre todo cercanas al Aeropuerto de Valencia que son puntos ciegos para el SACTA. Cada 30 segundos se ejecuta el método clean_ lost_ flights(). En este método volvemos a utilizar un *iterator* para recorrer el mapa. Al contrario que en los casos anteriores donde daba igual si empleábamos una enumeración o una iteración, en este caso es obligatorio el empleo de esta última debido a que buscamos eliminar elementos de Traffic y la enumeración no permite este proceso. El proceso es simple. Se define una variable *long* que representa el tiempo actual real y se calcula la diferencia con el tiempo de recepción del mensaje de la antena. Si este es mayor que 30 segundos se procede a eliminarlo del mapa. Antes de esto debemos cerrar el socket previamente abierto para evitar que se acumule basura residual en el ordenador. Para eliminarlo de la tabla la recorremos buscando el *HexIdent*, ya que es posible que este mensaje nunca se haya llegado a enviar a la tabla al no cumplir con los requisitos mínimos, y lo eliminamos empleando el método removeRow.

CAPÍTULO 4

Implementar tráfico del FSD en el XPlane

En este capítulo se desarrollará el código del FSD2XP de manera detallada, a la vez que se explicará la funcionalidad de las distintas clases del proyecto. El objetivo principal de este programa es, mediante un filtrado, seleccionar los aviones más cercanos al manejado por el usuario de *XPlane* que encontremos almacenados en el servidor FSD y enviar su posición al simulador de vuelo para que poder verlos en tiempo real.

4.1. Clase Principal

La clase principal del proyecto se denomina MainFinal. En ella se ejecuta el método *main* del proyecto que instancia una a una todos los objetos necesarios para el funcionamiento del programa. Se ha reutilizado bastante código del programa anterior debido a las similitudes entre ambos programas. El primer paso será ejecutar la clase MensajeConexionXP. Esta interfaz gráfica solicitará al usuario los parámetros necesarios [1] para que la conexión con el *XPlane* sea la deseada. Si se quiere modificar estos valores fuera de los establecidos por defecto se debe estar seguro de que en el *XPlane* coinciden. De lo contrario, no se podrá establecer la conexión y el programa finalizará. También se pide el *callsign* del avión controlado desde el XPlane. Si se pretende utilizar en las prácticas de Gestión del Espacio Aéreo II, no tendremos necesidad de cambiar nada de lo predeterminado ya que nos interesa enviar los datos a la misma máquina que ejecuta el *XPlane*. Acto seguido, instanciaremos la clase Traffic. Esta clase es bastante similar a la del ITS. Al igual que la anterior, hereda de la clase ConcurrentHashMap con la clase FlightObject como valor almacenado para cada key. En esta clase se almacenarán todos los aviones extraídos del FSD. También se define, dentro de esta clase, un Mapa aparte en el cual se almacenan los aviones filtrados según la distancia entre ellos y el avión del *XPlane*. Dentro de este HashMap, el identificador será el *callsign*. Los métodos para hacer posible esto también se encuentran dentro de esta clase.

Después de ejecutar Traffic, se instanciará la interfaz gráfica MensajeUsuarioES, prácticamente idéntica a la del ITS. En ella se piden el usuario, la contraseña y el nombre para conectarse al FSD. Se comprueba que esta conexión es correcta. Si lo es, se pasará a ejecutar la última interfaz que compone este programa denominada MensajeAviones. En ella se pide el modelo de avión y el número de aviones que queremos representar dentro

[1] véase Capítulo 2.4

del *XPlane*. Hecho esto, ya tenemos la información necesaria para ejecutar el programa. Empezaremos por instanciar el método enviaXP de la clase Traffic que se encarga de marcar las casillas de datos que queremos que el *XPlane* nos envíe. Para ello, mediante un *socket* UDP enviaremos la etiqueta DSEL a las posiciones 3,20,17 y 104. Estas se corresponden con las casillas que observamos en la Figura 4.1 necesarias para hacer los cálculos de distancias para el filtrado. Esto también se puede hacer a mano pero de esta forma nos aseguramos de que están marcadas en el caso de que al usuario se le olvidase.

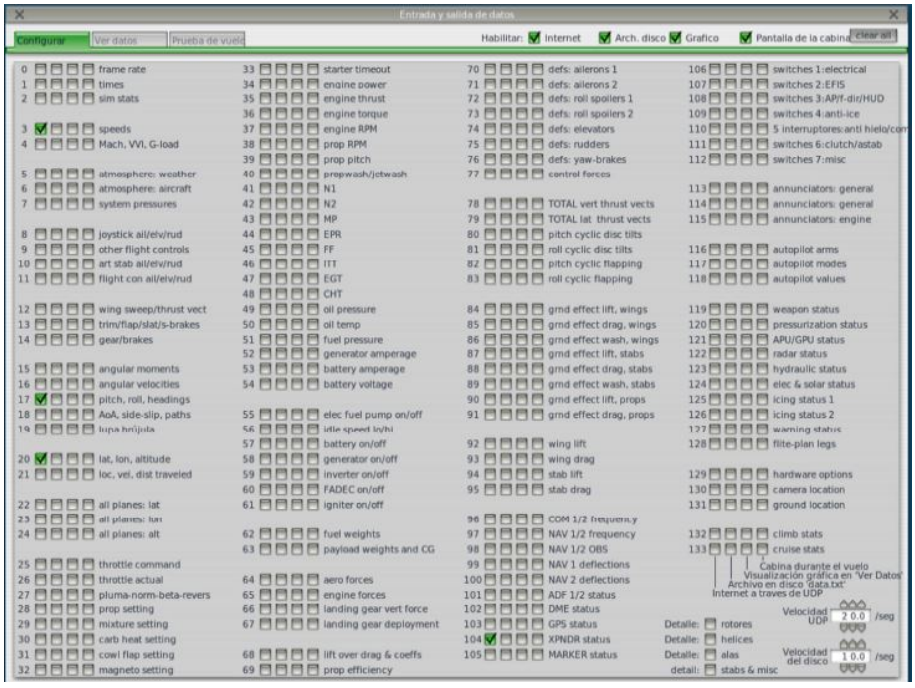


Figura 4.1: Entrada y salida de datos. XPlane

El siguiente paso es generar el socket TCP encargado de enviar la información de nuestro avión piloto al FSD. Como ya hemos mencionado en el Capítulo 2.2, el FSD automáticamente envía información de todos los vuelos que se encuentran dentro de un diámetro determinado a cada cliente conectado que envía su posición. Por lo tanto, tendremos que enviar constantemente la posición de nuestro avión piloto para recibir las del resto del tráfico y, así, efectuar el cálculo de las distancias con mayor precisión. Comenzaremos instanciando la clase RecibeXPlane que será la encargada de recibir esta información del *XPlane*, mediante *sockets* UDP. A su vez, será la encargada de enviar los tres mensajes necesarios para la conexión con el FSD mediante el *socket* creado anteriormente. Cada vez que se recibe posición desde el simulador, inmediatamente se envía al servidor. Ahora necesitamos recibir la información de cada tráfico del FSD. Para ello se emplea la clase RecibeES que, sabiendo la estructura de los mensajes que recibimos, almacena los datos de cada vuelo dentro de la clase Traffic.

Por último, falta realizar el filtrado de los aviones. Esto es, seleccionar los más cercanos (depende del número de aviones que hayamos decidido representar) y enviarlos al XPlane. La clase que se encarga de esta función es TraficosCercanos. Mediante mensajes

UDP realizaremos el envío de la posición de cada avión. El programa se ejecutará hasta que el usuario decide cerrarlo. Debemos mencionar que se ha creado una clase llamada Cerrar que es un hilo de Java que se encarga de comprobar, después de la ejecución de cada interfaz gráfica, si la ventana se ha cerrado de manera abrupta lo que provoca que su variable interna *cerrada* se convierta en *true*. Esta clase, si detecta este cambio, provoca que se cierre el programa completamente.

4.2. Clase Interfaz

4.2.1. Clase MensajeConexionXP

Esta clase es la primera interfaz que el método main instancia. Se despliega una ventana en la que, mediante el método iniciar(), se han creado una serie de **labels** para solicitar los datos de *XPlane* (puerto de recepción y envío de datos, por defecto 49.000 y 4902) y la dirección IP en la que se encuentra el FSD. El último parámetro que introducimos es el *callsign* del avión que manejamos desde el *XPlane*. Mediante un manejador de eventos, al pulsar el botón "OK" se almacenarán como variables los datos introducidos y, gracias a los *getters* de la clase, podremos utilizarlos en el resto del programa.

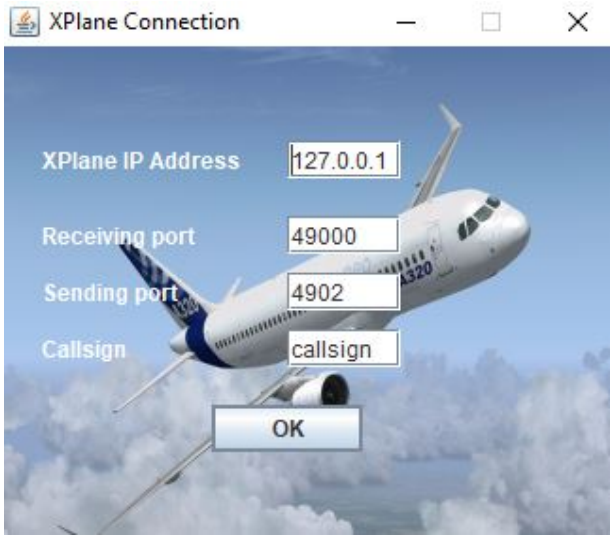


Figura 4.2: Clase MensajeConexionXP

4.2.2. Clase MensajeUsuarioES

La clase MensajeUsuarioES es prácticamente la misma que la explicada en la sección 3.2.2. Se genera una ventana que pide los datos necesarios para conectarse al FSD. Como parámetro se introduce el tráfico creado (para poder utilizar el método enviarES) y la dirección IP que recoge la clase MensajeConexionXP). Al introducirlos se debe pulsar

el botón "OK" para que el programa recoja los valores en las variables definidas en la clase. Se ejecuta el método validarUsuario() que se encarga de comprobar si se puede establecer la conexión con el FSD en la dirección IP introducida como entrada. Si se valida la conexión, se cierra la ventana y se procede a la instancia de la siguiente clase.

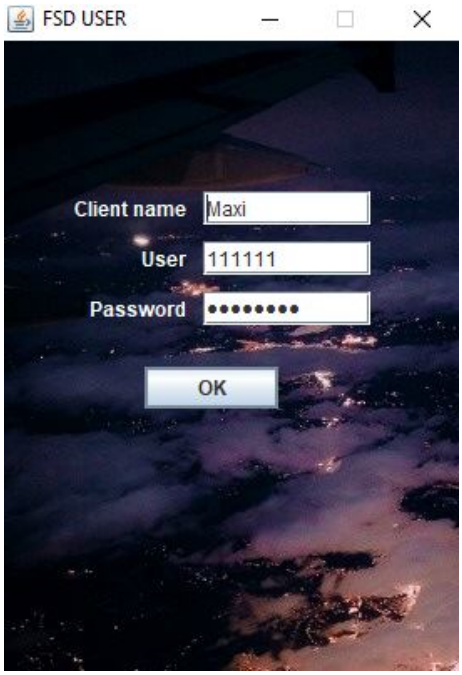


Figura 4.3: Clase MensajeUsuarioES2

4.2.3. Clase PlanDeVuelo

La última interfaz encargada de pedir información para enviar al FSD. En este caso, esta clase se encarga de solicitar al usuario el plan de vuelo, establecido previamente a volar. Los datos que se piden serán los necesarios para enviar el mensaje de plan de vuelo al FSD. Los campos que se deben introducir son los que están resaltados en rojo claro. Si no se introducen estos campos o se introdujesen incorrectamente, saltaría una venta a de error para poder corregirlo. Si no se introduce ningún valor en el resto de campos, se enviará como vacío al FSD.

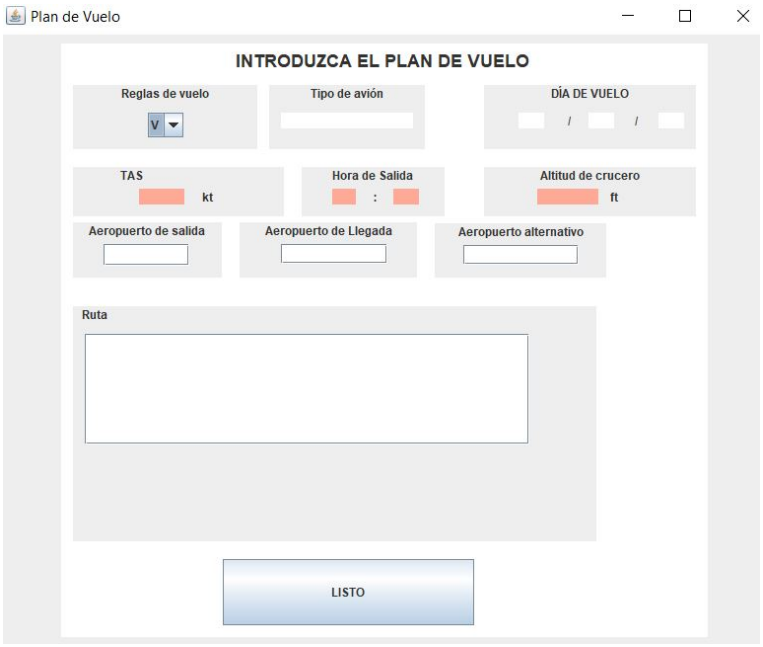


Figura 4.4: Clase PlanDeVuelo

4.2.4. Clase MensajeAviones

Esta clase, al igual que las anteriores, hereda de la clase JFrame por lo que genera una ventana cuando es instanciada. En este caso será empleada para solicitar el número de aviones que se van a enviar al XPlane y el tipo de avión que deseamos que muestre la simulación. El número de aviones tiene un rango de 1 a 19 ya que el simulador no soporta un número de aviones simulados mayor. Mediante el manejador de eventos que controla el botón "OK" obtenemos los valores introducidos y se evalúan. En el caso de que el usuario introdujese un número mayor a 19, saltaría una ventana emergente indicando que el número de aviones es incorrecto.

Figura 4.5: Clase MensajeAviones

Por otra parte, esta interfaz también solicita el tipo de avión que se quiere simular dentro de *XPlane*. Para ello tenemos en cuenta que el simulador, mediante un mensaje UDP, lee la ruta a la cual accede para obtener el modelo gráfico de cada avión. Estos modelos se encuentran dentro de la carpeta donde está instalado el *XPlane*, en la carpeta "Aircraft/General Aviation/". Por defecto, está escrita la avioneta Cessna 172SP, muy empleada en las prácticas de Ingeniería Aeroespacial debido a su facilidad a la hora de pilotar. En este proyecto no está implementado el método para definir el tipo de avión de manera externa al simulador y se propone como futuras expansiones de este proyecto.

4.3. Clases para definir objetos

4.3.1. FlightObject

Es un objeto librería donde simplemente se almacenan todos los parámetros que definen un avión para guardarlos dentro de la clase Traffic (explicada a continuación). Cuenta con el método actualizar que se encarga de, a partir de un vector de cadenas de caracteres, dar valor a las variables internas que definen un vuelo. El mensaje de entrada tendrá la estructura de un mensaje de posición del FSD. Por lo tanto, lo términos del vector serán por orden el *callsign*, el valor del transpondedor, la latitud, la longitud, la altitud, GS y el rumbo. Además, si la altitud recibida es mayor que 1640 pies, la variable Gear que representa el tren de aterrizaje será 1. Esta aproximación sólo es válida para zonas cercanas al nivel del mar como es Valencia.

4.3.2. Clase Traffic

Al igual que en el ITS, Traffic es el objeto principal de nuestro proyecto. Su objetivo consiste también en almacenar los distintos tráficos. Sin embargo, en este caso, nos centraremos en archivar los vuelos dentro del FSD (enviados por el ITS o por algún otro programa externo como el ATSIM). Para ello, el objeto hereda de la clase *ConcurrentHashMap* con los *callsign* como *key* y la clase FlightObject como valores almacenados. El constructor de la clase solicita como entrada los puertos de entrada y salida de datos del *XPlane* y la dirección IP en la que se encuentra el FSD y simplemente las almacena

como variables privadas internas. Como variable interna contamos también con el HashMap *aviones* encargado de, almacenar los distintos vuelos con su respectiva distancia ortodrómica al avión de referencia.

Esta clase cuenta con una serie de métodos nuevos con respecto al anterior Traffic. El primero de ellos es deg2rad es muy sencillo. Simplemente, a partir de un valor en grados, obtenemos su equivalente en radianes. Lo emplearemos para enviar los valores al *XPlane*, ya que los mensajes UDP deben contener el rumbo en radianes y el FSD nos lo aporta en grados. El segundo método auxiliar es ft2m, idem a deg2rad pero con distancias. Por último, dentro de los métodos auxiliares, tenemos distanciaOrto. Como parámetros de entrada solicita las coordenadas de los dos puntos en radianes y, a partir de ellos, calcula la distancia ortodrómica. Este cálculo no tiene en cuenta la altura, ya que se trata de distancias en el plano. Esto puede causar un leve error a la hora del filtrado para saber que avión es el más cercano al de referencia. Sin embargo, para el objetivo de este proyecto, nos es indiferente.

Pasamos a los métodos principales de la clase. En primer lugar contamos con EnviarES, cuyo funcionamiento y finalidad es idéntico al expuesto en la sección 3.3.2. Continuamos con enviaXP. Este método se encarga del envío mediante un mensaje UDP del mensaje DSEL a los parámetros de los cuales queremos que el *XPlane* nos envíe información. Las posiciones deseadas son la 3,20,17 y 104. Estas posiciones representan las velocidades del avión, los ángulos de Euler, la posición en el espacio y el estatus del XPNDR, respectivamente. Serán empleadas más tarde en la clase RecibeXP para obtener la posición del avión del piloto. El método principal de esta clase, el que se encarga del almacenamiento de los tráficos, es el método message. Cuenta como parámetros de entrada el mensaje recibido desde el FSD. Para ello nos hemos aprovechado de que el servidor reenvía todos los mensajes de posición almacenados (dentro de un diámetro predeterminado) al cliente que se conecte a él. El primer paso dentro de message es comprobar que, efectivamente, este mensaje recibido es un mensaje de posición de un tráfico. Para ello establecemos como elemento diferenciador que el mensaje comience con una @ (todos los mensajes de posición comienzan igual). Acto seguido tendremos que comprobar si el avión cuenta con un *callsign* o, en su defecto, está empleando su código hexadecimal como tal. Ya explicamos previamente que los tráficos recibidos sin *callsign* suelen tener datos espurios o imprecisos ya que suelen encontrarse muy lejos de la antena. Por lo tanto optamos por filtrar los vuelos que no lo tengan. Si se cumplen estos requisitos, el método generará un vector de *Strings* donde cada valor se corresponde con un término almacenado dentro del FSD (posición, TAS, rumbo...). Mediante el método actualizar definimos el FlightObject que queremos introducir en el HashMap. Además, utilizamos distanciaOrto para calcular la distancia e introducirla en el HashMap *aviones* como valor para cada *callsign*.

El último método de esta clase es checkAviones que se encargará de eliminar los vuelos de aviones que ya hayan sido eliminados de Traffic. Para ello, recorremos *aviones* mediante una iteración y comprobamos si los tráficos se encuentran dentro de Traffic. Si no lo encuentra, se borra.

4.3.3. Clase RecibeXPlane

Esta clase implementa la interfaz Runnable por lo que se tratará de un hilo de Java que se estará ejecutando constantemente. Su función principal es recibir los datos de XPlane enviados por el puerto de salida. Tiene implementada la clase Serializable que permite trabajar con *bytes*. Como parámetros de entrada tendremos las *Strings* necesarias para enviar el mensaje de conexión al FSD. También contaremos con la clase Traffic para poder utilizar sus métodos.

En primer lugar se crea un socket UDP con el puerto 4902 (puerto de envío de datos del *XPlane*. Acto seguido se crea un paquete para la recepción de datos en el cual, mediante el *socket* creado anteriormente, se almacenarán los datos recibidos. Tenemos que tener en cuenta que todos los mensajes de *XPlane* están formados por una cabecera de 5 *bytes* y un espacio de datos de 36. El número de mensaje que los definen viene incluido al principio tras la cabecera. Hallamos este índice empleando una sencilla fórmula [5+36k] donde k es el orden en el cual se solicita el mensaje dentro de enviaXP. Los mensajes contienen vectores de 4 *bytes* transformados a decimales con coma flotante. Simplemente, almacenaremos estos valores en una serie de variables locales que emplearemos a continuación.

Esta clase también se encarga de enviar inmediatamente después de recibir un mensaje con los datos de XPlane el mensaje de posición. Previamente comprobaremos si el mensaje de conexión ha sido enviado mediante la variable *conexion*. Obraremos de igual manera con el mensaje de plan de vuelo. Finalmente, ya podemos proceder a enviar el mensaje de posición. El *socket* se cierra al finalizar para evitar la acumulación de archivos residuales.

4.3.4. Clase RecibeES

La siguiente clase que se ejecuta en el main es la clase RecibeES. Al igual que la clase anterior, implementa las clases tanto Runnable como Serializable. Por lo tanto, consistirá en un hilo de Java que puede manipular *bytes* con facilidad. El objetivo de esta clase es recibir los mensajes enviados por el FSD al este recibir la posición que enviamos con la clase RecibeXP. La forma de comunicación es TCP (*Transmission Control Protocol*). Por otra parte, esta clase también se encarga de almacenar los tráficos en Traffic. Como entrada, esta clase solicita el *socket* necesario para la comunicación, la clase Traffic para emplear sus métodos y la clase RecibeXP en la cual están almacenadas las variables necesarias para conocer la posición del avión piloto.

Se comienza creando un DataInputStream a partir del *socket* que hemos introducido como parámetro de entrada. Mediante un bucle *while* comprobaremos si este *stream* contiene *bytes* dentro para poder leerlas mediante el método available. Si es distinto de cero, procedemos a leer los valores del canal. Para ello, creamos una cadena de caracteres a la cual le vamos añadiendo los *bytes*(transformados a caracteres previamente) hasta llegar a un retorno de carro, donde supondremos que el mensaje enviado por el FSD concluye.

Esta cadena de caracteres sera la que introduciremos en el método de Traffic message para poder filtrar los mensajes obtenidos. A su vez, se instancia el método checkAviones, para comprobar que el número de vuelos dentro de ambos HashMap coincide. Esta clase será ejecutada constantemente, siempre y cuando el DataInputStream contenga *bytes* dentro.

4.3.5. Clase TraficosCercanos

Se trata de la clase final del proyecto. Es la encargada de ordenar los aviones según su distancia al avión piloto y de enviarlos, vía mensajes UDP, al simulador de vuelo. Junto a Traffic es la clase más importante del programa. Como se estará ejecutando constantemente, implementará la clase Runnable. El constructor del objeto solicitará como parámetros de entrada la clase Traffic, una variable entera que simboliza el número de aviones que queremos enviar al simulador y el puerto de entrada de datos del *XPlane*. Al instanciar la clase se generarán una lista de *sockets* de tamaño el número de aviones y un HashMap donde se almacenan los callsigns de los aviones que vamos a enviar y el número de avión que se va a simular dentro del *XPlane*. Esta se genera en una primera estancia con todos los campos vacíos y se irán rellenando según se vayan ordenando los aviones. Esta clase cuenta con cuatro métodos principales para la ejecución del run del hilo. Estos son, en orden de ejecución:

- **SortHashMapValues**. Este método es el encargado de ordenar los distintos *callsigns* dentro del HashMap aviones según su distancia al avión piloto. Nos devolverá una lista llamada callsignsOrdenados con todos los *callsigns* en orden ascendiente según la distancia. Para ello se crean dos listas con las keys y con los valores del HashMap introducido como entrada. Empleamos el método sort para ordenar ambos HashMap. Acto seguido se crea el ArrayList que devolverá callsignsOrdenados. Lo que hacemos ahora es recorrer el array de los valores ordenados y buscamos, mediante un *iterator*, su key (*callsign*) correspondiente. Esta sera añadida en orden a callsignsOrdenados.

- **Listar**. Es el método mas complejo de esta clase. Se encarga de almacenar los *callsigns* que vamos a enviar al *XPlane*. Sin embargo, cuenta con una peculiaridad que la hace más compleja de lo que parece. Hay que tener en cuenta que cuando enviamos los mensajes UDP al *XPlane*, estos van acompañados de un número que simboliza el avión que queremos simular dentro del programa. Nosotros ocuparemos los números del 19 hacia abajo, ya que los primeros los representan los aviones del resto de usuarios empleados en las prácticas de GEA II. También hay que considerar que el array callsignsOrdenados estará cambiando de orden sus aviones, ya que en un momento puntual pueden haber un orden determinado y este puede variar al momento siguiente. El objetivo de este método es que, si un callsign se encuentra dentro de los aviones seleccionados para enviar, se mantenga dentro del HashMap bajo el mismo número de avión. Solo en el caso de que este desapareciera de los aviones más cercanos, sería sustituido ocupando el número que ha dejado el avión anterior. Dentro del *XPlane* simplemente veríamos como desaparece y se reposiciona en otro sitio.

- **send**. Recibe como parámetros la latitud, longitud, altitud, rumbo, estado del tren, el número de avión dentro del *XPlane* y el *socket* UDP que se va a utilizar para

cada avión. En primer lugar crearemos un array de *bytes* donde iremos poniendo en cola todos los mensajes que tenemos que enviar. El primer mensaje para que el simulador detecte que se le está enviando una posición de un avión es "VEH1". Es el identificador empleado para interpretar que se están enviando valores para simular una aeronave. Los parámetros que debe contener el array, en orden y trasnformados a *bytes* son: el número del avión, la latitud, longitud, elevación, el rumbo, el cabeceo y alabeo (supuestos a 0 ya que no los conocemos), el tren de aterrizaje, el estado de los flaps (supuesto a 0) y el empuje generado por el motor (supuesto a 0). A partir de este array generamos un paquete de datagramas que enviaremos a través del *socket* al simulador.

- **EnviarXP**. Este método simplemente recorre el HashMap Lista con todos los *callsigns* ordenados y su número de avión, recoge los parámetros necesarios de Traffic para ese *callsign* y los envía mediante el método anterior send.

Dentro del método run simplemente instanciaremos los métodos anteriores en el orden expuestos, repitiendo el proceso cada un segundo. Es posible que dentro del *XPlane*, al comenzar a recibir información, el avión comience a moverse desde el suelo en horizontal. Esto es un *bug* del simulador de vuelo y no se ha podido identificar a que se debe.

CAPÍTULO 5

Ejemplo de utilización

5.1. ITS

En esta sección haremos una demostración práctica del funcionamiento del primer programa que compone este proyecto. Se pretende observar el tráfico real de la antena y mostrarlo en el programa EuroScope, donde se podrá manipular a voluntad del usuario. En primer lugar ejecutaremos el .jar Programa1 para que arranque el programa. Al hacerlo nos aparecerá la siguiente ventana: El programa nos solicita que introduzcamos la dirección IP

Figura 5.1: Interfaz MensajeConexionES

de la máquina que está ejecutando el FSD. En este ejemplo, el FSD está siendo ejecutado desde la misma computadora por lo tanto introducimos la dirección *loopback*. El programa ya esta listo para conectarse al servidor. A continuación se nos abre otra interfaz en la que se nos solicitan los datos para establecer conexiones con el FSD.

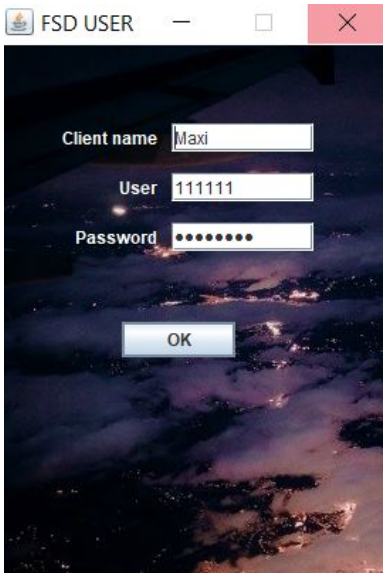


Figura 5.2: Interfaz MensajeUsuarioES

Introducidos estos datos, comienza la ejecución del programa y los valores obtenidos por la antena se pueden observar en la interfaz principal del programa, que estará constantemente actualizando los datos que se reciben.

Tráfico SACTA

Nombre: *Maxi*
Usuario: *111111*
IP: *127.0.0.1*

HexIdent	Callsign	Lat (°)	Lon (°)	Alt (ft)	Track (°)	GS (kt)
34260E	34260E	39.69598	0.11346	33000	307.27999999999975	441.8
345102	VLG13GJ	40.42296	0.97852	25000	168.46399999999994	404.4
484135	484135	40.36582	1.59258	33250	151.76	462.8
3C0A48	3C0A48	40.36181	0.73533	36000	548.048	505.3
3444C3	AEA6066	39.71448	-0.99367	31000	74.0	435.2
440176	440176	39.79537	-0.36334	27650	111.44000000000005	389.1
02005B	02005B	39.77572	-0.02948	33975	531.92	508.5
02007D	02007D	38.57374	-3.42566	39000	151.75999999999976	411.0
424549	424549	39.16283	-0.79137	33975	513.4879999999999	497.1
5110D5	5110D5	39.99696	-0.33240	35000	174.79999999999973	414.7
4072C2	4072C2	38.02533	0.06931	37000	94.73599999999988	399.0
48520C	TRA48V	40.63954	0.28652	33325	44.624	464.0
020110	020110	38.17140	-1.11581	35000	57.29599999999982	411.5
345107	345107	40.51195	1.21521	29200	168.46399999999994	437.2
342058	AEA1004	40.54985	-0.06714	24300	110.28800000000001	384.7
345686	IBE16FR	40.95698	0.32559	29000	170.192	384.7
4CA2A7	4CA2A7	39.73265	2.74216	38975	208.78399999999965	390.0
478369	SAS7526	41.28406	0.42059	36000	30.799999999999997	479.5
3964E1	3964E1	41.02300	-2.70151	35000	44.623999999999796	428.1
342090	342090	39.76154	3.40588	38000	209.36	502.9
5110F5	JAF59A	40.22360	-0.19594	31000	107.98399999999992	416.7
0A008A	DAH1087	40.76683	0.01947	37000	113.74399999999969	423.9
0A008C	0A008C	37.71725	1.06287	36000	523.856	523.3
345615	345615	38.94987	-1.24402	14200	348.75199999999995	356.1
0A007A	DAH1084	38.67906	0.54872	36000	166.736	495.3
34530D	ANE20CV	39.44499	-0.81651	26000	340.6880000000001	454.2
4243D6	4243D6	40.80597	1.37254	34025	551.504	501.9
4CA34A	4CA34A	40.20614	3.82362	37975	361.424	467.5
4401F4	4401F4	39.61075	3.76783	34325	620.624	487.8
4CAB97	4CAB97	38.07787	-2.03390	37000	266.38399999999956	375.8
345687	345687	41.33336	2.66119	36000	141.39199999999994	490.5
020124	020124	38.11368	-2.31625	35975	2.0	
396444	396444	39.72177	3.16321	39000	262.9279999999999	468.1
405456	405456	38.92378	1.22760	7225	95.88799999999992	235.6
4CAF8C	4CAF8C	38.68520	-3.56668	38000	42.32	485.9
34568D	34568D	41.75978	2.03713	30975	16.399999999999864	449.2
5110D6	5110D6	41.10699	3.63898	33325	99.92000000000007	417.2
4CA727	4CA727	40.90874	-2.68121	26950	143.6959999999999	435.2
342582	342582	38.93285	2.19965	17000	2.0	
3443D5	3443D5	38.99993	1.56104	8275	15.247999999999593	280.7
398501	398501	40.46571	3.15436	39000	275.5999999999999	451.0

Figura 5.3: Interfaz Principal

A partir de este momento sabemos con certeza que los vuelos reflejados en la tabla estarán almacenados en el FSD. Por lo tanto, ya es posible utilizar los vuelos recogidos en el servidor en otros programas externos. En nuestro caso emplearemos el EuroScope, ya que es una herramienta muy empleada en las prácticas de GEA II. El resultado es el mostrado en la Fig.5.4

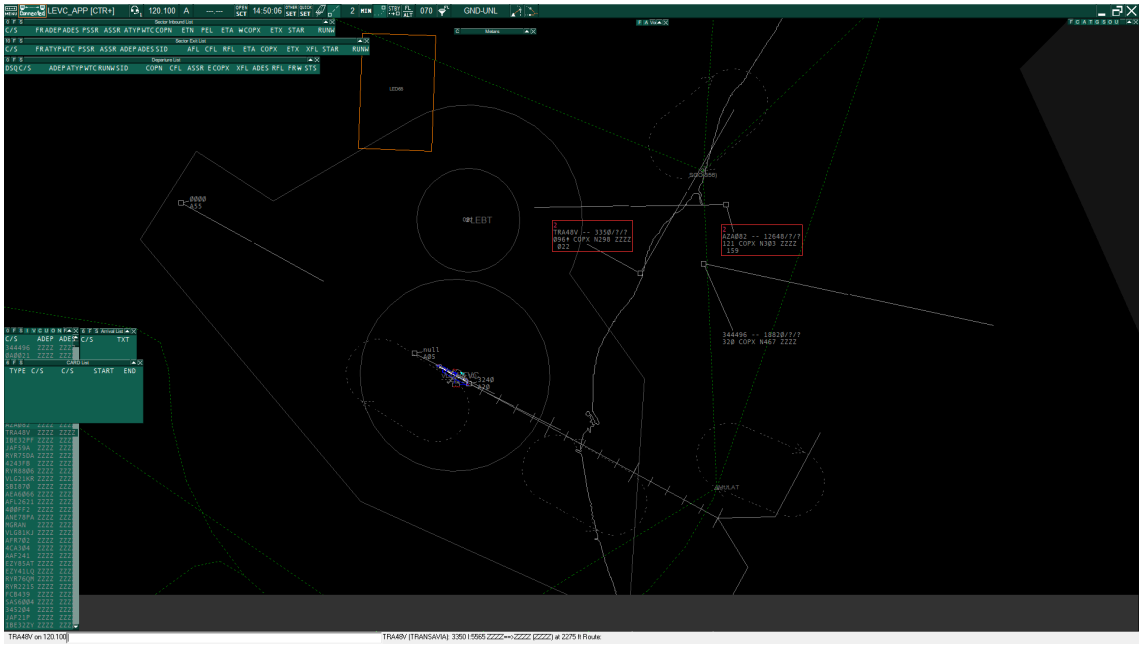


Figura 5.4: EuroScope

En la imagen anterior podemos una vista aérea sobre el espacio aéreo controlado del aeropuerto de Valencia. Se pueden apreciar dos aviones descendiendo por el oeste a punto de aterrizar. Podemos ver también dos aviones por el centro de la imagen con las etiquetas desplegadas en rojo que se encuentran en conflicto ya que uno se encuentra descendiendo y el otro ascendiendo.

5.2. FSD2XP

En esta sección se mostrará un ejemplo de ejecución del FSD2XP que compone este proyecto. Para ello haremos uso del programa de simulación de vuelo *XPlane 10*. El objetivo de este ejemplo es que el piloto, mientras usa el simulador de vuelo, pueda observar los aviones a su alrededor a partir de los datos obtenidos del FSD. El programa se inicia con la interfaz siguiente:

Figura 5.5: Clase MensajeConexionXP

Con ella el *software* solicita al usuario la dirección IP del FSD, los puertos de entrada y salida de datos del XPlane y el *callsign* que identificará el vuelo. No se recomienda cambiar los valores de los puertos ya que están establecidos como predeterminados dentro del simulador. La dirección IP que introduciremos será la de *loopback* ya que el FSD se está ejecutando en el mismo ordenador. La siguiente interfaz que se abre es la de conexión al FSD. Introducimos los datos solicitados para poder conectarnos al servidor.

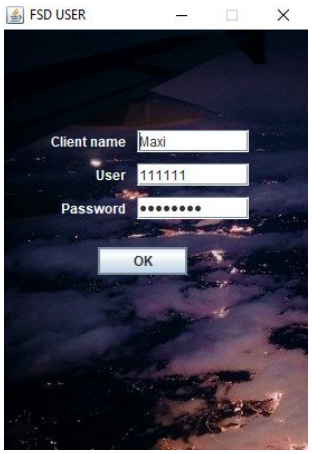


Figura 5.6: Clase MensajeUsuarioES2

A continuación introducimos el plan de vuelo deseado, de forma que se almacene en el FSD junto a nuestro identificador. Los parámetros en rojo se tendrán que rellenar obligatoriamente. Los siguientes datos a introducir son el número de aviones que se desea

Figura 5.7: Plan de vuelo

visualizar y el tipo de avión. Optaremos porque el número de aviones sea 2, ya que se desprecia el error de vibración que sufre la aeronave al recibir información el *XPlane*.

Figura 5.8: Aviones

Hecho esto, ya tenemos todos los datos necesarios para la ejecución del programa. Al cabo de unos diez segundos el programa comenzará a enviar información al *XPlane*, por lo que ya estará listo para comenzar la simulación.

CAPÍTULO 6

Conclusiones y trabajos futuros

Este proyecto ha sido diseñado con el objetivo de que sea empleado en estudios de tráfico aéreo que incluyan tanto tráfico real como tráfico simulado. Partiendo de proyectos anteriores, se deja abierta una línea de investigación que podrá ser continuada por alumnos en el futuro. Durante la realización de este proyecto se han ido encontrado una serie de dificultades, problemas y limitaciones a esperas de ser resueltas en un futuro. La mayor dificultad que hemos tenido es la fusión de los datos obtenidos por la antena que han provocado que el modelo de filtrado de mensajes sea muy preciso y con un sistema de actualización que implemente los nuevos datos a medida que se van recibiendo. Como problemas podemos encontrar la interpretación por parte del programa del tipo de avión que se introduce, que se ha dejado programado a medias. Esto se debe a que, a pesar de enviar la información mediante UDP de la forma que especifica el manual de *XPlane*, este no lee la información obtenida. Además, encontramos un problema en el simulador de que al enviar información de más de dos aviones simultáneamente, el avión local sufre una leve vibración que hace que se desplace a la izquierda. Esto es un error del *XPlane* que se procurará solucionar en proyectos venideros, ya que está fuera de los objetivos de este trabajo. Por otra parte, a la hora de diseñar el FSD2XP nos hemos encontrado con una serie de limitaciones como la escasa información que nos proporciona el FSD de una aeronave. Por este motivo, tanto el ángulo de cabeceo como el ángulo de alabeo se han tenido que suponer como cero, debido a la imposibilidad de obtenerlo. Como ampliación se propone la obtención de un modelo matemático de estos dos ángulos a partir de la variación de la posición y el rumbo en el tiempo.

Otra serie de ampliaciones a tener en cuenta pueden ser controlar el ordenador de abordo desde un programa externo o perfeccionar el sistema de filtrado por cercanía, de manera que identifique si los aviones que se encuentran más cerca del piloto no se corresponden con los controlados por los demás alumnos, de forma que no se observen duplicados dentro del programa. También se propone como ampliación al primer programa la obtención de los datos de bases de datos externas como podría ser Flight Radar. Incluso se podría obtener datos reales en formato .csv y enviar periódicamente al FSD, reproduciéndolo como si fuese un vuelo real. De esta forma el usuario podría simular situaciones reales dentro del *XPlane*. También se puede intentar una conexión con el programa EuroScope en la que, al controlar desde dentro un vuelo, se deje de enviar información sobre él y este pase a ser controlado mediante las ecuaciones del ATSIM.

Hablando ahora sobre mi experiencia personal, este proyecto me ha permitido exigirme mucho más dentro del campo de la programación orientada a objetos, ampliando así mis conocimientos del lenguaje Java. Estos conocimientos se basan en la comunicación mediante los protocolos UDP y TCP, el desarrollo de interfaces gráficas y la implementación final de todo este conjunto mediante hilos de ejecución y colecciones de Java. Además, mediante este trabajo he puesto en práctica lo aprendido en la asignatura Gestión del Espacio Aéreo II.

APÉNDICE A

Presupuestos

En este anexo proporcionaremos la información necesaria para la elaboración de los presupuestos del proyecto *Integración de tráficos en entornos de simulación*. Para ello se ha divido el trabajo en dos partes: programación del ITS y programación del FSD2XP, ambos desarrollados en el entorno de programación *NetBeans* del lenguaje Java. Se especificará para cada programa los costes directos, los beneficios y los impuestos.

A.1. Precios de la mano de obra

La mano de obra de este proyecto está formada por un ingeniero trabajando a tiempo completo, encargado del código de ambos programas en su totalidad.

Mano de obra					
Empleado	**Salario bruto** (€/mes)	**Horas de trabajo al día**	**Días de trabajo al mes**	**Salario** (€/h)	**Salario anual**
Ingeniero	1700	8	20	10.60	20400 €

Seguridad Social (€/mes)	**Coste anual del trabajador**	**Precio (€/h)**
510	26520 €	13.81

A.2. Precios de materiales

En esta sección se mostrarán los precios de los materiales empleados (ordenador y software) para el proceso siguiente.

Material	**Coste**	**Amortización**
Ordenador Lenovo 20245	765€	63.75 €
Licencia XPlane	60€	60€
Software Antena	800 €	100€
Total		**223.75 €**

A.3. Precios descompuestos

En este apartado hemos descompuesto los costes de cada tarea, dividiéndola en pequeñas tareas concretas y cuantificándolas según las horas dedicadas.

ITS			
	Horas	**Precio (€/h)**	**Importe (€)**
Desarrollo del código en Java	80	13.81	1104.8
Desarrollo de las interfaces gráficas	20	13.81	277.4
Fase de depuración	100	13.81	138.1

FSD2XP			
	Horas	**Precio (€/h)**	**Importe (€)**
Desarrollo del código en Java	40	13.81	552.4
Desarrollo de las interfaces gráficas	10	13.81	138.1
Fase de depuración	60	13.81	828.6
TOTAL	310	13.81	4281.1

Materiales			
	Cantidad	**Precio (€/ud.)**	**Importe (€)**
Ordenador Lenovo	1	63.75	63.75
Licencia XPlane	1	60	60 €
Software Antena	1	100	100€
TOTAL			4504.85 €

A.4. Presupuesto de ejecución material, presupuesto de inversión y presupuesto base de licitación

En esta sección se incluirán los gastos de gestión, los beneficios y los impuestos.

Concepto	**Coste**
Coste del proyecto	4504.85 €
Gastos generales (5 %)	225.24 €
Beneficio industrial (10 %)	450.485 €
TOTAL	**5180.575 €**
IVA (21 %)	1.087.9 €
TOTAL	**6268.5 €**

Bibliografía

[1] Servicio de flight tracking SACTA. Joan Vila, Enrique Hernández, José Simó. DISCA/UPV

[2] www.enaire.es

[3] http://dsp.mx/blog/sistemas-de-informacion/49-sockets-tcp-udp

[4] http://www.nuclearprojects.com/xplane/xplaneref.html

[5] https://docs.oracle.com/

[6] https://stackoverflow.com/

[7] Educational Framework fortheTeaching of ATM at the Universitat Politècnica de València (UPV). Pedro Yuste, Joan Vila, Hèctor Usach.