UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Sistema automático para vuelos de enjambres de multicópteros con guiado manual

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Masanet López, Joan

Tutor: Tavares Calafate, Carlos Miguel
Fabra Collado, Francisco José

Curso 2017 - 2018

Resumen

En la actualidad el uso de multicópteros se está extendiendo a gran velocidad. Existen gran cantidad de campos de aplicación donde los drones se pueden utilizar y una gran vía para su desarrollo de cara al futuro. En ocasiones se necesita que muchos de ellos realicen labores en cooperación, paralelización, así como permitir la redundancia de diferentes sensores, incluyendo la utilización de cámaras o infinidad de herramientas. Aunque hay algunas soluciones para la automatización de vuelos con enjambres de drones, en determinadas situaciones se requiere que el guiado sea manual.

En estos casos concretos, los drones de un enjambre deberán seguir al dron pilotado manualmente, que actuará como líder. Debido a ello, nace la necesidad de desarrollar un proyecto que permita que un dron actúe como líder del enjambre, siendo su trayectoria desconocida. Este hecho requiere, por lo tant,o una respuesta a las acciones del dron líder por parte del resto de drones que forman el enjambre. Para asegurar el correcto funcionamiento de la formación, este proceso se realiza en tiempo real.

Con la ayuda del software de simulación ArduSim de vehículos aéreos no tripulados, desarrollado por el Grupo de Redes de Computadores (GRC) de la UPV, se ha desarrollado un protocolo que permite realizar vuelos en enjambre con múltiples cantidades de drones, permitiendo adoptar distintas formaciones en el aire.

Previamente a la realización del protocolo, se ha desarrollado un elemento software implementado en Raspberry Pi, el cual permite recopilar información de un vuelo real, para más tarde replicar el vuelo en el simulador ArduSim.

Palabras clave: Dron, Enjambre, Raspberry Pi, Java

Resum

En l'actualitat l'ús de multicópters s'està estenent a gran velocitat. Existeixen gran quantitat de camps d'aplicació on els drons es poden utilitzar i una gran via per al seu desenvolupament de cara al futur. A vegades es necessita que molts d'ells realitzen labors en cooperació, paral·lelització, així com permetre la redundància de diferents sensors, incloent la utilització de càmeres o infinitat d'eines. Encara que hi ha algunes solucions per a l'automatització de vols amb eixams de drons, en determinades situacions es requereix que el guiat siga manual.

En aquests casos concrets, els drons d'un eixam hauran de seguir al dron pilotat manualment, que actuarà com a líder. A causa d'això, naix la necessitat de desenvolupar un projecte que permeta que un dron vaig actuar com a líder de l'eixam, sent la seua trajectòria desconeguda. Aquest fet requereix per tant una resposta a les accions del dron líder per part de la resta de drons que formen l'eixam. Per a assegurar el correcte funcionament de la formació, aquest procés es realitza en temps real.

Amb l'ajuda del programari de simulació ArduSim de vehicles aeris no tripulats, desenvolupat pel Grup de Xarxes de Computadors (GRC) de la UPV, s'ha desenvolupat un protocol que permet realitzar vols en eixam amb múltiples quantitats de drons, permetent adoptar diferents formacions en l'aire.

Prèviament a la realització del protocol, s'ha estat treballant en el desenvolupament d'un programari implementat en Raspberry Pi, que permet recopilar informació d'un vol real, per a més tard replicar el vol en dit simulador ArduSim.

Paraules clau: Dron, Eixam, Raspberry Pi, Java

Abstract

Currently, the use of multicopters is spreading fast. There are many fields of application where drones can be used, and a huge amount of possibilities to develop them further in the future. At some point, it could be necessary that many of them perform tasks in cooperation, parallelization, as well as allowing the redundancy of different sensors, including the use of cameras or a big variety of tools. Although there are some solutions for the automation of drone swarm flights, in certain situations manual guidance can be required.

In these specific cases, the swarm's drones must follow the manually piloted drone, which will act as a leader. That is why there is a necessity of developing a project that allows a drone to act as swarm leader, being its trajectory unknown. Therefore, it is required a response to the actions of the leading drone by the rest of the drones that form the swarm. To ensure the proper functioning of the full squad, this process is done in real time.

With the support of the simulation software ArduSim, which was developed by the "Grupo de Redes de Computadores" (GRC) of the UPV, and that allows deploying scenarios with multiple unmanned aerial vehicles, a protocol has been developed that allows swarm flights with multiple numbers of drones, allowing different formations to be adopted during flight.

Prior to the completion of the protocol, a lot of work has been done on the development of a small software implemented in a Raspberry Pi, which enables the information collection from a real flight, so as to replicate the real flight conditions in the ArduSim simulator later on.

Key words: Drone, Swarm, Raspberry Pi, Java

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

La tecnología principal en la que se centra el presente proyecto es toda aquella vinculada a los vehículos aéreos no tripulados del tipo multirrotor, comúnmente conocidos como drones. Los drones son capaces de volar mediante diferentes tipos de control de vuelo. Normalmente cuentan con pilotaje mediante una emisora radiocontrol, pero en otras ocasiones, cada vez con mayor frecuencia son pilotados desde aplicaciones en dispositivos móviles o tabletas. Incluso pueden ser vehículos autónomos que deciden por ellos mismo su propio camino, en base a una ”inteligencia programada” de seguimiento al usuario.

Los drones son utilizados para muchos fines y objetivos distintos. Algunos de los usos son los siguientes:

- Recreativo
- Industria
- Agricultura
- Policial
- Vigilancia
- Salvamento
- Militares
- Investigación
- Fotografía
- Reparto de mercancías ligeras

Las principales características que han causado el crecimiento en el uso de este tipo de vehículos en los últimos años son (i) su bajo coste económico respecto a un vehículo aéreo tripulado, y (ii) la facilidad que existe de adquirir un vehículo no tripulado de estas características. Además, el tamaño de las aeronaves tradicionales dificultan el proceso de ciertas tareas para las cuales los multicópteros son perfectamente adaptables.

Existen multicópteros de diferentes tipos y tamaños, de pocos centímetros hasta de algunos metros de extremo a extremo. En cuanto al número de motores y hélices, también existe una gran variedad, dependiendo del tamaño y de la utilidad del multicóptero.

Las aplicaciones de los multicópteros son muy variadas, y por ello esto requiere un desarrollo de funcionalidades genéricas que puedan ser adaptadas a cualquier tipo de actividad. Estas funciones necesitan ser implementadas por equipos de desarrolladores, y una de sus mejores herramientas son los simuladores de vehículo aéreo no tripulado (UAVs). En concreto, en el presente proyecto las aportaciones realizadas se centran en el uso de un simulador de multicópteros muy realista. Esto permite una mayor velocidad en el desarrollo de programas para los multicópteros, al mismo tiempo que garantiza un alto nivel de portabilidad del código obtenido.

En la actualidad existen varios simuladores de vuelo para multicópteros, pero en general están orientados a entrenamiento y/o simulación de carreras [1]. También existen algunos simuladores de código abierto que permiten realizar usos similares al tratado en el proyecto, como *AirSim* desarrollado por Microsoft [2].

Durante el proyecto se desarrollará un protocolo en el software de simulación ArduSim [24]. Este simulador, desarrollado por el Grupo de Redes de Computadores (GRC) de la Universidad Politécnica Valencia (UPV), permite recrear vuelos reales de una multitud de multicópteros volando en enjambre, permitiendo así cumplir con los propósitos del proyecto.

1.1 Objetivos

El objetivo principal del proyecto es desarrollar un sistema automático que permita realizar vuelos de enjambres de multicópteros con guiado manual, lo cual significa que habrá un piloto controlando manualmente el dron líder del enjambre, y los demás le seguirán automáticamente.

Como objetivos secundarios, los cuales permiten realizar el objetivo principal, se tienen en cuenta los siguientes:

- Obtener una traza de los controles recibidos desde una emisora radiocontrol a un dron principal, que lo denominaremos 'maestro'.

- Utilizar la traza del vuelo real para dirigir un único UAV en el simulador.

- Implementar una comunicación maestro - esclavo entre el UAV principal y el resto, permitiendo identificar los drones, dar órdenes de despegue, indicar que han finalizado el despegue y están listos, envío de coordenadas de posición actual del maestro y, por último, comunicar que tienen que aterrizar.

- Cálculo específico para cada UAV esclavo para determinar las coordenadas que tiene que alcanzar para seguir una formación establecida.

1.2 Estructura del documento

El presente documento se encuentra dividido en cuatro bloques. El primer bloque describe el estado del arte actual, los pasos previos y la configuración de Raspberry Pi 3, y la descripción de los componentes del multicóptero de diseño propio.

El segundo bloque describe el proceso de desarrollo de un software que permite obtener las trazas de un vuelo real, se describe, los movimientos que es capaz de realizar un multicóptero, y los mensajes MAVLink. Los contenidos del fichero guardado, también se describen en el presente bloque. Por último los pasos para ejecutar un software en Raspberry Pi al iniciar el sistema.

En el tercer bloque se realiza una descripción simple de la utilización de ArduSim y sus funciones más utilizadas, como los protocolos que se están desarrollando actualmente. También se describe con una máquina de estados, los pasos de la ejecución del software ArduSim.

En el cuarto bloque se encuentra la base del presente TFG, el protocolo desarrollado. En primer lugar se explica el funcionamiento del protocolo. A continuación, se detallan las versiones que se han implementado, incluyendo los beneficios de cada una de las versiones y los errores que han surgido. También se analiza las comunicaciones entre drones, la implementación de las formaciones, y el cálculo que realizan los UAV para mantener una formación. Por último se detalla la estructura del software y las mejoras realizadas respecto a la versión base.

Como parte final del documento se describen las conclusiones y los trabajos futuros.

Existe un apéndice que contiene los datos técnicos del multicóptero utilizado para la obtención de la traza en vuelo real y las características del PC donde se han realizado las simulaciones.

CAPÍTULO 2
Situación Actual

En el presente apartado, se detalla cual es la situación actual de los multicópteros y sus principales usos. Además se realizará un pequeño inciso en los inicios de los multicópteros o vehículos aéreos no tripulados.

2.1 Inicios

En los inicios de la creación y construcción de vehículos aéreos no tripulados, los pioneros Cayley, Stringfellow, Du Temple, intentando desarrollar aeronaves tripuladas, previamente construyeron las primeras aeronaves no tripuladas a lo largo de la primera mitad del siglo XIX. Estos modelos fueron las pruebas previas para la posterior creación de aeronaves de mayor tamaño y envergadura con piloto a bordo, siendo los precursores de la aviación tripulada.

Más adelante, las aeronaves no tripuladas evolucionaron como armas de guerra. Durante la primera guerra mundial, la evolución de la aviación convencional avanzó rápidamente. En cambio, la aviación no tripulada se quedó estancada por cuestiones tecnológicas.

Elmer Ambrose Sperry fue el primero en solucionar los problemas que existían de estabilización automática, control remoto y navegación autónoma, realizó pruebas exitosas con un giróscopo de aplicación marítima. [3]

2.2 Actualidad

En la actualidad, el uso de multicópteros se ha visto incrementado, formando parte de muchas tareas, tanto para realizar tareas inexistentes anteriormente, como en el pasado no se podían realizar con la facilidad que permiten los multicópteros actuales.

En general son una gran herramienta de trabajo que supone una evolución tecnológica muy importante. Esta tecnología avanza rápidamente, ofreciendo gran cantidad de prestaciones nuevas.

Existen gran cantidad de usos y ámbitos muy diversos, en el futuro podremos observar que se extenderán todavía más sus usos y prestaciones.

En España, el Ministerio de Hacienda ha utilizado drones para realizar vuelos de reconocimiento de obras sin declarar, con el uso de cámaras que registran el suelo. La revisión ha supuesto que los drones detecten 765.000 inmuebles sin declarar, tan sólo en la Comunidad Valenciana [5].

Las emergencias también se han visto modificadas con el uso de los drones. En el área del salvamento marítimo, los drones se han colocado junto a los vigilantes de las playas más concurridas durante los meses de verano [6]. Esto ayuda a que el dron pueda llegar lo antes posible al lugar del accidente y cargar algún elemento flotador. Además, ayuda a la prevención de posibles rescates obteniendo datos de las mareas, corrientes y fuertes oleajes.

El nuevo boletín de la ley española [7] también contempla la manera en el que se pueden utilizar las aeronaves no tripuladas, ya sea para el uso comercial o de ocio. El nuevo decreto ley se basa, en gran parte, en cuatro puntos clave los cuales, las empresas que quieran utilizar los drones como herramienta de trabajo, deberán contemplar, son los siguientes [8]:

- El tipo de dron, establecidos en dos categorías respecto sus características de peso.
- El espacio aéreo, controlado por la Agencia Estatal de Seguridad Aérea (AESA).
- La seguridad, haciendo cumplir el reglamento. Se exponen a sanciones entre 3.000€ y 60.000€.
- Carnet de piloto de drones en España, realizar unos exámenes teóricos y pruebas oficiales para los pilotos.

Las startup también se abren camino con la creación de nuevas tecnologías en el sector de los drones. Entre las muchas empresas emergentes, es interesante mencionar, la liderada por Pablo Flores, un ingeniero aeronáutico creador de un dron que puede actuar como bombero desde el cielo. Este tipo de aeronaves no tripuladas podrán complementar a los hidroaviones y helicópteros actuales. Este dron en concreto es capaz de nebulizar 300 litros de agua, y dispone de sistemas de control, geolocalización y cámaras térmicas que permite ubicar el punto exacto de la extinción. También se puede utilizar la misma tecnología para la fumigación en el ámbito de la agricultura. [9]

El proyecto actual podría beneficiar el uso de esta tecnología en conjunto con el del multicópteros apagafuegos, ya que realizando enjambres de este tipo de UAVs se puede extinguir el fuego en superficies más extensas en un tiempo menor. Igualmente permite acelerar tareas de sensorización de una determinada área, al permitir que cada dron cargue un sensor distinto y todos actúen sobre una misma zona simultáneamente.

CAPÍTULO 3
Solución propuesta

Este proyecto busca realizar un software que permita realizar enjambres de drones y que estos realicen una formación.

Por sencillez, el proyecto está dividido en dos grandes partes. La primera parte se trata del desarrollo de un software que permite realizar un registro de un vuelo real, obteniendo una traza. La traza tiene la función de complementar la segunda parte del proyecto. Esta primera parte comprende la realización de una aplicación JAVA que permite la adquisición en tiempo real de información de vuelo de un dron cuando se encuentra en vuelo. Los datos que capta el software y quedan registrados en la traza son los emitidos por una emisora radiocontrol. La necesidad de realizar la primera parte del proyecto, parte del uso del simulador ArduSim. La utilización de un simulador de aeronaves no tripuladas tiene cantidad de beneficios, tanto económicos como de tiempo de desarrollo. El simulador dispone de herramientas para poder simular una emisora radiocontrol, pero no se encuentra implementado. Durante el proyecto, se necesita esta funcionalidad. Por lo tanto se ha desarrollando un hilo que actúa como emisora radiocontrol. La emisora simulada permite recrear un vuelo real realizado en la primera parte del proyecto. Dicha emisora radiocontrol simulada, utiliza el contenido de la traza para controlar al dron maestro. Los datos enviados son los cuatro canales que contiene la traza, estos son enviados en los instantes de tiempo indicados también en la traza. Como consecuencia el dron maestro simulado, tiene un comportamiento similar al realizado en el vuelo real.

La segunda parte del proyecto permite realizar un enjambre de drones esclavos, estos siguen al maestro realizando una formación, por ello se ha diseñado un protocolo que coordina los dos roles, maestro y esclavo.

Entre los distintos UAVs existe una comunicación, que permite paso de información para realizar identificaciones, indicar que realicen acciones de despegue, indicar que están listos, realizar seguimiento de coordenadas y finalizar con el aterrizaje. Los mensajes tienen como parámetros como el identificador del dron del que procede el mensaje, y el tipo de mensaje. El resto de parámetros, si existen, son específicos de cada tipo de mensaje.

Se han implementado tres tipos de formaciones, formación Línea, formación Matriz y formación Círculo. Durante la implementación, se observo que parte de los cálculos se repetían periódicamente en cada iteración, así que se decidió calcular la posición de esclavo respecto al maestro (offset) una sola vez y lo antes

posible, en base a la formación escogida. Cuando el dron esclavo recibe el mensaje con las coordenadas del maestro, se aplica el offset y el cálculo se realiza de manera más simple, obteniendo el mismo resultado.

Por último, el algoritmo de seguimiento que se ha diseñado, garantiza que siempre que un esclavo recibe el mensaje de coordenadas, este mensaje activa los procesos que realizaran el desplazamiento adecuado para poder alcanzar la formación establecida.

En la Figura 3.1, se puede observar cual es el funcionamiento que realiza el protocolo desarrollado en el presente proyecto. El piloto del dron maestro, indica que el dron avance. El resto de drones del enjambre, actúan de manera idéntica.

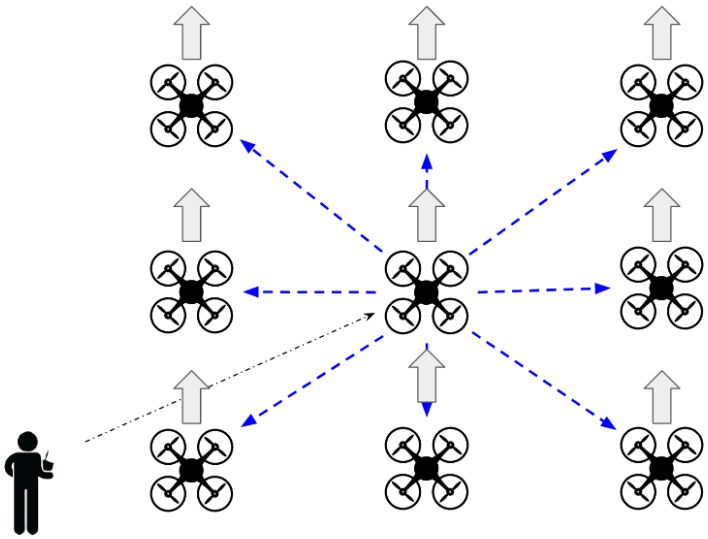

Figura 3.1: Esquema del funcionamiento del protocolo FollowMe

CAPÍTULO 4
Pasos iniciales

En el actual capítulo se realiza una explicación detallada de todo los pasos previos al desarrollo del sistema de guiado de drones, mencionando los componentes del dron que se utilizan, sus principales características técnicas y la estructura.

4.1 Componentes de los multicópteros

En el proyecto actual existen dos tipos de multicópteros, uno de ellos es el dron maestro y el resto son multicópteros esclavos. El proyecto es escalable, permitiendo utilizar gran cantidad de drones esclavos.

A continuación se detallan las principales características comunes para todos los drones de este proyecto.

4.1.1. Componentes de un dron

Para poder tener una fiabilidad durante el vuelo, cada dron debe de disponer de una serie de componentes que garanticen el correcto funcionamiento y la fiabilidad esperada. A continuación se especifican los componentes de un dron.

Chasis

El chasis es el cuerpo sobre el que se sustenta el propio dron. Todos los elementos, piezas y componentes están atornillados al chasis. Existen diferentes tipos de chasis para multicópteros, dependiendo de la función o utilidad para la que esté diseñado el multicóptero.

Los chasis más pequeños suelen tener entre tres y cuatro brazos donde van atornillados los motores con sus respectivas hélices, que proporcionan la sustentación del dron. Los chasis medianos suelen tener 4 o 6 brazos. Estos brazos dependiendo del chasis pueden ser intercambiables entre sí, para facilitar posteriores reparaciones. Los chasis más grandes pueden tener hasta 8 brazos o más, ya que el diseño del multicóptero depende exclusivamente del fabricante.

Las principales características sobre las que se escoge un chasis es por su forma y material de construcción. En cuanto a la forma viene dada por el diseño del dron y existen múltiples combinaciones. En la Figura 4.1 se observa un esquema simple de las combinaciones mas habituales en el diseño de un dron.

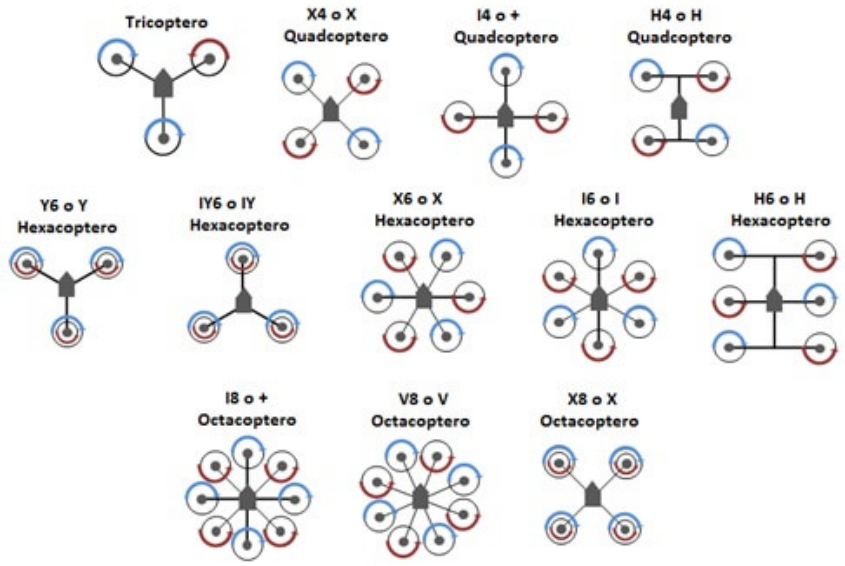

Figura 4.1: Tipos de multicópteros

Motores

Los motores son la parte móvil del multicóptero, se encuentran en los extremos de los brazos del chasis. Están conectados a las hélices y se encargan de hacer girar las hélices. El sentido del giro es un tema muy preciso, los giros de las hélices deben de estar compensados, si un motor gira hacia un lado el del lado contrario debe de girar en el sentido opuesto.

Como se puede observar en la Figura 4.2, la mitad de los motores giran al revés que el resto. Los verdes realizan giros a derecha y los de color azul giros a izquierda, realizando una compensación de giros y estabilizando el multicóptero. Esta es la configuración utilizada en el dron maestro empleado para obtener la traza de vuelo.

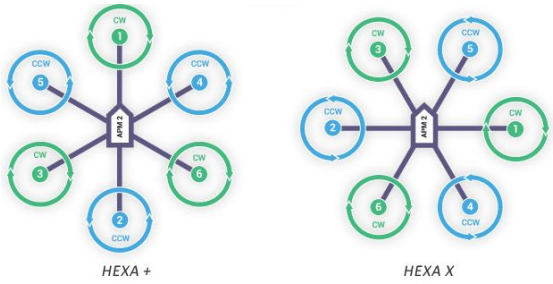

Figura 4.2: Sentido de giro de motores en un hexacóptero

En la Figura 4.3, al haber cuatro motores dos giran a la izquierda, los de color azul, y dos giran a la derecha, los de color verde.

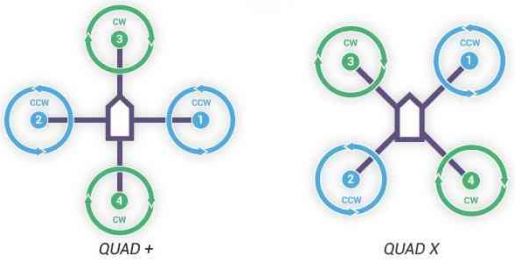

Figura 4.3: sentido de giro de motores en un quadcóptero

Hélices

La función que tienen las hélices es desplazar el aire y como consecuencia mover el dron. Existen diferentes tipos, tamaños, formas, etc. En general, los drones convencionales utilizan hélices bipala, o tripala, mientras que algunos de los drones para uso recreativo tienen hélices plegables, al realizar el giro se despliegan permitiendo sustentar al multicóptero.

Figura 4.4: Hélices

Controladores de velocidad

Los controladores de velocidad son los componentes que se encargan de proporcionar la electricidad a los motores, y regulan su velocidad aportando más o menos potencia según indica la controladora de vuelo. Como en cualquier tipo de hardware, existen de varios tipos según potencia y calidad.

Figura 4.5: Controladores de velocidad

Controladora de vuelo

Es el dispositivo que se encarga de mantener el control del vuelo, la orientación, estabilidad y velocidad según las ordenes proporcionadas por el mando radiocontrol. El dron maestro tiene instalada una controladora de vuelo denominada PIXHAWK, como se muestra en la Figura 4.6.

Figura 4.6: Controladora de vuelo

Baterías

La batería es el componente que almacena la electricidad, proporcionando corriente eléctrica a todos los componentes electrónicos. Una batería puede tener mucha capacidad para mantener el multicóptero más tiempo en el aire, pero en tal caso el peso aumenta.

Figura 4.7: Batería

4.1.2. Raspberry Pi 3.0

Además de las elementos que indicados, al dron utilizado como maestro se le ha añadido una Raspberry Pi 3.0 para su control.

Se ha optado por utilizar una Raspberry Pi 3.0 por sus prestaciones y la capacidad de computación que ofrece esta placa computadora.

Este dispositivo permite establecer un enlace de comunicación entre los drones mediante un adaptador Wifi. La capacidad de procesamiento que caracteriza a la Raspberry Pi 3.0 permite realizar cálculos para poder realizar movimientos de coordinación junto con el resto de drones.

4.2 Configuración de Raspberry Pi 3.0

En el siguiente apartado se explica cuales son los pasos para preparar el dispositivo para ejecutar el programa desarrollado.

La Raspberry Pi es compatible con multitud de sistemas operativos:

- Raspbian
- Ubuntu Mate
- Snappy Ubuntu Core
- Windows 10 IOT Core
- Pinewt
- Risc OS
- Chromium OS
- FreeBSD Raspberry Pi
- Alpine Linux
- CentOS

En la solución del proyecto utilizamos Raspbian, uno de los sistemas operativos más utilizados para Raspberry Pi.

Raspbian en un sistema operativo basado en Debian y orientado a la enseñanza de informática. Su primer lanzamiento fue en junio de 2012 y la última versión estable es de marzo del 2018.

La distribución utiliza como escritorio LXDE, que se trata de un entorno de escritorio libre para Unix. Además contiene herramientas para desarrollo en Python y Scratch.

Al ser una distribución Linux las posibilidades prácticamente son infinitas, pues todo el software de código abierto puede ser compilado en la propia Raspberry Pi.[4]

La instalación del sistema operativo se realiza de forma habitual en la placa según determina el desarrollador del sistema operativo.

Al iniciar la instalación escoge la opción de instalación con interfaz gráfica, como se puede observar en la Figura 4.8.

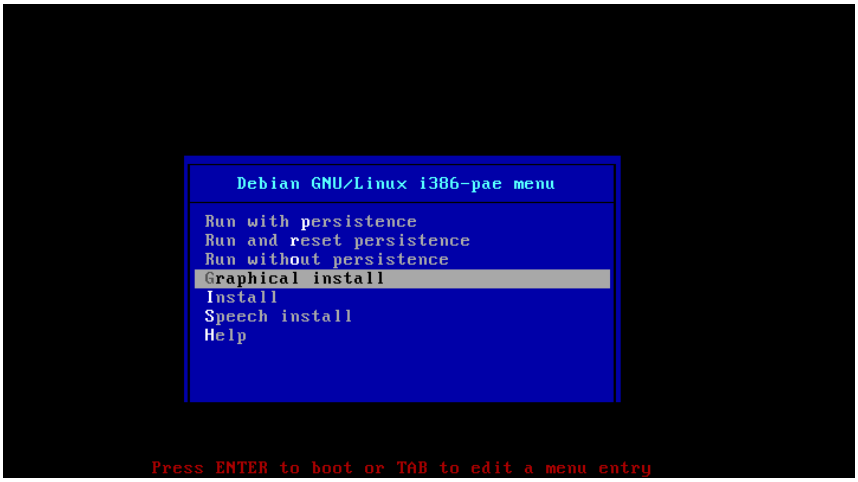

Figura 4.8: Menú inicial instalación Raspbian

Según los pasos indicados en la instalación. Obtenemos el sistema operativo instalado y funcionando ver Figura 4.9

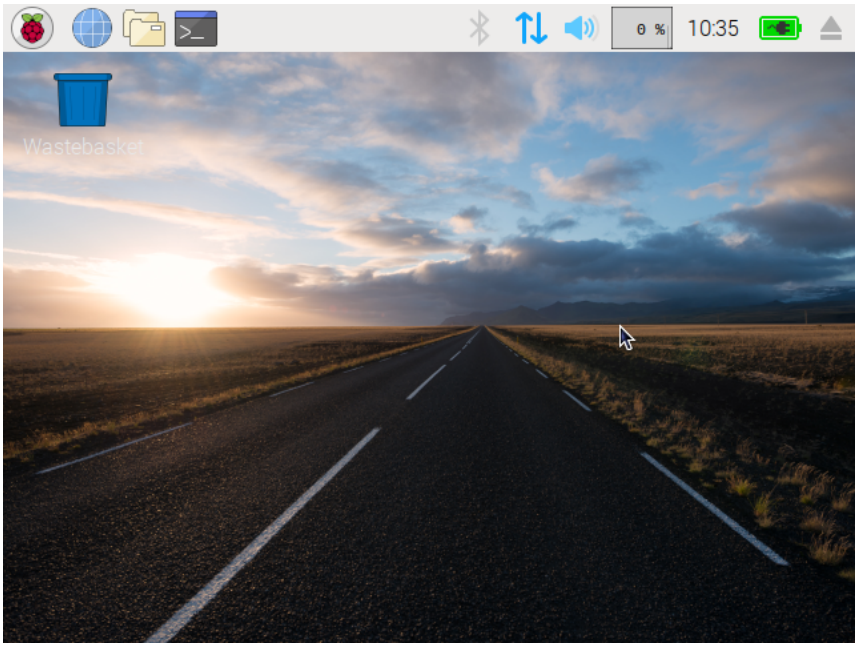

Figura 4.9: Escritorio Raspbian

A continuación, es necesario realizar la configuración de sistema para ejecutar la aplicación.

Raspbian tiene una opción por defecto habilitada, la salida de la consola del sistema operativo en el puerto serie. Esta opción hay que desactivarla, porque de lo contrario se estaría enviando datos que no son necesarios por el puerto, que se quiere utilizar como conexión entre las *Raspberry Pi* y la controladora de vuelo *PIXHAWK*.

Para ello hay que iniciar un terminal e introducir el siguiente comando.

```
sudo raspi-config
```

Se abre una utilidad de configuración donde se selecciona la opción de **Advanced Options**, la opción **Serial** y indicamos la opción **NO**.

La instalación del sistema operativo en ocasiones no contiene la instalación de java, para ello realizamos la comprobación con el siguiente comando.

```
java -version
```

Si falla la ejecución significa que java no esta instalado. Será necesario realizar la instalación:

```
sudo apt-get update
sudo apt-get install oracle-java8-jdk
```

4.3 Conexión Raspberry Pi 3 y controladora de vuelo

La placa Raspberry Pi debe estar conectada a la controladora de vuelo PIXHAWK. A continuación se explicaran los procesos que hay que realizar para hacer efectiva la conexión.

El multicóptero de construcción propia, tiene su origen en proyectos relacionados anteriormente encabezados en una línea de investigación previa. Los pasos de montaje de hardware serán obviados ya que no se han realizado durante el proyecto actual.

En primer lugar hay que habilitar el puerto serie donde se realiza la conexión. El modo de realizar esta acción es el siguiente.

```
sudo nano /boot/config.txt
```

Se abre un fichero en la terminal, hay que añadir al final del archivo la siguiente línea de texto.

```
enable_uart=1
```

Estos pasos serían suficientes para todos los modelos anteriores a la *Raspberry pi 3*, pero no en el caso actual que tiene conexión Bluetooth. Hay que realizar un intercambio de puertos, pues los pines GPIO utilizados por la conexión serie están siendo utilizados por el Bluetooth.

El cambio afecta a la calidad y velocidad del Bluetooth, pero en el proyecto actual no se utiliza dicha tecnología. Para realizar el cambio se siguen los siguientes pasos.

```
sudo nano /boot/config.txt
```

Con el comando del sistema se abre un fichero en la terminal, al que hay que añadir la siguiente línea de texto.

```
dtoverlay=pi3-miniuart-bt
```

Para realizar la comprobación de que todo se ha realizado correctamente, se introduce el siguiente comando.

```
ls -l /dev
```

Si todo está correcto, se muestra la siguiente configuración.

```
serial0 -> ttyAMA0
serial1 -> ttyS0
```

En la Figura 4.10 se puede observar la conexión entre los pines GPIO con la entrada de telemetría de PIXHAWK, este paso ya se ha realizado en anteriores proyectos.

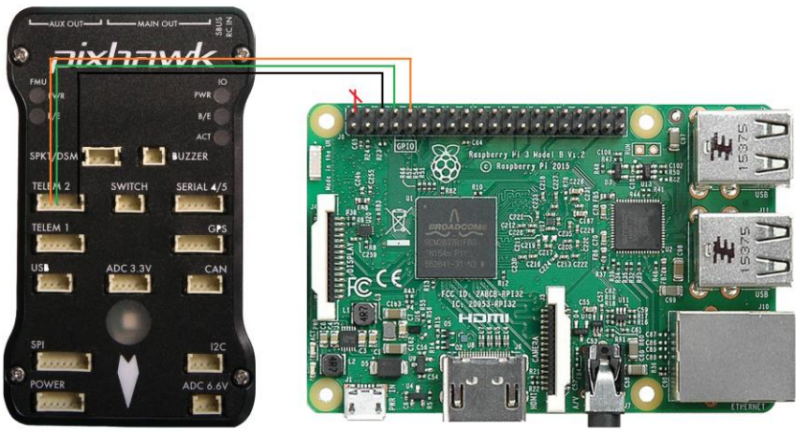

Figura 4.10: Esquema conexión Raspberry Pi y PIXHAWK

Una vez realizadas las configuraciones y las conexiones adecuadas. Hay que instalar las librerías y programas necesarios.

Las librerías que se instalarán son:

- screen
- python-wxgtk2.8
- python-matplotlib
- python-matplotlib
- python-matplotlib
- python-numpy
- python-dev
- libxml2-dev
- libxslt-dev

La instalación de todas las librerías se hace efectiva al realizar los siguientes comandos.

```
sudo apt-get update
sudo apt-get install screen python-wxgtk2.8 python-matplotlib
    python-opencv python-pip python-numpy python-dev libxml2-
    dev libxslt-dev
```

Y los programas siguientes:

- pymavlink
- mavproxy

Para instalar los programas, se realizan los siguientes comandos.

```
sudo pip install pymavlink
sudo pip install mavproxy
```

CAPÍTULO 5
Obtención de trazas

En la primera parte del proyecto se pretende obtener un registro de los comandos, enviados desde una emisora radiocontrol convencional al dron maestro.

El archivo de registro consiste en obtener los valores que emite el mando radiocontrol, y otros valores que se describen a continuación.

5.1 Control de un dron

Como se menciona en apartados anteriores, un dron convencional es generalmente controlado o conducido por una emisora de radiocontrol.

Las emisoras tienen varios canales para poder enviar información al dron, y cada canal representa un eje de coordenadas. La emisora de radiocontrol envía un valor por cada uno de estos canales. La controladora de vuelo procesa estos valores aplicando la velocidad adecuada a los motores para realizar el movimiento adecuado.

Canal RC	Movimiento
Canal 1	Roll
Canal 2	Pitch
Canal 3	Throttle - Acelerador
Canal 4	Yaw

Tabla 5.1: Canales RC

El Roll es la inclinación de lado a lado del dron. Este movimiento, si incrementa, hará que el dron se incline hacia la derecha, y un valor en decremento hará que el dron se incline hacia la izquierda.

Figura 5.1: Movimiento Roll

El Pitch es la inclinación hacia delante y hacia atrás del dron. Un valor creciente hará que el dron se incline y avance, y un valor decreciente hará que el dron se incline y retroceda.

Figura 5.2: Movimiento Pitch

El Throttle controla el movimiento vertical hacia arriba y hacia abajo del dron. Un valor incrementado hará que el dron vuele más alto, y una aceleración menor hará que el dron vuele más bajo.

Figura 5.3: Movimiento Throttle - Acelerador

Yaw es la rotación izquierda y derecha del dron. Un valor mayor hará que el dron gire hacia la derecha, y un valor menor hará que el dron gire hacia la izquierda.

Figura 5.4: Movimiento Yaw

5.2 Implementación del guardado de registros

Se ha implementado un programa en Java cuya finalidad es obtener un archivo que contenga los valores recibidos por el mando radiocontrol.

La finalidad del archivo, con toda la cantidad de datos recabados, es permitir replicar los movimientos realizados en un vuelo real de dicho dron en el simulador ArduSim.

Para obtener los datos de cada uno de los canales recibidos por el dron, la placa Raspberry Pi está conectada a la controladora de vuelo como se indica en el apartado 4.3. Esto permite ir obteniendo, en tiempo real, los cuatro valores que recibe de cada uno de los canales para ser controlado.

A medida que la placa Raspberry Pi hace la lectura en la controladora de vuelo, estos valores se van almacenando en un fichero que se crea al principio del experimento. El nombre del fichero viene dado por la fecha: día, mes, año y hora actual: hora, minutos, segundos. Esto permite realizar varias ejecuciones del programa sin sobrescribir cada uno de los archivos anteriores.

El programa en Java implementa una máquina de estados que trata de representar tres estados.

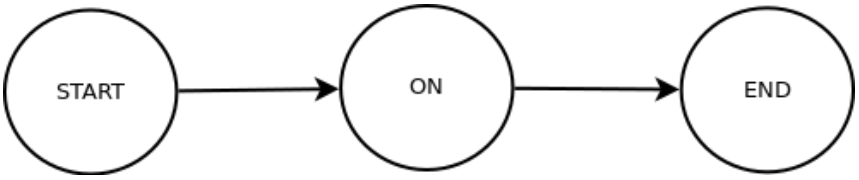

Figura 5.5: Maquina de estados Guardado de registros

Se ha utilizado un elemento enum [11] ya que permite realizar un código limpio y contiene funciones muy útiles para controlar el estado actual.

```
1  public enum Status {
2      START(0),
3      ON(1),
4      END(2);
5
6      private final int id;
7      private Status(int id) {
8        this.id = id;
9      }
10     public int getStateId() {
11       return this.id;
12     }
13   }
```

La función del estado *START* es crear el fichero con el nombre del día, mes y año actual concatenado con la hora, minuto y segundo actual. Una vez creado el *buffer*[12] de escritura, empieza a leer mensajes que recibe de la controladora de vuelo.

Los mensajes recibidos pertenecen al protocolo *MAVLink*, del cual se ofrecerán más detalles en el apartado 5.3.

Durante el estado *START* se van analizando los mensajes que recibe. Existen varios tipos de mensajes, en este caso solo se espera el tipo de mensaje *MAVLINK_MSG_ID_HEARTBEAT*, el cual se explica detalladamente en el apartado 5.3.

Avanzando un pequeño detalle, este tipo de mensaje contiene el estado actual del dron. Cuando el valor del estado es superior a 209 significa que el dron está volando.

Este es el valor que efectúa el cambio de estado, pasando del estado *START* al estado *ON*.

El estado intermedio *ON*, tiene la función de almacenar informacion dividida en dos tipos de mensajes *MAVLINK_MSG_ID_RC_CHANNELS_RAW* y *MAVLINK_MSG_ID_GLO*

BAL_POSITION_INT. Los diferentes tipos de mensajes restantes, que pueden recibir la placa Raspberry Pi de la controladora de vuelo, quedarán descartados ya que, por el momento, no son valores útiles para este proyecto.

Por último, la máquina de estados contiene un estado final llamado *END*. La función de dicho estado es terminar la ejecución. Concretamente, finaliza los *buffer*[12] de escritura, y cierra adecuadamente el fichero para que el contenido recopilado en el estado anterior quede guardado correctamente.

De nuevo, para realizar el cambio de estado de *ON* al estado *END* se utiliza el valor del estado actual del dron. Al descender por debajo de *209*, indica que el dron está en el suelo, y como consecuencia se realiza el cambio de estado.

5.3 Mensajes MAVLink

MAVLink es una librería que se utiliza para la comunicación directa con la controladora de vuelo, y está diseñada para drones y micro vehículos aéreos muy ligeros.

Se sigue un patrón de diseño híbrido entre publicación-suscripción y punto a punto. Los datos se envían como temas, mientras que los sub-protocolos de configuración, como el protocolo de una misión o el de parámetros, son punto a punto con retransmisión.

Los mensajes se definen dentro de archivos XML, que luego se pueden generar en el código fuente apropiado para cada uno de los lenguajes admitidos. Cada archivo XML define el conjunto de mensajes admitidos por un sistema en particular, también denominado *"dialect"*. El conjunto de mensajes de referencia implementado por la mayoría de las estaciones de control terrestre y los pilotos automáticos se define en *common.xml*[14]. MAVLink fue lanzado por primera vez a principios de 2009 por Lorenz Meier,y a día de hoy cuenta con un número significativo de colaboradores [13].

A continuación se explicarán los tipos de mensajes utilizados para realizar la adquisición de datos para el proyecto.

5.3.1. MAVLINK_MSG_ID_RC_CHANNELS_RAW

Este tipo de mensaje se utiliza para conocer los valores actuales que recibe la controladora de vuelo por cada uno de los canales. En el proyecto se utilizan los cuatro primeros canales correspondientes a *Pitch, Roll, Throttle* y *Yaw.*

Tabla 5.2: Estructura mensaje MAVLINK_MSG_ID_RC_CHANNELS_RAW

Nombre del campo	Tipo	Descripción
time_boot_ms	uint32_t	Marca de tiempo
puerto	uint8_t	Puerto de salida servo (conjunto de 8 salidas = 1 puerto). La mayoría de los MAV solo usarán uno, pero esto permite más de 8 servos.
canal1_raw	uint16_t	Valor del canal RC 1, en microsegundos. Un valor de UINT16_MAX implica que el canal no se usa.
canal2_raw	uint16_t	Valor del canal RC 2, en microsegundos. Un valor de UINT16_MAX implica que el canal no se usa.
canal3_raw	uint16_t	Valor del canal RC 3, en microsegundos. Un valor de UINT16_MAX implica que el canal no se usa.
canal4_raw	uint16_t	Valor del canal RC 4, en microsegundos. Un valor de UINT16_MAX implica que el canal no se usa.
canal5_raw	uint16_t	Valor del canal RC 5, en microsegundos. Un valor de UINT16_MAX implica que el canal no se usa.
canal6_raw	uint16_t	Valor del canal RC 6, en microsegundos. Un valor de UINT16_MAX implica que el canal no se usa.
canal7_raw	uint16_t	Valor del canal RC 7, en microsegundos. Un valor de UINT16_MAX implica que el canal no se usa.
canal8_raw	uint16_t	Valor del canal RC 7, en microsegundos. Un valor de UINT16_MAX implica que el canal no se usa.
rssi	uint8_t	Indicador de intensidad de la señal de recepción. 0: 0 %, 100: 100 %, 255: no válido / desconocido.

5.3.2. MAVLINK_MSG_ID_GLOBAL_POSITION_INT

El tipo de mensaje *MAVLINK_MSG_ID_GLOBAL_POSITION_INT* se utiliza para obtener los valores que proporciona el GPS a la controladora de vuelo: latitud, longitud, altura, y altura relativa para ubicarse en el espacio, así como velocidad en el eje de las X, Y y Z para establecer el movimiento que realiza. El rumbo viene dado por el valor del último parámetro, siendo 0 el norte geográfico.

Tabla 5.3: Estructura mensaje MAVLINK_MSG_ID_GLOBAL_POSITION_INT

Nombre del campo	Tipo	Descripción
time_boot_ms	uint32_t	Marca de tiempo
lat	int32_t	Latitud, expresada en grados * 1E7
lon	int32_t	Longitud, expresada en grados * 1E7
alt	int32_t	Altitud en metros, expresada como * 1000 (milímetros)
relative_alt	int32_t	Altitud sobre el suelo en metros, expresada como * 1000 (milímetros)
vx	int16_t	Velocidad expresada en el eje X (Latitud, positiva hacia el norte), expresada como m / s * 100
vy	int16_t	Velocidad expresada en el eje Y (Longitud, positiva hacia el este), expresada como m / s * 100
vz	int16_t	Velocidad expresada en el eje Z (Altitud, positiva hacia abajo), expresada como m / s * 100
hdg	uint16_t	Rumbo del vehículo (ángulo de guiñada) en grados * 100, 0.0.359.99 grados.

5.3.3. MAVLINK_MSG_ID_HEARTBEAT

El mensaje de *HEARTBEAT* muestra que un sistema está presente y responde. El tipo de hardware MAVLink y Autopilot permite que el sistema receptor trate los mensajes adicionales de este sistema apropiados.

En el proyecto se utiliza para obtener el valor *base_mode*, el cual indica si el multicóptero se encuentra en el aire.

Tabla 5.4: Estructura mensaje MAVLINK_MSG_ID_HEARTBEAT

Nombre del campo	Tipo	Descripción
type	uint8_t	Tipo de MAV (quadrotor, helicóptero, etc., hasta 15 tipos, definidos en MAV_TYPE ENUM)
autopilot	uint8_t	Tipo / clase del piloto automático. definido en MAV_AUTOPILOT ENUM
base_mode	uint8_t	Modo de sistema bitfield, como se define en MAV_MODE_FLAG enum
custom_mode	uint32_t	Un campo de bits para usar en indicadores específicos de piloto automático
system_status	uint8_t	Indicador de estado del sistema, como se define en MAV_STATE enum
mavlink_version	uint8_t_ mavlink_ version	La versión de MAVLink, que no se puede escribir por el usuario, se agrega por protocolo debido al tipo de datos: uint8_t_mavlink_version

5.4 Estructura de fichero de registro

En cada fichero se almacenan dos tipos de mensaje, uno de 11 datos y otro con 9 datos. Estos tipos de datos contienen varias cifras para poder obtener una gran precisión, sin embargo en el proyecto actual, la exactitud de los valores en este caso son irrelevantes, por lo que se redondean los valores obtenidos.

La traza que se ha diseñado consiste, en primer lugar, de un identificador, el cual es cero para la traza asociada al tipo de mensaje *MAVLINK_MSG_ID_GLOBAL_POSITION_INT*, y uno para la traza asociada al tipo de mensaje *MAVLINK_MSG_ID_RC_CHANNELS_RAW*.

La traza con el identificador cero, como segundo parámetro, contiene el tiempo actual dado en nanosegundos, tal y como es proporcionado por la función *System.nanoTime()*, la cual devuelve un número del tipo *long*. El tercer parámetro se trata de la longitud en la que se encuentra el dron, parámetro que se ha redondeado a tres decimales. Para el cuarto parámetro se aplica exactamente el mismo redondeo a tres decimales, ya que se trata de la latitud. El quinto parámetro es la altura a la que se encuentra el dron, redondeado a tres decimales. El siguiente parámetro es la altura relativa a la que se encuentra el dron desde el suelo como referencia, redondeada a tres decimales, ya que la exactitud de los sensores no es significativa a baja escala. El penúltimo parámetro es la velocidad a la que avanza el dron. Esta velocidad viene filtrada desde la controladora de vuelo, y se aplica un redondeo de tres decimales. El último parámetro indica el rumbo que toma el dron, y se expresa en grados. Se convierte en radianes para próximas aplicaciones.

```
String res = "0," +
    System.nanoTime() + "," +
  Listener.round(xy.Easting, 3) + "," +
  Listener.round(xy.Northing, 3)+  "," +
  Listener.round(z, 3) + "," +
  Listener.round(zRel, 3) + "," +
  Listener.round(speed, 3)+ "," +
  heading; // En radianes
```

La traza que contiene el identificador con el valor a uno permite representar sus parámetros de la siguiente manera. El primer parámetro contiene exactamente le mismo valor, que la traza anterior, el tiempo actual en nanosegundos. Los parámetros siguientes, contienen los valores de los canales. El valor que recibe la controladora de vuelo en cada uno de los cuatro primeros canales, correspondientes a Pitch, Roll, Throttle y Yaw, surgen en el mismo orden del primer canal al cuarto canal.

```
  String res =  "1," +
    time + "," +
    mensaje.chan1_raw + "," +
    mensaje.chan2_raw + "," +
    mensaje.chan3_raw + "," +
    mensaje.chan4_raw;
```

El siguiente apartado de código contiene un ejemplo de traza obtenida con el anterior formato. Como se puede observar, no se establece un orden en el tipo de traza si equivale al tipo cero o uno, pero sí que se establece un orden de tiempos. Los registros de la traza vienen dados por el orden de tiempo en el instante en el cual se recibe el mensaje, ya sea de tipo cero o uno, indistintamente.

```
1,26552460561,994,2006,982,2006
0,26719912071,694233.55,4399724.48,393.62,0.03,0.032,5.085540374461077
1,26781296446,994,2006,982,2006
1,26967649727,994,2006,982,2006
1,27161917956,994,2006,982,2006
0,27209023633,694233.53,4399724.48,393.62,0.03,0.03,5.085365841535878
1,27353364623,994,2006,982,2006
1,27553234102,994,2006,982,2006
0,27709118685,694233.5,4399724.48,393.64,0.05,0.04,5.085540374461077
1,27753513425,994,2006,982,2006
1,27968703946,994,2006,982,2006
1,28167908893,994,2006,982,2006
0,28209249935,694233.49,4399724.49,393.66,0.07,0.05,5.085191308610679
1,28353263320,994,2006,982,2006
1,28553701029,994,2006,982,2006
0,28709104622,694233.45,4399724.49,393.69,0.1,0.061,5.085714907386277
1,28753565195,994,2006,982,2006
1,28977172643,994,2006,982,2006
1,29173677747,1297,1661,982,1490
0,29229147174,694233.41,4399724.5,393.7,0.11,0.07,5.085365841535878
```

5.5 Ejecución en Raspberry Pi

Una vez completado correctamente el programa de Java, se tiene que exportar a un archivo ".jar". De este modo se puede ejecutar en cualquier sistema operativo que tenga instalado la máquina virtual de Java.

El programa desarrollado se ejecuta al iniciar el sistema. Para realizar esta función hay que crear un script que ejecute el programa ".jar".

5.5.1. Obtener archivo ejecutable

El proyecto se ha desarrollado en Eclipse, y por ello se detallaran los pasos a seguir para realizar todo el proceso bajo este entorno de desarrollo.

Para generar el archivo '.jar' hay que seguir los siguientes pasos. En el menú archivo, seleccionar la opción de Exportar.

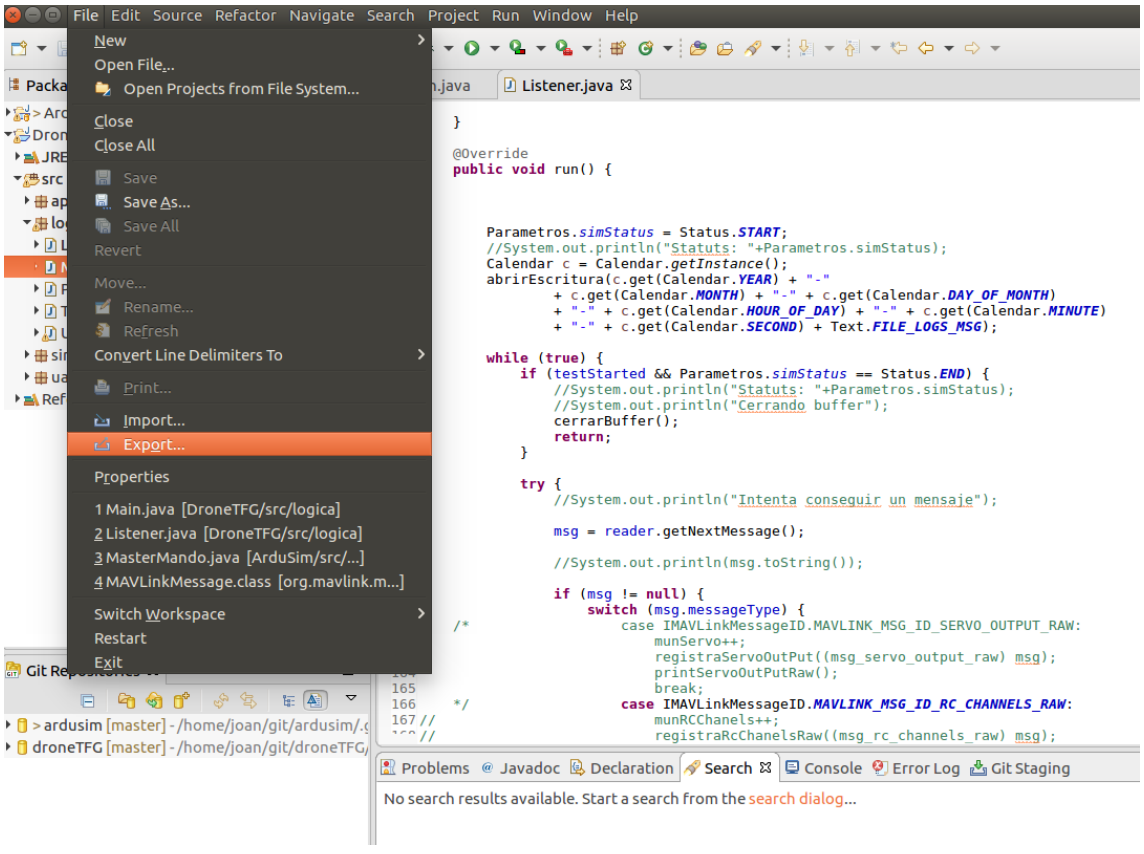

Figura 5.6: Menú Archivo, opción Exportar

Una vez creado el archivo ejecutable *DroneTFG.jar*, se copia en la placa Raspberry Pi y se ejecuta para comprobar que funciona correctamente. La opción más fácil es copiar el archivo en la ruta '/'.

```
1 $ java -jar DroneTFG.jar
```

5.5.2. Iniciar ejecutable Raspberry Pi

La configuración siguiente se realiza en la placa Raspberry Pi. Es necesario seguir los siguientes pasos para que se ejecute correctamente.

Se crea un nuevo script, el cual será el que el sistema ejecutará al iniciar. Este script debe de estar ubicado en la siguiente ruta.

```
$ sudo nano /etc/init.d/droneTFG-init
```

El contenido del script debe ser el siguiente:

```
#! /bin/sh
# /etc/init.d/droneTFG-init
java -jar /DroneTFG.jar
exit 0
```

Ahora se cambian los permisos para que el archivo sea ejecutable.

```
$ sudo chmod 755 /etc/init.d/droneTFG-init
```

Se comprueba que todo funciona correctamente

```
$ sudo /etc/init.d/droneTFG-init start
```

Por último, se realiza el arranque automático.

```
$ sudo update-rc.d droneTFG-init defaults
```

El conjunto de todas las anteriores configuraciones permite ejecutar el programa desarrollado al encender el dron, y junto al dron la Raspberry Pi 3. Una vez el dron se conecte al mando y reciba la información de los cuatro canales, el programa crea las trazas, que más tarde se utilizan para representar el vuelo en el simulador ArduSim, tal y como se detalla en el Capítulo 6.

5.6 Vuelo real y obtención de trazas

La obtención de las trazas se han realizando con vuelos en modo loiter. El modo loiter trata de mantener automáticamente, con la ayuda de sus sensores, la altura y posición alcanzadas, permitiendo total libertad de movimiento en horizontal sin necesidad de ajustar la altura desde los sticks del mando.

El UAV utilizado para la obtención de las trazas es un hexacoptero de diseño propio. El pilotaje que realizó el piloto, buscó trazar rutas sencillas y rectilíneas. Se formó un rectángulo; más tarde, en el capítulo siguiente, se puede observar en el simulador el trazado de la ruta real trazada. Durante el vuelo se obtuvieron parámetros de vuelo, posiciones de los sticks del mando, y la ubicación geográfica, que se transformaron en los datos de las trazas, guardando en cada instante de tiempo el contenido de estos valores.

Figura 5.7: Multicóptero obteniendo traza

CAPÍTULO 6
Software ArduSim

En el siguiente capítulo se trata a grandes rasgos cómo se utiliza el simulador ArduSim, desarrollado por Grupo de Redes de Computadores del Departamento de Informática de Sistemas y Computadores (DISCA), Universidad Politécnica de Valencia, en concreto por Francisco Fabra [22].

6.1 Simulador

ArduSim es un software de simulación basado en tres capas. La capa inferior se encarga, en general de la simulación de cómo se comporta un dron real. En la capa intermedia se configura todo el estado, misiones, coordenadas, movimientos, etc. La capa superior es la parte que ve el usuario, la cual se muestra como se inicia el dron, como se eleva, el estado en que se encuentra, etc.

La arquitectura de ArduSim se puede ver representada en la Figura 6.1, pero no es una cuestión para este proyecto incidir en la arquitectura a fondo. Se citarán algunos de los elementos que se observan en el esquema.

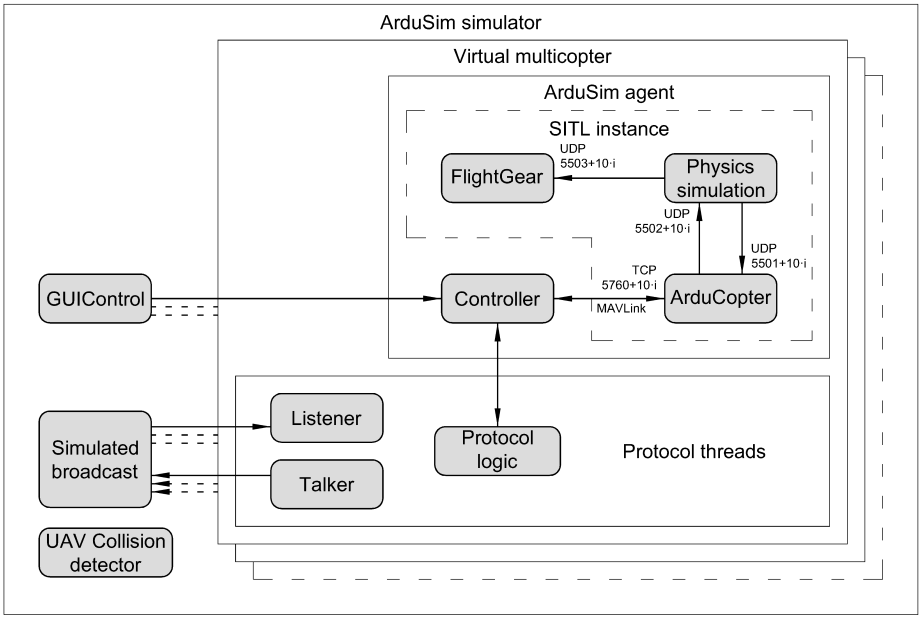

Figura 6.1: Arquitectura interna ArduSim

6.1.1. Setup ArduSim

En primer lugar, antes que cualquier configuración extra, se instala Java proporcionado por Oracle[19], recomendado por la documentación oficial de ArduSim [21]. La instalación en Ubuntu 16.04 consiste en ejecutar los siguientes comandos.

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

Los pasos siguientes son los ofrecidos por la documentación oficial de Ardupilot [20]. Se realizará la instalación de Git para poder descargar las versiones de Ardupilot.

```
sudo apt-get update
sudo apt-get install git-core
```

Una vez instalado git podemos realizar la descarga de ArduPilot con los siguientes comandos.

```
git clone git://github.com/ArduPilot/ardupilot.git
cd ardupilot
git submodule update --init --recursive
```

Se necesita instalar algunas librerías, como se indica en los siguientes comandos.

```
sudo apt-get install python-matplotlib python-serial python-wxgtk3.0
    python-wxtools python-lxml
sudo apt-get install python-scipy python-opencv ccache gawk git python-
    pip python-pexpect
sudo pip install future pymavlink MAVProxy
```

Se actualiza el path del sistema en el archivo */home/user_name/.bashrc*, y se añade al final del archivo de texto las siguientes lineas.

```
export PATH=$PATH:$HOME/ardupilot/Tools/autotest
export PATH=/usr/lib/ccache:$PATH
```

Se necesita recargar de nuevo el path del sistema, y para ello se ejecuta el siguiente comando.

```
. ~/.bashrc
```

Ejecutar el siguiente comando permite crear los archivos necesarios para poder simular todo tipo de aeronave, en este proyecto multicoptero.

```
cd $HOME/ardupilot/ArduCopter
sim_vehicle.py -w
```

Una vez finalizado todo el proceso, se cierra el programa con *CTRL + C*. Se obtiene el archivo Arducopter, el cual se utilizará en la carpeta del proyecto para ejecutar su contenido como librería.

6.1.2. Capa base ArduSim

ArduSim se basa en el simulador *Software In The Loop (SITL)*[23], que es capaz de simular con gran precisión diferentes tipos de UAV. *SITL* es un software de código abierto, multiplataforma, y que ofrece comunicaciones directas con el controlador de vuelo simulado, utilizando para ello una conexión TCP. Además permite la simulación de las características físicas de los UAV, y también es capaz de simular la presencia de viento. ArduSim está desarrollado de manera que cada instancia de *SITL* se ejecute como un proceso independiente. Por lo tanto, ArduSim crea una capa lógica de alto nivel donde se lanzan tantas instancias *SITL* como sea necesario para cada experimento.

6.1.3. Lógica de ArduSim.

El hilo del controlador transmite mensajes de control en el formato *MAVLink* a través de TCP a una instancia de *SITL*. Simultáneamente, recibe y procesa las respuestas a las instrucciones dadas, y los mensajes de información proporcionados por el controlador de vuelo virtual. Esta información (por ejemplo, posición actual, velocidad, etc.) puede ser utilizada por el protocolo que se está desarrollando con el fin de lograr la funcionalidad deseada. En UAVs reales, tales comunicaciones dependen de una conexión de puerto serie hacia el controlador de vuelo.

API

Para simplificar la implementación de los UAV, los desarrolladores de ArduSim han incluido una serie de funciones útiles para facilitar el desarrollo de nuevos protocolos. Las funciones de la API se encuentran distribuidas en las tres clases más importantes:

- GUI, las funcione relacionadas con la interfaz gráfica.
- Tools, las funciones relacionadas con el simulador.
- Copter, las funciones relacionadas con los UAV.

En los siguientes apartados se hace referencia a cada una de ellas, y a sus funciones.

GUI

Esta clase consiste exclusivamente de métodos estáticos que ayudan al desarrollador a validar y mostrar información en la interfaz gráfica. En la Tabla 6.1 se detalla la función de cada uno de los métodos, los parametros que necesita, y el resultado que devuelve.

Tabla 6.1: Métodos estáticos pertenecientes a la clase GUI

Método	Descripción
getUAVPrefix(int) - String	Devuelve un prefijo que identifica el UAV que está ejecutando un comando para fines de registro.
log(String) - void	Envía información al registro y a la consola de la ventana principal. El registro de la ventana solo se actualiza cuando se realizan simulaciones.
isVerboseLoggingEnable() - boolean	Devuelve verdadero si la función de registro detallado está habilitada.
updateGlobalInformation(String) - void	Envía información a la etiqueta de la esquina superior derecha de la ventana principal cuando un protocolo lo necesita. La etiqueta solo se actualiza al realizar simulaciones.
updateprotocolState(int, String) - void	Actualiza el estado del protocolo en el cuadro diálogo de progreso. El diálogo de progreso solo se actualiza al realizar simulaciones.
exit(String) - void	Finalización del programa cuando ocurre un error fatal. En un UAV real, muestra el mensaje en la consola y finaliza. En la simulación, el mensaje se muestra en un cuadro de diálogo, los UAV virtuales se detienen y finalizan.
warn(String) - void	Advierte al usuario con un cuadro de diálogo, al realizar simulaciones. En un UAV real, se utiliza la consola.
isUTMZoneSet() - boolean	Devuelve verdadero si se pueden realizar cálculos UTM a coordenadas geográficas, es decir, cuando se ha detectado la zona UTM.
locatePoint(double,double) - Double	Localiza un punto UTM en la pantalla, usando la escala de pantalla actual.
getUAVColor(int) - Color	Proporciona el Color asociado a un UAV y que debe usarse para dibujar elementos de protocolo.
getDetectedUAVs() - StatusPacket[]	Proporciona una lista con los UAV detectados por PCCompanion para ser utilizados por el diálogo de protocolo.

Tools

La clase Tools contiene todos los métodos estáticos que se utilizan para realizar cálculos. También contiene métodos que facilitan el desarrollo de nuevos protocolos en el simulador, control de los diferentes estados del simulador, métodos de espera, colisiones de UAVs, carga de misiones, detección de modo real, etc. A continuación se muestra en la Tabla 6.2 los métodos que contiene Tools y su descripción.

Método	Descripción
DATAGRAM_MAX_LENGTH - int	Parámetro TCP: tamaño máximo de la matriz de bytes utilizada en los mensajes
isRealUAV() - boolean	Devuelve verdadero si el protocolo se está ejecutando en un UAV real, o falso si se está realizando una simulación.
getNumUAVs() - int	Devuelve la cantidad de UAV que se ejecutan en la misma máquina. 1 Cuando se ejecuta en un UAV real (las matrices son del tamaño 1 y numUAV == 0). n Cuando se ejecuta una simulación.
setNumUAVs() - void	Establece el número de vehículos aéreos no tripulados que se simularán. Se utiliza únicamente en el cuadro de diálogo de configuración del protocolo, y cuando un parámetro limita el número de UAVs que deben simularse.
getIdFromPos(int) - long	Devuelve la identificación de un UAV. En el UAV real, se devuelve un valor basado en la dirección MAC. El UAV virtual devuelve la posición del UAV en las matrices utilizadas por el simulador.
setProtocol Configured(boolean) - void	Se utiliza esta función para afirmar que la configuración del protocolo ha finalizado cuando se cierra el cuadro de diálogo correspondiente. Para que los parámetros del protocolo funcionen correctamente, es necesario establecer valores predeterminados para que todos ellos se utilicen automáticamente cuando se carga ArduSim.
areUAVsNotAvailable() - boolean	Devuelve verdadero mientras los UAV están arrancando y no disponen de coordenadas válidas. Esta función se utiliza en la Fase 1.
areUAVsReadyForSetup() - boolean	Devuelve verdadero si los UAV están disponibles y listos para el paso de configuración. Esta función se utiliza en la Fase 2.
isSetupInProgress() - boolean	Devuelve verdadero mientras el paso de configuración está en progreso. Esta función se utiliza en la Fase 3.
isSetupFinished() - boolean	Devuelve verdadero si el paso de configuración ha finalizado pero el experimento no ha comenzado. Esta función se utiliza en la Fase 4.
isExperimentInProgress() - boolean	Devuelve verdadero mientras el experimento está en progreso.

Sigue en la página siguiente.

Método	Descripción
isExperimentFinished() - boolean	Devuelve verdadero si el experimento ha finalizado.
loadXMLMissionsFile(File) - List<Waypoint>[]	Carga misiones desde un archivo kml de Google Earth. Devuelve null si el archivo no es válido está vacío.
loadMissionFile(String) - list<Waypoint>	Carga una misión desde un archivo QGroundControl estándar. Devuelve null si el archivo no es válido o está vacío.
setLoadedMissions FromFile(List<Waypoint>[]) - void	Carga la primera misión encontrada en una carpeta. Carga prioritaria: primer archivo xml, segundo archivo QGRoundControl, tercer archivo txt. Solo se utiliza el primer archivo/misión válido encontrado. Devuelve null si no se encontró una misión válida.
getLoadedMissions() - List<Waypoint>[]	Establece las misiones cargadas desde el archivo/s para los UAV, en coordenadas geográficas. Las misiones deben cargarse y configurarse en el cuadro de diálogo de configuración de protocolo cuando sea necesario.
getUAVMission() - List<Waypoint>[]	Proporciona las misiones cargadas desde archivos en coordenadas geográficas. La misión solo está disponible una vez que han sido cargadas. Devuelve null si no está disponible.
getUAVMission(int) - List<Waypoint>	Proporciona la misión actualmente almacenada en el UAV en coordenadas geográficas. La misión solo está disponible si previamente se envía al dron con sendMission (int, List <Waypoint>) y se recupera con retrieveMission (int).
getUAVMission Simplifield(int) - List<WaypointSimplifield>	Proporciona la misión simplificada que se muestra en la pantalla en coordenadas UTM. La misión solo está disponible si previamente se envía al dron con sendMission (int, List <Waypoint>) y se recupera con retrieveMission (int).
isCollisionCheckEnabled() - boolean	Informa si la verificación de colisión está habilitada o no.
getCollisionHorizontal Distance() - double	Proporciona la distancia terrestre máxima entre dos UAV para afirmar que ha ocurrido una colisión.

Sigue en la página siguiente.

Método	Descripción
getCollisionVertical Distance() - double	Proporciona la distancia vertical máxima entre dos UAV para afirmar que ha ocurrido una colisión.
isCollisionDetected() - boolean	Informa cuando ha ocurrido al menos una colisión entre UAVs.
UTMToGeo(double, double) - GeoCoordinates	Transforma las coordenadas UTM en coordenadas geográficas. Ejemplo: Tools.UTMToGeo (312915.84, 4451481.33). Se supone que esta función se usa cuando se recibe al menos un conjunto de coordenadas del UAV, para obtener la zona y la letra de la proyección UTM, disponible en GUIParam.zone y GUIParam.letter.
geoToUTM(double, double) - UTMCoordinates	Transforma coordenadas geográficas a coordenadas UTM.
timeToString(long, long) - String	Devuelve el tiempo recuperados por System.curentTimeMillis (), desde la hora inicial hasta la hora final, en el formato h:mm:ss.
round(double, int) - double	Redondea un número double al numero de dígitos decimales.
isValidPort(String) - boolean	Valida un puerto TCP
isValidInteger(String) - boolean	Valida un número entero positivo
isValidPositiveDouble(String) - boolean	Valida un número double positivo
isValidDouble(String) - boolean	Valida un número double
getCurrentFolder() - File	Obtiene la carpeta donde se ejecuta el simulador (carpeta .jar o raíz del proyecto en eclipse).
getFileExtension(File) - String	Obtiene la extensión de archivo. Devuelve cadena vacía si no hay extensión de archivo
storeFile(File, String) - void	Almacena una cadena en un archivo.
isVerboseStorageEnabled() - boolean	Devuelve verdadero si la función de almacenamiento detallado está habilitada. Si se establece en verdadero, el desarrollador puede almacenar archivos adicionales para información no relevante.
waiting(int) - void	Hace que el hilo espere ms milisegundos.

Sigue en la página siguiente.

Método	Descripción
getExperimentStartTime() - long	Devuelve el instante en que el experimento comenzó en la hora local. Devuelve 0 si el experimento no ha comenzado.
getExperimentEndTime(int) - long	Devuelve el instante en que un UAV específico ha terminado el experimento en la hora local (la hora de inicio no es 0). Devuelve 0 si el experimento no ha finalizado, o aún no ha comenzado.
getUTMPath(int) - List<LogPoint>	Devuelve la ruta seguida por el UAV en la pantalla en coordenadas UTM. Útil para registrar datos de protocolo relacionados con la ruta seguida por el UAV, una vez que el experimento ha finalizado y el UAV está en tierra.

Tabla 6.2: Métodos estáticos pertenecientes a la clase Tools

Copter

Copter es la clase preparada para realizar todas las funciones sobre los UAVs, proporcionando acceso al estado del UAV, permitiendo cambiar de modo de vuelo, dar información sobre el estado de vuelo, motores, geolocalización, misiones. Permite realizar despegue de los UAVs y control sobre la emisora radiocontrol. En la Tabla 6.3 se indican todos los métodos de la clase Copter, y una descripción de cada uno de los métodos.

Método	Descripción
setParameter(int, ControllerParam, double) - boolean	Establece un nuevo valor para un parámetro del controlador de vuelo. Los parámetros que comienzan con "SIM_"solo están disponibles en la simulación. Devuelve verdadero si el comando fue exitoso.
getParameter(int, ControllerParam) - Double	Obtiene el valor de un parámetro de controlador de vuelo. Los parámetros que comienzan con "SIM_"solo están disponibles en la simulación. Devuelve el valor del parámetro si el comando fue exitoso. Devuelve nulo si ocurre un error.
setFlightMode(int , FlightMode) - boolean	Cambia el modo de vuelo UAV. Devuelve verdadero si el comando fue exitoso.
getFlightMode(int) - FlightMode	Obtiene el último valor del modo de vuelo proporcionado por un controlador de vuelo.
isFlying(int) - boolean	Devuelve verdadero si el UAV está volando.

Sigue en la página siguiente.

Método	Descripción
armEngines(int) - boolean	Arma los motores. Anteriormente debe haber presionado el interruptor de hardware por seguridad, si se encuentra disponible. El UAV debe estar en un modo de vuelo para poder ser armado (STABILIZE, LOITER, ALT_HOLD, GUIDED). Devuelve verdadero si el comando fue exitoso.
guidedTakeOff(int, double) - boolean	Despega un UAV previamente armado. Despega a la altitud indicada sobre el suelo. El UAV debe estar armado y en modo GUIADO. Devuelve verdadero si el comando fue exitoso.
startMissionsFromGround() - boolean	Hace despegar todos los vehículos aéreos no tripulados que se ejecutan en la misma máquina y comienza las misiones planificadas. Método concurrente (todos los vehículos aéreos no tripulados comienzan la misión al mismo tiempo). Previamente, en un UAV real, tienes que presionar el interruptor de hardware, si está disponible. Los UAV deben estar en el suelo y en un modo de vuelo varmable (STABILIZE, LOITER, ALT_HOLD, GUIDED). Devuelve verdadero si todos los comandos fueron exitosos.
startMissionFromGround(int) - boolean	Despega un UAV y comienza la misión planificada. Previamente, en un UAV real, tienes que presionar el interruptor de hardware, si está disponible. El UAV debe estar en el suelo y en un modo de vuelo armable (STABILIZE, LOITER, ALT_HOLD, GUIDED). Devuelve verdadero si todos los comandos fueron exitosos.
takeOffAllUAVsNonBlocking(double[]) - boolean	Hace despegar todos los vehículos aéreos no tripulados, cambia el modo a GUIDED, arma motores y luego realiza el despegue guiado. Método sin bloqueo. Matriz con la altitud relativa del objetivo para todos los UAV (asegúrese de que altitudes.length == Tools.getNumUAVs()). Devuelve verdadero si todos los comandos fueron exitosos.
takeOffAllUAVs(double[]) - boolean	Hace despegar todos los UAV uno por uno: cambia el modo a guiado, arma motores, y luego realiza el despegue guiado. Método de bloqueo Espera hasta que todos los vehículos aéreos no tripulados alcancen el 95 % de la altitud relativa objetivo. Matriz con la altitud relativa del objetivo para todos los UAV (asegúrese de que altitudes.length == Tools.getNumUAVs()). Devuelve verdadero si todos los comandos fueron exitosos.
takeOffNonBlocking(int, double) - boolean	Despega hasta la altitud relativa del objetivo: cambia el modo a guiado, arma motores, y luego realiza el despegue guiado. Método sin bloqueo. Devuelve verdadero si el comando fue exitoso.

Sigue en la página siguiente.

Método	Descripción
takeOff(int, double) - boolean	Despega hasta la altitud relativa del objetivo, cambia el modo a guiado, arma motores, y luego realiza el despegue guiado. Método de bloqueo: espera hasta que el UAV alcance el 95 % de la altitud relativa objetivo. Devuelve verdadero si el comando fue exitoso.
setSpeed(int, double) - boolean	Cambia la velocidad de vuelo planificada (m/s). Devuelve verdadero si el comando fue exitoso.
setCurrentWaypoint (int, int) - boolean	Modifica el waypoint actual de la misión almacenada en el UAV. Devuelve verdadero si el comando fue exitoso.
stopUAV(int) - boolean	Suspende temporalmente una misión en modo AUTO, ingresando en el modo de vuelo más lento para forzar una parada rápida. Devuelve verdadero si todos los comandos fueron exitosos.
setHalfThrottle(int) - boolean	Mueve el acelerador a la mitad de potencia usando el canal RC correspondiente. Devuelve verdadero si el comando fue exitoso. Útil para iniciar el vuelo automático cuando se está en el suelo, o para estabilizar la altitud al salir del modo automático.
channelsOverride(int, int, int, int, int) - void	Anula la salida del control remoto. Los valores del canal en microsegundos. Típicamente chan1 = roll, chan2 = pitch, chan3 = throttle, chan4 = yaw. El valor 0 significa que el control de ese canal debe devolverse a la radio RC. El valor UINT16_MAX significa ignorar este campo. Modulación estándar: 1000 (0 %) - 2000 (100 %). Los valores no se aplican de inmediato, sino cada vez que se recibe un mensaje del controlador de vuelo.
moveUAV(int, GeoCoordinates, float, double, double) - boolean	Mueve el UAV a una nueva posición. Coordenadas geográficas a las que el UAV debe moverse. Altitud relativa a la que el UAV debe moverse. Distancia horizontal desde el destino para afirmar que el UAV ha llegado allí. Distancia vertical desde el destino para afirmar que el UAV llegó allí. Método de bloqueo: Devuelve verdadero si el comando fue exitoso. El UAV debe estar en modo guiado.
clearMission(int) - boolean	Elimina la misión actual del UAV. Devuelve verdadero si el comando fue exitoso.
sendMission(int, List<Waypoint>) - boolean	Envía una nueva misión al UAV. Devuelve verdadero si el comando fue exitoso. El Waypoint 0 debe ser las coordenadas actuales recuperadas del controlador. El Waypoint 1 debe despegar. El último Waypoint puede ser land o RTL.

Sigue en la página siguiente.

Método	Descripción
retrieveMission(int) - boolean	Recupera la misión almacenada en el UAV. Devuelve verdadero si el comando fue exitoso. Nuevo valor disponible en UAVParam.currentGeoMission[numUAV]. Versión simplificada de la misión en coordenadas UTM disponible en UAVParam.missionUTMSimplified[numUAV].
simplifyMission(int) - void	Crea la misión simplificada que se muestra en la pantalla, obligando a reescalar.
cleanAndSendMissionToUAV(int, List <Waypoint>) - boolean	Elimina la misión actual del UAV, envía una nueva y la muestra en la GUI. Método bloqueante. Debe usarse solo una vez por UAV, y si el UAV debe seguir una misión. Devuelve verdadero si todos los comandos fueron exitosos.
setWaypointReachedListener(WaypointReachedListener) - void	Se utiliza para añadir a la lista de listeners aquel que ha alcanzado el waypoint (se puede establecer más de uno).
getCurrentWaypoint(int) - int	Obtiene el waypoint actual para un UAV. Úselo solo cuando el UAV esté realizando una misión planificada.
isLastWaypointReached (int) - boolean	Identifica si el UAV ha alcanzado el último waypoint de la misión. Úselo solo cuando el UAV esté realizando una misión planificada.
landIfMissionEnded(int) - void	Aterriza el UAV si está lo suficientemente cerca del último punto de recorrido. Úselo solo cuando el UAV esté realizando una misión planificada.
landUAV(int) - boolean	Aterriza un UAV si está volando. Método bloqueante. Devuelve verdadero si el comando fue exitoso o no es necesario.
landAllUAVs() - boolean	Aterriza todos los UAV que están volando. Método bloqueante. Devuelve verdadero si los comandos fueron exitosos o no necesarios.
getController(int) - UAVControllerThread	Método que proporciona el hilo del controlador de un UAV específico. Uso temporal para fines de depuración.
sendBroadcastMessage (int, byte[]) - void	Envía un mensaje a los otros UAV. Método bloqueante
receiveMessage(int) - byte[]	Recibe un mensaje de otro UAV. Método bloqueante. Devuelve nulo si ocurre un error fatal con el socket.
getPlannedSpeed(int) - double	Obtiene la velocidad planificada.
getData(int) - Quintet <Long, Double, Double, Double, Double>	Proporciona los últimos datos recibidos del controlador de vuelo. Tiempo, coordenadas UTM, altitud absoluta, velocidad, aceleración.

Sigue en la página siguiente.

Método	Descripción
getUTMLocation(int) - Double	Proporciona la última ubicación en metros UTM (x, y) recibidos del controlador de vuelo.
getGeoLocation(int) - Double	Proporciona la última ubicación en coordenadas geográficas (x = longitud, y = latitud) recibidas del controlador de vuelo.
getLastKnownLocations (int) - Point3D[]	Obtiene n últimas ubicaciones conocidas (x, y, z) del UAV en coordenadas UTM (altitud relativa). El tiempo de las ubicaciones aumenta con la posición en la matriz.
getZRelative(int) - double	Proporciona la última altitud relativa desde el suelo (m) recibida del controlador de vuelo.
getZ(int) - double	Proporciona la última altitud (m) recibida del controlador de vuelo.
getSpeed(int) - double	Proporciona la última velocidad de avance (m/s) recibida del controlador de vuelo.
getSpeeds(int) - Triplet<Double, Double, Double>	Proporciona los últimos componentes de tres ejes de la velocidad (m/s) recibida del controlador de vuelo.
getHeading(int) - double	Proporciona el último rumbo (rad) recibido del controlador de vuelo.

Tabla 6.3: Métodos estáticos pertenecientes a la clase Copter.

6.1.4. Interfaz gráfica

El cuadro de diálogo de la ventana de configuración se muestra cuando se inicia ArduSim Figura 6.2. Éste permite al usuario especificar varios parámetros de simulación, la velocidad del vuelo, algunos parámetros de rendimiento, el protocolo de sincronización que se probará, el modelo inalámbrico que se utilizará, y algunas de sus propiedades, ya sea para detectar cuándo ocurren las colisiones, además de definir la velocidad y la orientación del viento simulado.

Ventana de configuración del simulador

En la Figura 6.2 se pueden observar algunas de las configuraciones ya mencionadas anteriormente. Vamos realizar un explicación más detallada de cada uno de los parámetros de configuración.

En el apartado de parámetros de simulación, la primera opción que viene por defecto es el software compilado para simular una gran cantidad de vehículos aéreos no tripulados. Esta opción permite seleccionar un archivo diferente para comprobar distintas versiones y los efectos que causa. Durante el proyecto se ha ejecutado el protocolo desarrollado con distintas versiones, como se indica en los siguientes capítulos.

El segundo parámetro es un fichero de velocidades; este parámetro es indispensable en el simulador. Se trata de un fichero con extensión *csv* que contiene la velocidad máxima que puede alcanzar cada UAV determinando, por orden, des-

de el cero hasta el final del archivo. El fichero de velocidades que se ha utilizado se ha ido modificando para poder alcanzar mayor velocidad, como se mostrará con más atención en los siguientes apartados.

El último parámetro perteneciente a los parámetros de simulación es el número de UAVs que se generarán para la simulación. Este parámetro va condicionado por el archivo de velocidades, el número máximo de UAVs que permite establecer el simulador, y se trata de la cantidad de valores que contiene el archivo de velocidades, otorgando cada velocidad máxima a un UAV.

Respecto al apartado de parámetros de rendimiento, durante el transcurso de todo el proyecto no se ha necesitado modificar ningún valor. No obstante, se indica para que se utilizan cada uno. El primer parámetro se trata de el ratio de refresco de pantalla, y el valor por defecto es *500 ms*, el cual indica que, cada medio segundo, se actualiza la pantalla. El segundo parámetro indica la distancia mínima a la cual se tiene que mover un objeto para volver a dibujar en pantalla, y su valor por defecto es de *5 píxeles*. El parámetro para habilitar el consumo de batería se utiliza para simular una batería con la capacidad en *mAh* que se indica. La batería por defecto es ilimitada. El último parámetro indica la calidad del proceso de renderización, y en el proyecto actual no se ha modificado en ninguna de las ejecuciones.

Uno de los apartados más importantes de la configuración es el apartado de selección de protocolo. En este proyecto se han realizado algunas ejecuciones con el protocolo *none*, las cuales han servido para verificar que el simulador está funcionado correctamente. Para el resto de ejecuciones siempre se ha utilizado el protocolo *Follow Me*, ya que se trata del protocolo que se ha desarrollado en este proyecto.

En el apartado de los parámetros de comunicación entre UAVs se destacan dos del total de los parámetros. Uno de ellos es la detección de colisiones entre paquetes; este parámetro se ha habilitado y deshabilitado para verificar el correcto funcionamiento del protocolo. Por defecto se encuentra habilitado, simulando la existencia de colisiones entre los paquetes de la comunicación. El segundo parámetro a destacar es el tipo de conexión inalámbrica. Este parámetro permite simular el alcance de una antena *WiFi*, definiendo la distancia máxima que alcanza la señal. El valor por defecto es de distancia ilimitada, y durante las ejecuciones de este proyecto se ha utilizado esta configuración ya que este ámbito no esta contemplado en este proyecto.

La detección de colisiones entre drones forma parte del simulador. Este apartado permite que los UAV detecten si están cerca de colisionar con otro UAV. Los parámetros relacionados permiten el indicar cual es el periodo entre comprobaciones de detección de colisiones, que por defecto está establecido con un valor de *0.5s*. La distancia en un radio horizontal a la que se detecta que los drones pueden colisionar, se encuentra por defecto a *10 metros*. Por último, respecto a la diferencia de altitud a la que se detecta que pueden colisionar, se toma el valor por defecto de *20 metros*.

El último apartado establece la simulación de ráfagas de viento. Esta característica la proporciona el software *SITL* mencionado anteriormente. Los parámetros que se pueden configurar son la dirección y la velocidad del viento. Durante

las ejecuciones de este proyecto se han modificado en alguna ocasión, como se detalla en los siguientes capítulos.

Por último, en la parte baja de la ventana hay dos botones: el primero inicia la ejecución del simulador ArduSim con la configuración establecida en los apartados anteriores y con los parámetros correspondientes. El segundo botón restablece todos los parámetros a la configuración inicial por defecto.

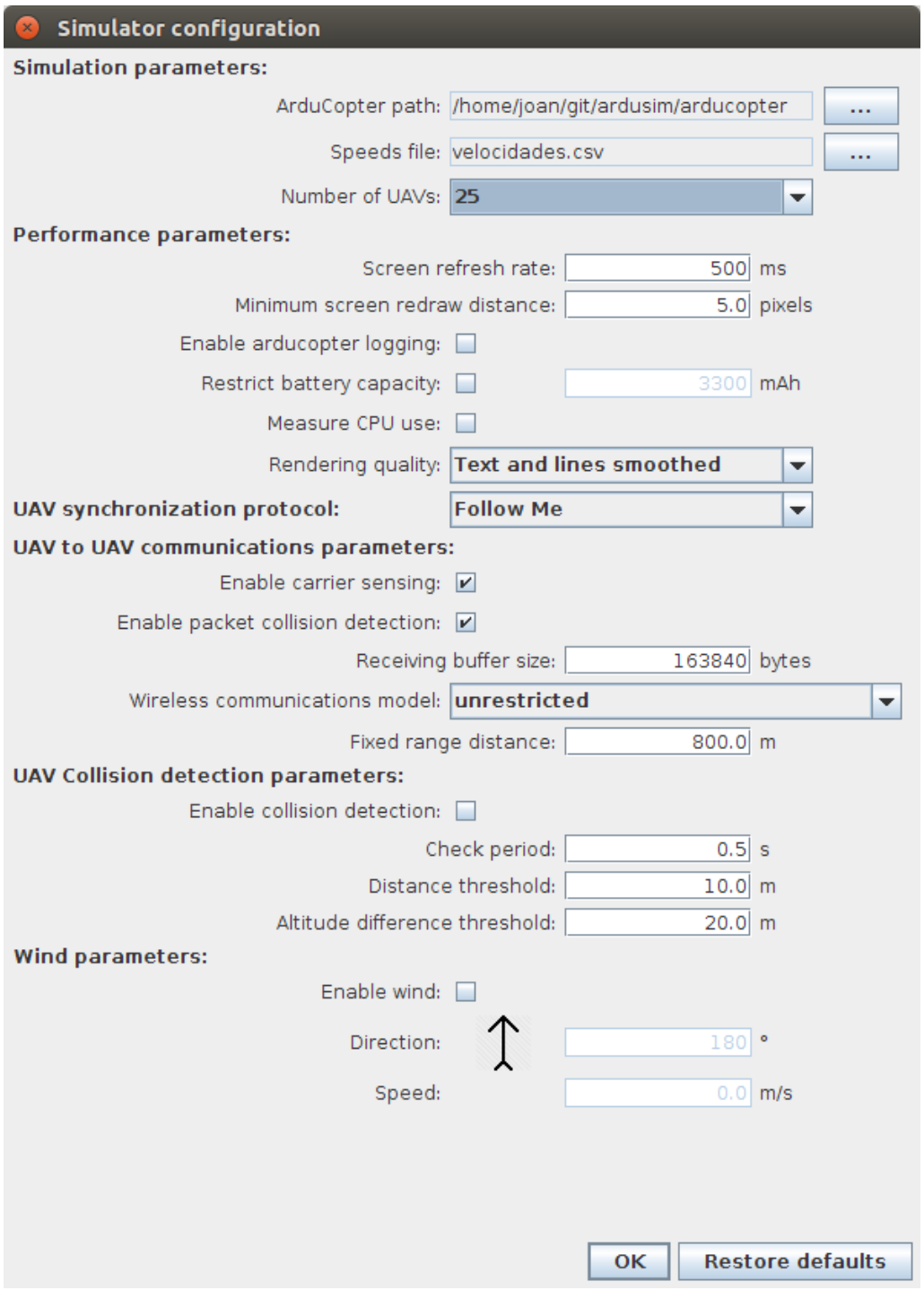

Figura 6.2: Ventana de configuración del simulador

Ventana principal

La ventana principal donde se muestra visualmente el simulador está dividida en tres apartados (Figura 6.3). En la parte superior izquierda se encuentra la información importante mostrada por el simulador, un registro donde se muestra el procedimiento que está siguiendo cada elemento del protocolo.

La división que se encuentra en la parte superior derecha trata del control del simulador. Existen cuatro botones para controlar el inicio y fin de la simulación. El botón *Setup* se utiliza para iniciar el estado del simulador *SETUP_IN_PROGRESS*. Una vez terminada esta etapa, el estado del simulador cambia a *READY_FOR_TEST*; cada protocolo utiliza estos estados como le es más conveniente.

El botón *Start test* se utiliza para actualizar el estado del simulador a *TEST_IN_PROGRESS*. Este estado es el que se encarga en la mayoría de protocolos de realizar la función principal del protocolo, ya sea realizar misiones, o realizar seguimientos en enjambre.

El botón *Exit*, como indica su nombre en inglés, se utiliza para cerrar el simulador. En desarrollo, es recomendable cerrar desde el botón, ya que también existen otro tipo de cierre.

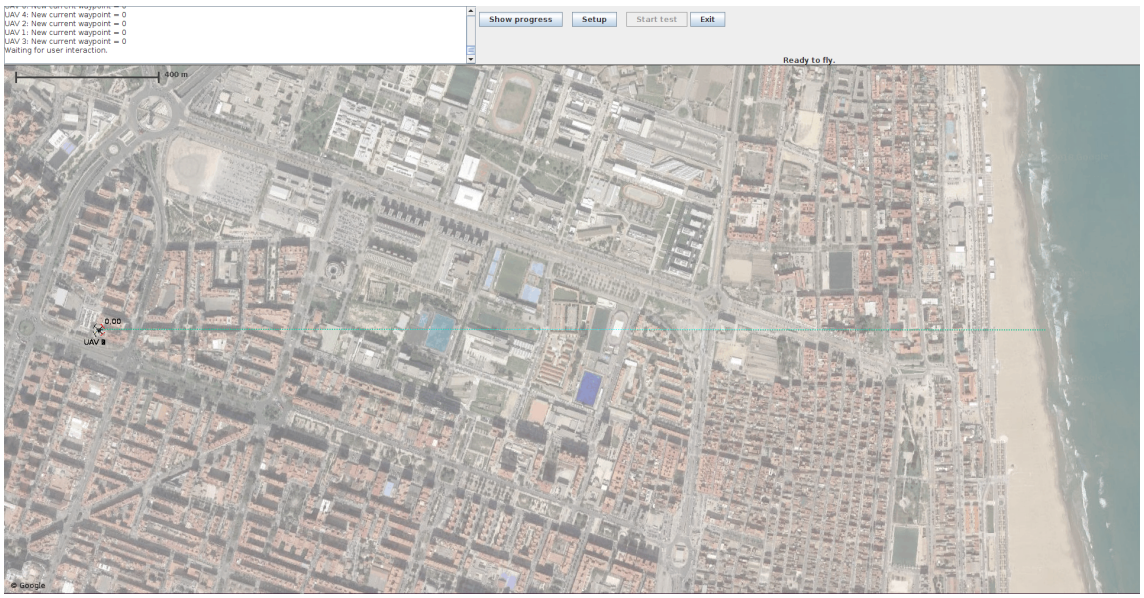

Figura 6.3: Ventana principal de ArduSim

Por último existe un botón el cual sirve para abrir de nuevo la ventana de progreso de los UAV, mencionado en la Figura 6.4. Este elemento contiene el tiempo trascurrido desde el inicio del experimento.

Para cada dron, en la parte superior, indica su identificador, las coordenadas X e Y en valores UTM, la altitud y la velocidad, que viene indicada en metros por segundo. En este apartado se verifica que nunca supera la velocidad máxima establecida por el archivo de velocidades. Por último, se incluye el modo de vuelo en el que se encuentra el UAV. En algunos protocolos, se puede observar un valor adicional que se establece por el estado del UAV definido para el protocolo en cuestión.

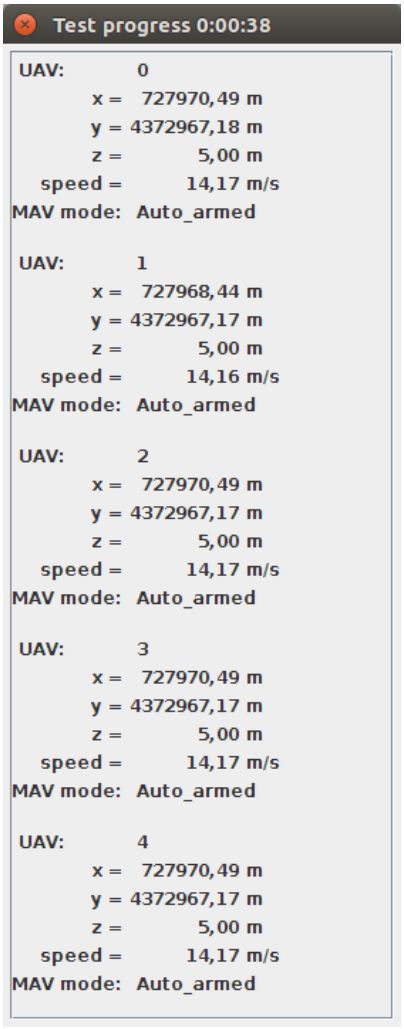

Figura 6.4: Ventana parámetros UAVs

6.2 Protocolos del simulador

El software de simulación ArduSim se encuentra en desarrollo. Actualmente existen varios protocolos en desarrollo en paralelo con el software ArduSim en sí. Algunos de los protocolos son trabajos de final de grado, incluido el desarrollado en este proyecto.

El protocolo de inicio, denominado *none*, es un protocolo que permite seleccionar archivos de misiones. Este archivo contiene una misión, que no es más que una lista de Waypoints, los cuales el UAV tiene que seguir. Como se puede observar, el protocolo es bastante simple y, durante el proyecto, se ha utilizado para observar el rendimiento del proceso de simulación, ya que se trata del protocolo más simple, y no comprende ningún cálculo extra. El número máximo de UAVs que permite este protocolo viene dado por el número máximo de UAVs que permite el simulador (256), condicionado también por las prestaciones del ordenador donde se está ejecutando.

El siguiente protocolo, denominado *MBCAP*, permite establecer una comunicación entre dos UAV que se interceptan durante su misión, para no colisionar. Se

realiza un intercambio de mensajes que permiten la coordinación entre los UAV, evitando choques entre ellos. La solución, ante el choque de dos UAV, consiste en desplazar un solo UAV horizontalmente, permitiendo el paso al otro UAV. El protocolo se ha ejecutado durante el proyecto a modo de prueba para observar comportamiento en el simulador.

El protocolo denominado *Swarm protocol* ha sido desarrollado para realizar enjambres de UAVs realizando misiones. Durante el proyecto actual no se ha ejecutado en ninguna ocasión. Se pretendía realizar funciones del protocolo, similares para el proyecto actual, pero la realidad es que no se pueden utilizar. Se trata de un protocolo completamente distinto al protocolo del proyecto, ya que utiliza misiones.

Por último existen varios protocolos actualmente en desarrollo por estudiantes realizando el trabajo final de grado. Por el momento se dispone de poca información sobre estos protocolos. Uno de ellos trata de realizar vuelos en búsqueda de fuentes de ozono, y está a punto de terminarse. Por otra parte, se pretende desarrollar un protocolo que permita el despegue y aterrizaje desde la cubierta de un barco, realizando vuelos con la finalidad de obtener información sobre la pesca. Estos protocolos tan solo se encuentran en fase de planificación, y de momento no se dispone de más información sobre ellos.

El software ArduSim, como se ha detallado anteriormente, permite realizar simulaciones de diferentes protocolos, de temáticas completamente diferentes. El plan de futuro del software simulador es que cualquiera pueda realizar la simulación de su propio protocolo, con la finalidad de exportar el protocolo a UAV reales. [24]

6.3 Estados del simulador

El simulador está implementado sobre una máquina de estados que permite definir cada proceso de carga o ejecución. Los desarrolladores de ArduSim han diseñado una máquina de estados como se define a continuación.

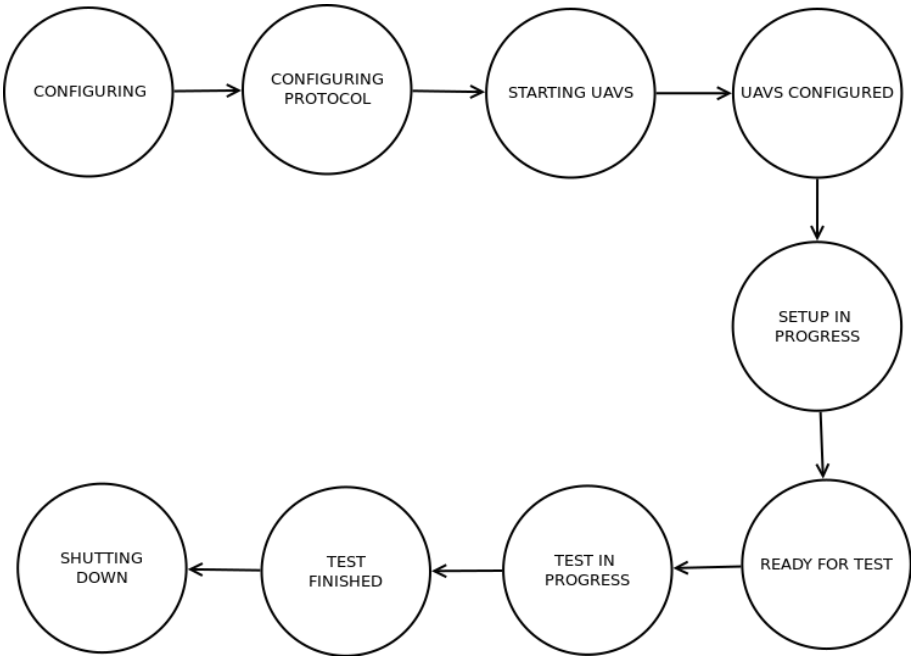

Figura 6.5: Maquina de estados de ArduSim

Configuring

El estado *Configuring* se establece como estado inicial, y es el estado en el que se configura el simulador y se realizan las cargas necesarias. Además, es el estado que se encarga de visualizar la ventana de configuración de ArduSim. Al terminar la configuración, el simulador cambia al estado siguiente.

Configuring protocol

El siguiente estado es *Configuring protocol*. Si el protocolo requiere alguna configuración especifica, el estado actual es donde se efectúa dicha configuración. Una vez realizados los ajustes del protocolo, el simulador cambia al siguiente estado.

Starting UAVs

Starting UAVs es el estado donde el simulador inicia las instancias *SITL* para cada uno de los UAVs que se han seleccionado, con un máximo de 256 UAVs por restricciones del simulador. También se establece el modo de vuelo inicial.

UAVs configured

Una vez termina la configuración de los UAVs, se para en este estado a la espera de la intervención de usuario.

Setup in progress

El estado *Setup in progress* es donde se realizan las preparaciones para iniciar la ejecución de la simulación.

Ready for test

Una vez termina la fase anterior se entra en el estado *Ready for test* a la espera de la intervención del usuario.

Test in progress

El estado *Test in progress* es el estado principal del simulador. La ejecución de la simulación se realiza en el estado actual, que incluye las misiones, seguimiento, realización de enjambres, etc. Cuando el usuario inicia el experimento el simulador inicia una cuenta del tiempo transcurrido desde que cambia al estado actual. Finalizada la ejecución del protocolo, se realiza un cambio al estado siguiente.

Test finished

Inmediatamente se muestra el resultado de la recopilación de los datos estadísticos obtenidos. El estado actual se mantiene hasta que el usuario presiona el botón de salida, cambiando al estado final.

Shutting down

Shutting down es el estado final, el cual termina la ejecución y se encarga de eliminar los ficheros temporales para la ejecución. Por último, finaliza el programa principal.

CAPÍTULO 7
Protocolo desarrollado

En este capítulo se detallan todos los procedimientos que se han realizado para desarrollar el protocolo principal de este proyecto. En primer lugar se realiza una explicación del funcionamiento del protocolo en simulación. En segundo punto se detalla la solución propuesta para el protocolo. En tercer lugar se menciona como se ha solucionado el problema de la inexistente emisora radiocontrol en el simulador. En cuarto lugar se detalla el diseño de los mensajes en la comunicación. El quinto punto indica la realización de las formaciones en el enjambre. Los dos puntos siguientes se realiza una explicación del contenido de cada uno de los roles del protocolo. En octavo lugar se incide en el funcionamiento del protocolo en la versión final. Por último se detallan las mejoras realizadas para la simulación del protocolo.

7.1 Protocolo FollowMe

El proyecto trata de establecer un enjambre de UAVs con una formación predefinida en vuelos en tiempo real. Para ello se necesita establecer un orden cronológico de las tareas previas a realizar dicha acción.

La secuencia de tareas que tiene que realizar cada UAV depende de su función específica en el enjambre. El dron **maestro** se encarga de realizar los movimientos indicados por la emisora de radiocontrol y enviar a todos los drones restantes su ubicación. Estos drones, los denominamos **esclavos**, reciben las coordenadas de donde se encuentra el dron maestro y calculan la posición que tienen que alcanzar para mantener una formación.

7.2 Implementaciones del protocolo

Durante el desarrollo se han planteado distintas configuraciones y diferentes implementaciones del protocolo, algunas han avanzado por sus ventajas y otras se han diseñado desde otro punto de vista. A continuación se detallan los tres principales planteamientos que se han realizado, y sus ventajas e inconvenientes, junto a los problemas que se han detectado.

7.2.1. Primera Versión

En primer lugar, se ha realizado un estudio sobre la implementación del protocolo de seguimiento en enjambre. Dicho estudio se basó en cómo realizar una secuencia de ejecuciones en el instante adecuado. Para ello se diseñó una secuencia de estados inicial, que consistía en el modelo de las Figuras 7.1 y la 7.2.

Se optó por una implementación de tres hilos por UAV. El primer hilo se encargaba del proceso central del UAV, de los cambios de estado, y del procesamiento de los mensajes y órdenes de actuar en cada momento. Se encargaba también de la creación del resto de hilos necesarios para las demás tareas. Uno de los hilos se encarga del envío de mensajes, y se denomina *Talker*. Cada vez que el hilo principal requería enviar un mensaje se creaba un nuevo hilo *Talker* que asumía esta función. El otro hilo, denominado *Listener*, se encargaba de realizar una escucha de mensajes, y cuando recibía el mensaje que tenía previsto recibir el hilo finalizaba.

Tanto para el dron maestro como para los esclavos, el funcionamiento de las comunicaciones se realiza de manera idéntica, tal y como se ha explicado en el párrafo anterior. Esta implementación imponía demasiada sobrecarga en cuanto a creación y finalización de hilos, lo cual resultaba en poca fiabilidad al realizar simulaciones con gran escalabilidad.

Suponiendo un ejemplo de ejecución de 25 UAVs, un dron adopta automáticamente el rol de maestro, y los 24 drones restantes asumen el rol de esclavos. Esta implementación implica los siguientes pasos:

1. Se crean 25 hilos, los cuales controlan los estados que cada UAV que tiene asociado. Se inicia el proceso de identificación, y todos los esclavos envían un mensaje por segundo al maestro. Este proceso implica crear 24 hilos, enviar 24 mensajes, y destruir 24 hilos cada segundo.

2. El maestro se encarga de ir almacenando los identificadores de los UAV que recibe. Estas operaciones se realizan hasta que el usuario presiona el botón de *Setup* y finaliza la fase de reconocimiento e identificación de los esclavos.

3. Todos los UAV actúan cambiando de un estado al siguiente. El maestro envía constantemente mensajes de despegue cada un segundo. Los esclavos, al recibir el primer mensaje de despegue, finalizan el hilo *Listener* y realizan el proceso de despegue.

4. Los esclavos, una vez alcanzada la altura deseada en el despegue, envían un mensaje al maestro, indicando que están listos. Este proceso implica crear un hilo por mensaje; tratándose del ejemplo de 25 UAVs, se generan 24 nuevos hilos. El maestro recibe los mensajes y se mantiene a la espera del usuario. El usuario inicia la ejecución al pulsar el botón *Start*.

5. El maestro inicia un nuevo hilo que envía, cada segundo, las coordenadas de donde se encuentra, a todos los esclavos. Cada uno de los esclavos crea un nuevo hilo para recibir todos los mensajes de coordenadas emitidos por el maestro.

6. Por último, al finalizar la traza, el maestro envía un último mensaje a los esclavos indicando que aterricen. Los esclavos, esta vez no inician un nuevo hilo *Listener*, simplemente se utiliza el último hilo creado para recibir coordenadas, y así procesar el mensaje de aterrizaje.

7. Al aterrizar, todos los UAV finalizan el protocolo.

Observaciones

La implementación de tres hilos por UAV no es del todo una desventaja; en ocasiones, sí hay que realizar gran cantidad de procesos de recepción, envío y procesado de datos. En general, la opción de repartir la carga de trabajo suele ser una de las mejores soluciones. No obstante, en el proyecto actual, no es el caso, ya que no existe una gran cantidad de mensajes, ni una gran cantidad de cálculos en el proceso central. Siendo así, la creación de tantos hilos puede resultar un inconveniente, ya que eleva el uso de CPU para procesos muy simples.

Como alternativa, se optó por realizar un nuevo planteamiento de la implementación como se presenta en la Subsección 7.2.2

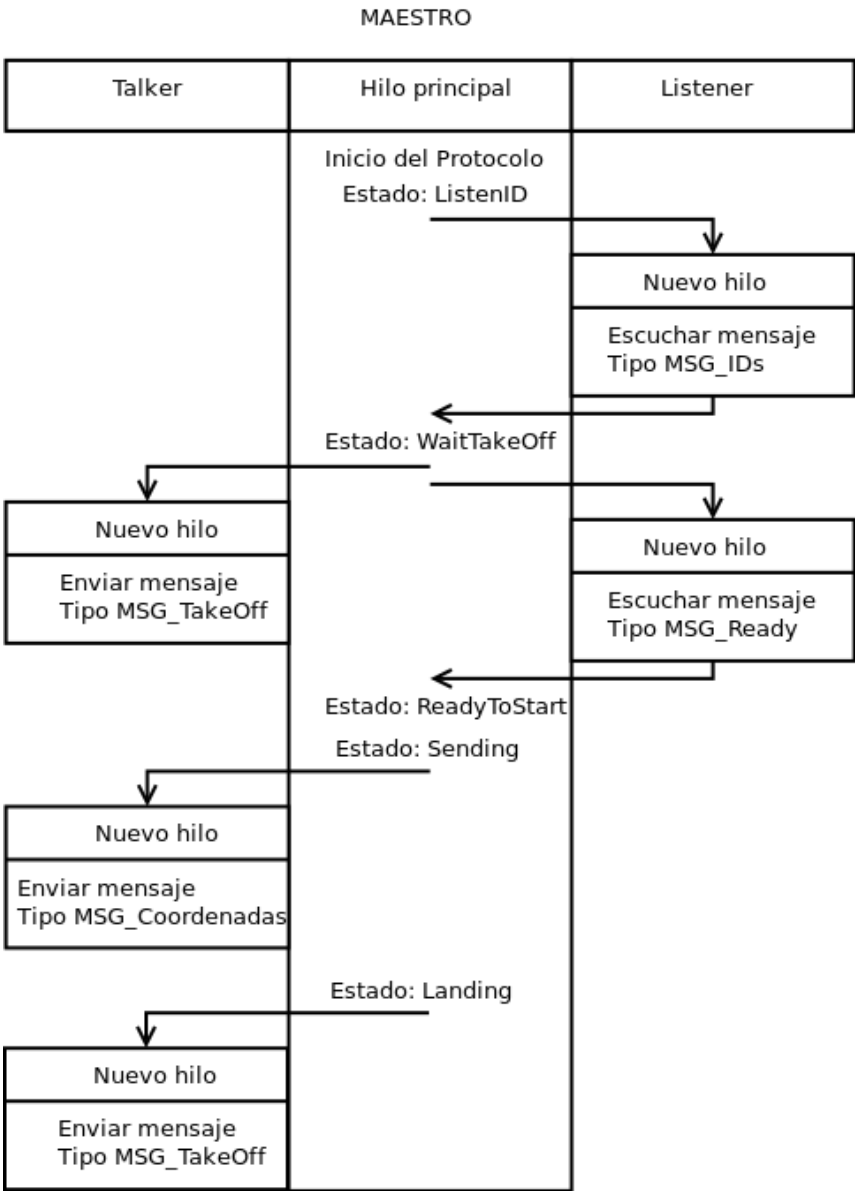

Figura 7.1: Esquema de ejecución Maestro V1

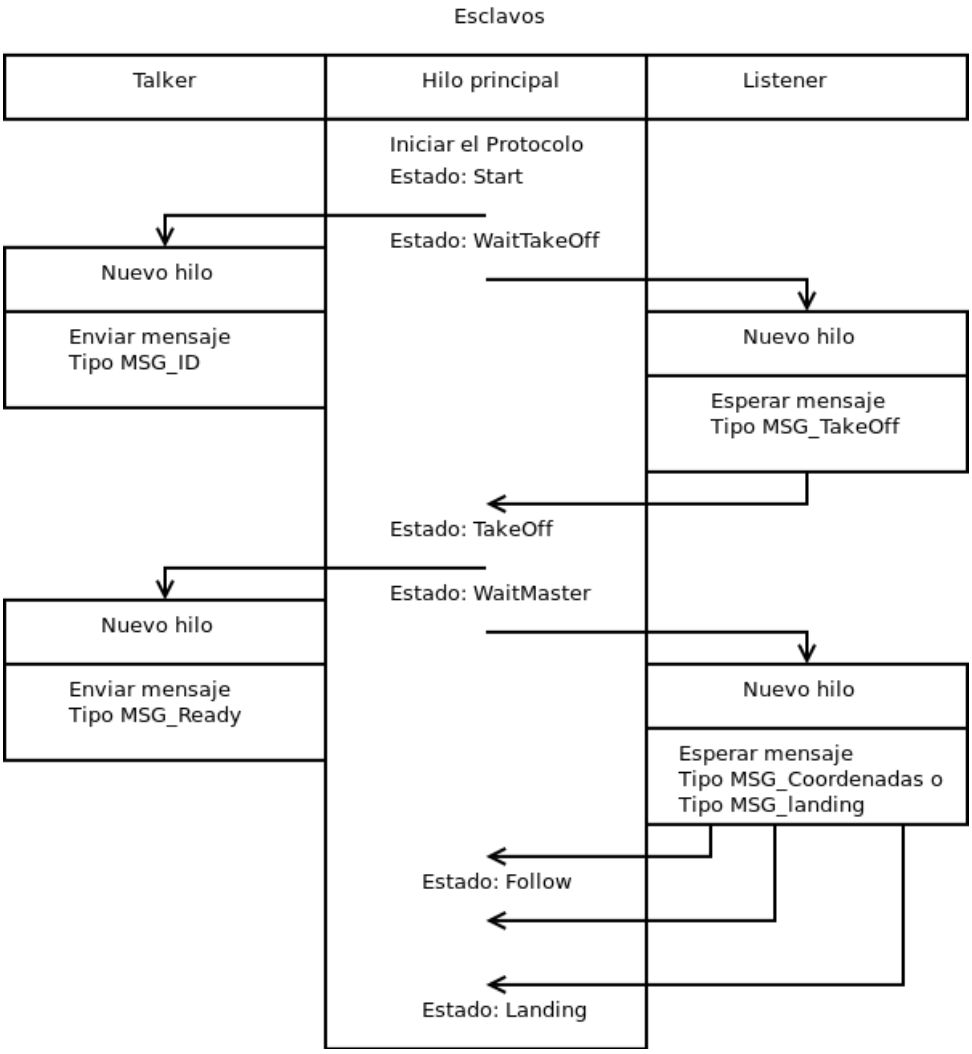

Figura 7.2: Esquema de ejecución Esclavo V1

7.2.2. Segunda Versión

La segunda versión se plantea desde un punto de vista diferente. El control de cada uno de los UAV está compartido entre dos hilos, el hilo de responsable por el envío de mensajes *Talker*, y el hilo responsable por la recepción de mensajes *Listener*. Solo uno de los dos se encarga de realizar el cambio de estado o un proceso en concreto, mientras el otro espera para no ocasionar condiciones contradictorias, típicas de la concurrencia.

En esta ocasión la secuencia de procesos es muy similar, y el cambio más circunstancial es el siguiente. Al inicio del protocolo se inician dos hilos por UAV. Estos dos hilos son los mencionados anteriormente. La particularidad que presentan es que, una vez iniciados, no finalizan hasta que termina por completo el protocolo. Hay momentos en que no es necesario que se mantengan realizando ejecuciones ni cálculos. En estos casos se mantienen a la espera durante períodos cortos, alrededor de medio segundo.

En la Figura 7.3 se puede apreciar los espacios en blanco que contiene el hilo *Talker* desde el inicio hasta la espera del botón *Setup*, entre el envío del mensaje de despegue y la activación del botón *Start*. El hilo *Listener* también se aprecian

huecos a la espera desde el envió del mensaje de Ids hasta el cambio de estado por un instante de tiempo, mientras el otro hilo envía mensajes. A continuación se observa otro espacio donde se mantiene al la espera hasta el mensaje que indica *Ready*. Estos espacios son las esperas que realiza cada hilo cuando no tiene tareas que realizar.

En la Figura 7.4 se puede apreciar que los espacios que existen entre mensajes en el caso del *Talker* son los mismos. En el caso del *Listener*, existe un tiempo de espera entre el mensaje enviado con el identificador, hasta que se presiona el botón *Setup*, y también se incluye una espera justo en el estado de despegue.

Observaciones

Se considera que la implementación con dos hilos para cada uno de los UAV es adecuada, a pesar de que persisten ciertos inconvenientes. De hecho, los mensajes elevan el uso de CPU, algo que en su momento ya se investigó.

Las causas del uso de CPU elevado solo se podía observar al realizar ejecuciones con más de 8 UAVs. Al final se llegó a la conclusión que los mensajes se enviaban correctamente, y que el UAV que los tenía que recibir lo hacía correctamente. Pero el problema se encontraba en los UAV que no tenían que recibir el mensaje en el momento adecuado.

La solución al problema de no escuchar cuando es debido, se encuentra en la Subsección 7.2.3.

Figura 7.3: Esquema de ejecución Maestro V2

Figura 7.4: Esquema de ejecución Esclavo V2

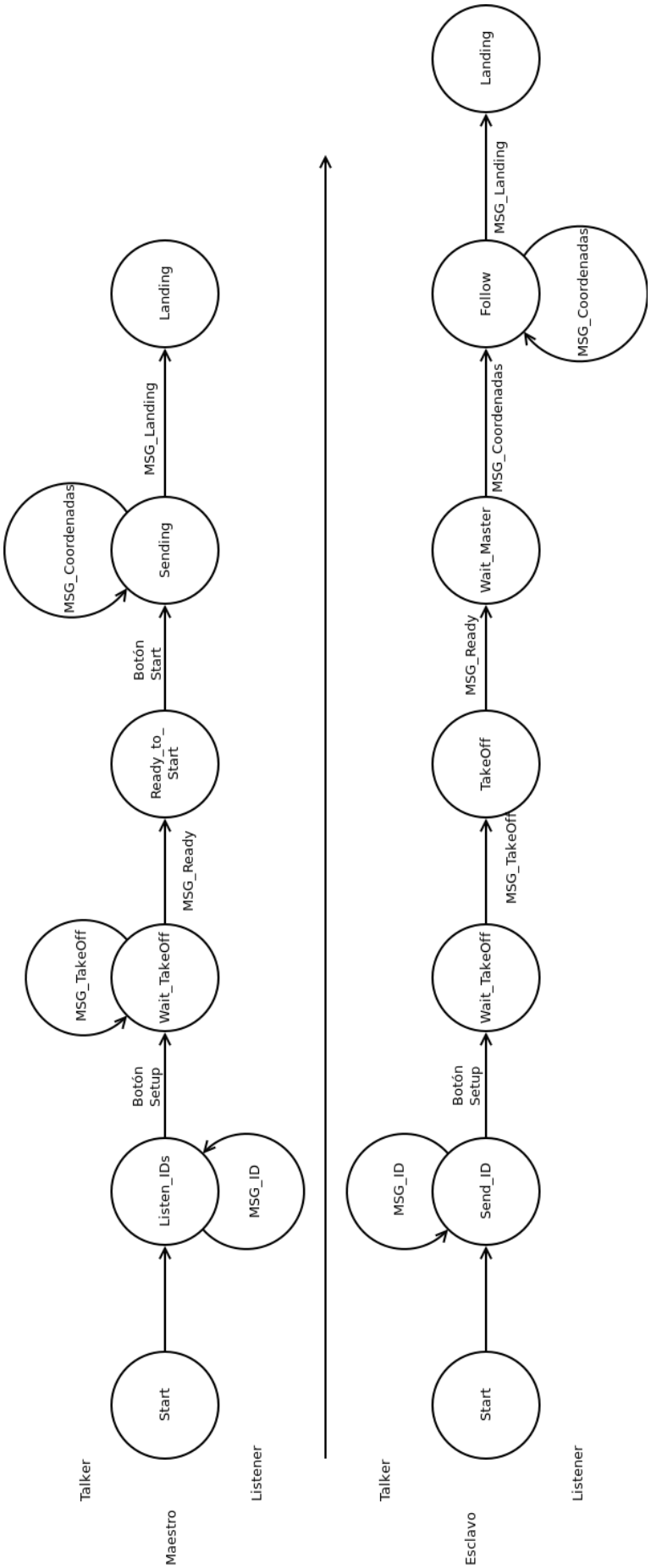

Figura 7.5: Maquina de estados V2

7.2.3. Tercera Versión

La versión definitiva para finalizar el proyecto es la implementada en la tercera versión. Las versiones anteriores tenían un elevado uso de CPU, lo cual era un efecto indeseable. A simple vista no se observaba un excesivo procesamiento asociado a los cálculos. Los mensajes se enviaban cuando se tenía que realizar esta acción. Los UAV que recibían en cada momento lo hacían correctamente.

Se observó que, con un número inferior a 8 UAVs, el sistema funcionaba perfectamente. En el momento que la cantidad de multicópteros ascendía, la gran mayoría seguía funcionando bien, pero entre 1 y 3 UAVs se quedaban rezagados.

Primero se supuso que sería cuestión de la CPU, pero, observando las características del equipo, como se indica en el Apéndice A, se comprobó que éste es suficientemente potente para realizar la simulación, por lo cual esta suposición se descartó.

La segunda suposición fue que los mensajes no se recibían bien. Se rastrearon las comunicaciones, y se observaron las escuchas en las máquinas de estados, hasta encontrar el problema.

El problema que ocurría se encontraba en los hilos *Listener*. Mientras no tenían que recibir mensajes para procesarlos, no escuchaban ningún tipo de mensaje. Esta acción daba como consecuencia un llenado masivo del *buffer* de recepción.

El *buffer* de recepción quedaba completamente lleno si se tenía una gran cantidad de UAVs. Lo que ocurre en el instante que el hilo *Listener* necesita recibir un mensaje para procesarlo, es que tiene que analizar todos los mensajes recibidos y almacenados en el *buffer* de recepción, al tratarse de comunicaciones simuladas. Este proceso, con pocos UAVs y pocos mensajes, es inapreciable. De lo contrario, con gran cantidad de UAVs, el proceso crece exponencialmente, dando lugar al colapso de la CPU por unos instantes.

La solución que se ha optado por desarrollar en esta tercera versión es la siguiente.

El hilo *Listener* nunca puede dejar de escuchar. Si no interesa procesar los mensajes, se descarta automáticamente, dando paso a poder leer el siguiente mensaje. La solución se aplicó en la maquina de estados del maestro, como se puede observar en la Figura 7.6. Lo más importante fue modificar la máquina de estados de los esclavos, ya que se trata de multicópteros replicados, y los cambios correspondiente se pueden observar en la Figura 7.7. El conjunto de cambios también queda reflejado en la Figura 7.8.

Observaciones

El cambio de planteamiento y la implementación de la solución se vio reflejada considerablemente en las ejecuciones. Realizando el mismo proceso de simulación, con la mismas cantidades de UAVs, el uso de CPU quedó reducido a una sexta parte respecto a las versiones anteriores.

Figura 7.6: Esquema de ejecución Maestro V3

Figura 7.7: Esquema de ejecución Esclavo V3

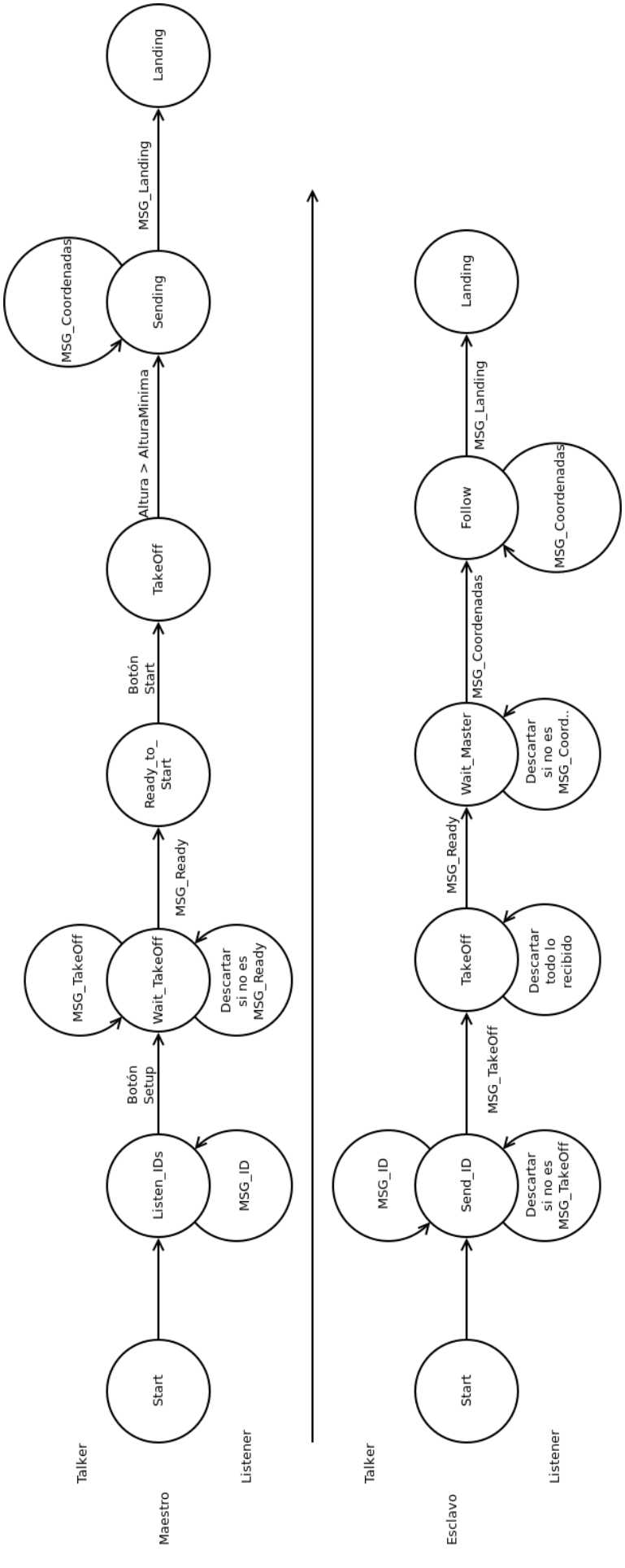

Figura 7.8: Maquina de estados V3

7.3 Traza de vuelo real

Un vuelo real conlleva el uso de una emisora radiocontrol y un piloto, a cargo del UAV. El simulador no dispone de la funcionalidad de conectar una emisora radiocontrol para simular un vuelo real en tiempo real.

Para solucionar este impedimento, se ha implementado un hilo que envíe al dron maestro una traza idéntica a la de una emisora radiocontrol. Para ello se necesita un archivo que contenga la traza real de un vuelo, tal y como se ha detallado en el Capítulo 5, para el cual se desarrolló un programa específico que realizará este guardado automático al iniciar un vuelo.

El hilo denominado *MasterMando* tiene la función de realizar una lectura del archivo y enviar los valores del mando al dron en el instante adecuado. En el estado del simulador *Configuring protocol* se selecciona el archivo que contiene la traza. El archivo es analizado y se reordenan los registros si es necesario, estableciendo un orden temporal. A continuación aplica un preproceso que consiste en restar a todos los valores del tiempo el tiempo de la primera traza, es decir, el tiempo inicial. De esta manera el tiempo inicial queda a cero.

Los procesos que se aplican más tarde quedan reflejados en la Subsección 7.6.1.

7.4 Comunicación

Las comunicaciones son esenciales en el protocolo desarrollado. Por ello se han diseñado distintos tipos de mensajes, para el intercambio de información o para indicar que se debe adoptar un estado nuevo según el protocolo. A continuación se indican los contenidos del mensaje y su utilización. En la Figura 7.9 se representa un esquema del envío y recepción de los mensajes. Se indica con tres o más flechas el envío múltiple de mensajes del tipo indicado en una dirección. Se representa con una sola flecha el envío de un solo mensaje del tipo indicado.

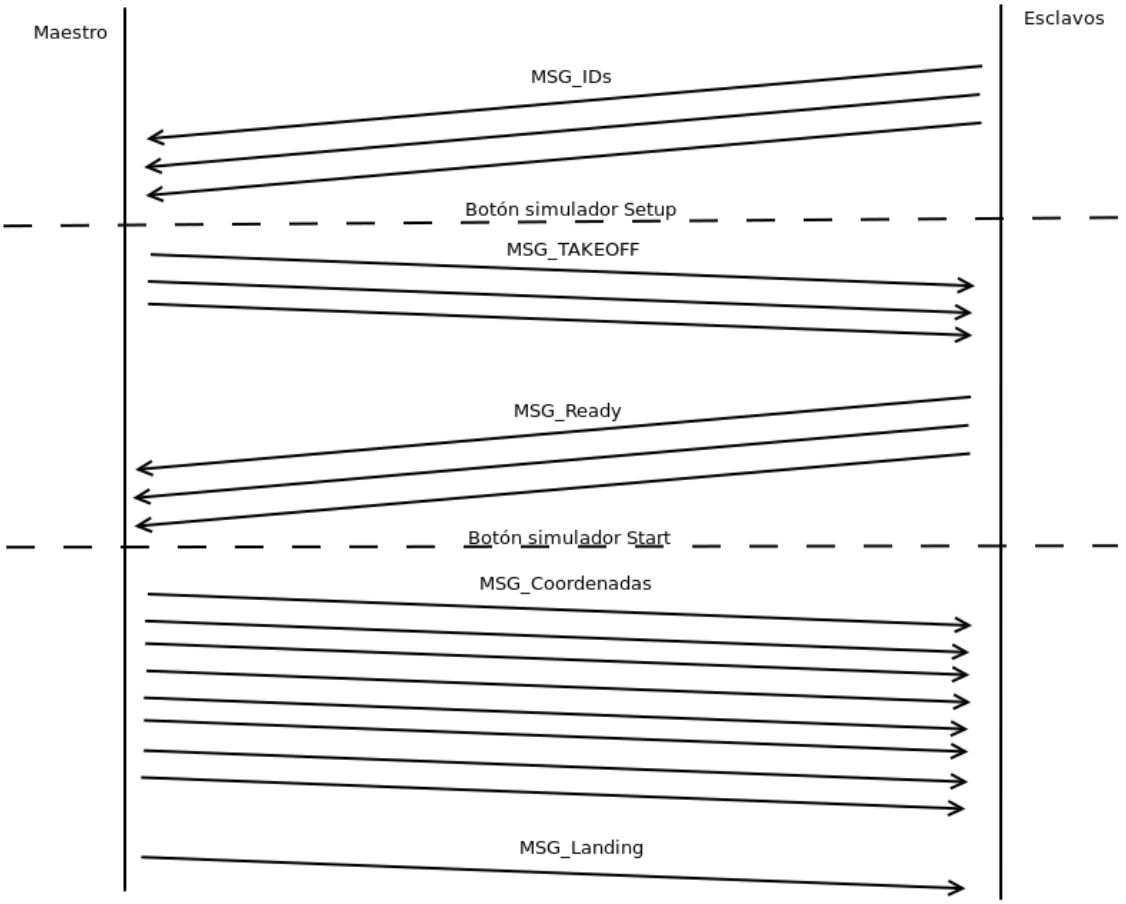

Figura 7.9: Esquema de comunicaciones

7.4.1. MSG_IDs

El mensaje de tipo *MSG_IDs* es siempre enviado desde un dron esclavo, y es recibido y procesado por un dron maestro. Se utiliza para reconocer el identificador de un dron esclavo. El maestro almacena los identificadores de cada dron esclavo una sola vez. Es la manera que tienen de comunicar que existen y están preparados para ser esclavos. Si un dron esclavo no envía este tipo de mensaje, el dron maestro no sabe que existe, y nunca más lo esperará durante la ejecución del protocolo.

El contenido del mensaje consiste en dos parámetros de tipo int, donde el primero es el identificador del dron, y el segundo es el tipo de mensaje específico.

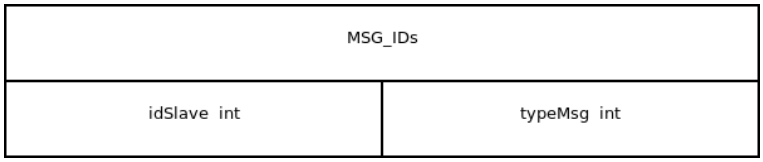

Figura 7.10: Esquema MSG_IDs

7.4.2. MSG_TakeOff

El mensaje de tipo *MSG_TakeOff* es siempre enviado por el dron maestro, y lo reciben todos los drones esclavos, que lo procesan una sola vez. El dron maestro envía este tipo de mensaje cuando el simulador entra en el estado *Setup in progress*. Los drones esclavos reciben el mensaje y se activa el despegue automático a la altura indicada.

El mensaje contiene información adicional que se utiliza para indicar su posición dentro de la formación.

La estructura del mensaje está comprendida por los siguientes parámetros: identificador del dron master de tipo int, identificador del tipo de mensaje de tipo int, tamaño del array de posiciones de tipo int, y un array de posiciones de tamaño igual al número de esclavos que se ejecuta en la simulación, también de tipo int. El tamaño del mensaje se ve afectado por la cantidad de drones que existen.

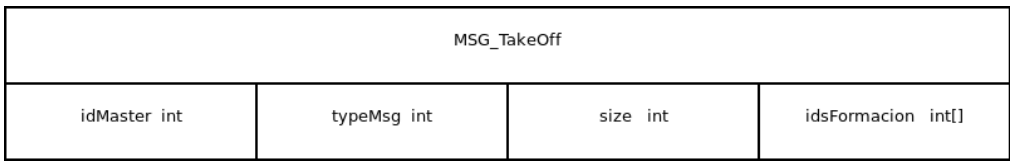

Figura 7.11: Esquema MSG_TakeOff

La versión inicial del diseño de este tipo de mensaje contempla la opción de la Subsección 7.5.1. Existe una segunda versión del mensaje que amplia la estructura del mismo, permitiendo realizar cálculos para crear un *offset* lo cual se explica en la Subsección 7.5.2.

Los datos que se añaden a los existentes son la latitud, almacenada en una variable de tipo *double*, y la longitud, siempre unida al parámetro anterior y con las mismas características. Por último el *heading*, el cual contiene el valor del rumbo del dron maestro en radianes.

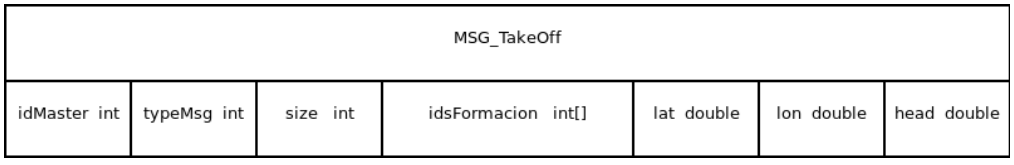

Figura 7.12: Esquema MSG_TakeOff versión 2

7.4.3. MSG_Ready

El mensaje de tipo *MSG_Ready* se envía desde todos los drones esclavos, y los recibe el maestro. Estos mensajes se utilizan para garantizar que cada uno de los esclavos alcanza la altura adecuada y su posición.

El contenido del mensaje es el siguiente: un primer parámetro, que indica la procedencia del mensaje, en este caso el identificador del esclavo; y un segundo parámetro, que indica el tipo de mensaje. Con esta combinación de identificador y tipo *MSG_Ready*, el maestro reconoce que el esclavo indicado está listo para realizar el seguimiento.

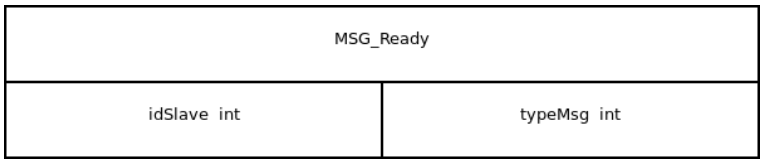

Figura 7.13: Esquema MSG_Ready

7.4.4. MSG_Coordenadas

El mensaje de tipo *MSG_Coordenadas* se envía desde el maestro, y lo recibe cada uno de los esclavos. La función principal del mensaje es comunicar dónde se encuentra el maestro. La primera vez que se recibe este tipo de mensajes en los drones esclavos, se efectúa un cambio de estado, pasando de *Wait_Master* a *Follow*. En el estado *Follow* se continúa recibiendo este tipo de mensaje, además del mensaje que se explica en el apartado siguiente.

La estructura del mensaje se ha diseñado de la siguiente manera. El primer parámetro, como es habitual, consiste en el identificador del UAV de procedencia; en este caso siempre es el maestro. El segundo parámetro es del tipo de mensaje *MSG_Coordenadas*. El tercer parámetro es la latitud en coordenadas geodésicas, y su valor se almacena en un *double*. El cuarto parámetro, emparejado con en anterior, consiste de la longitud con las mismas características. El quinto parámetro es el rumbo o *heading*, almacenado en un *double* y expresado en radianes. El siguiente parámetro es la altura, y es almacenado en un *double*. Los tres últimos parámetros son las velocidades en los ejes X, Y y Z, y se almacenan cada uno de ellos en un *double* según el orden indicado.

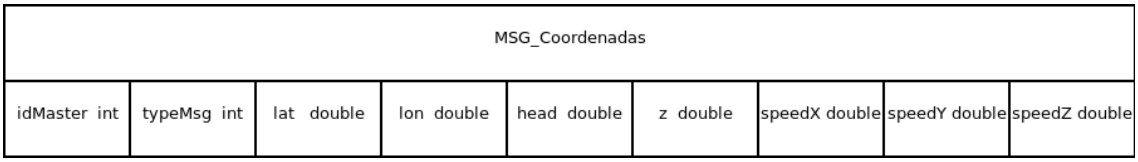

Figura 7.14: Esquema MSG_Coordenadas

7.4.5. MSG_Landing

El mensaje de tipo *MSG_Landing* se envía desde el maestro, y lo reciben todos los esclavos. El mensaje se envía cuando la traza de la emisora radiocontrol simu-

lada finaliza. Llegados a este punto, el maestro no tiene más órdenes que realizar y empieza su aterrizaje enviando la orden a todos los esclavos para que actúen de forma idéntica. La función que tiene en los esclavos es de no realizar más acciones y aterrizar. El mensaje se puede recibir en cualquier instante, mientras el *Listener* de los esclavos espera un mensaje de tipo *MSG_Coordenadas*.

Los datos que contiene el mensaje son los siguiente: el primer parámetro es un identificador del maestro de tipo int, y el segundo parámetro es el tipo de mensaje de tipo int.

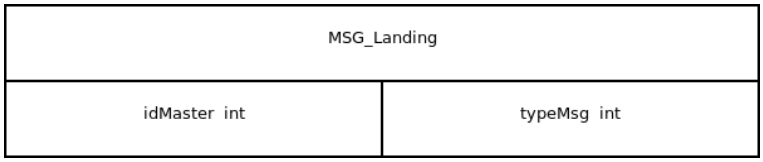

Figura 7.15: Esquema MSG_Landing

7.5 Formaciones de enjambres

El objetivo del proyecto trata de realizar vuelos en formaciones de enjambres. Los enjambres, con una estructura bien realizada, requieren del cálculo de esta estructura. Por ello se ha decidido implementar algunas funciones que permiten realizar distintas formaciones.

- Linea
- Matriz
- Círculo

7.5.1. Formación versión 1

En primer lugar se diseñó una implementación que realizaba los cálculos de cada posición en el instante de tiempo que se recibía el mensaje de tipo *MSG_Coordenadas*. Este cálculo es innecesario, ya que cada vez que se recibía un mensaje se recalculaban todos las operaciones necesarias para obtener las coordenadas, por lo cual se optó por otra implementación.

7.5.2. Formación versión 2

La segunda implementación se ha desarrollado de manera distinta. Existen cálculos que solo es necesario realizarlos una sola vez; estos cálculos se realizan al principio de la ejecución. Más tarde, en cada recepción de las coordenadas del maestro, se aplica el *offset* obtenido, obteniendo el mismo resultado que en la versión anterior.

7.6 Simulación de Maestro

La cantidad de UAVs simulados puede establecerse como un valor dentro del rango [0-255]; en el protocolo actual, el mínimo de UAVs se ve incrementado a dos. El primer UAV es siempre el maestro, y el resto adopta el rol de esclavos. En el apartado actual se trata las acciones y procesos que realiza el dron maestro. Las comunicaciones entre distintos UAVs también es muy importante, y se trata en las Subsección 7.2.1, 7.2.2 y 7.2.3.

El maestro está compuesto por dos hilos principales, y otro adicional. Los hilos *Talker* y *Listener*, como se ha mencionado en apartados anteriores, realizan internamente todas las acciones sobre el UAV maestro. Adicionalmente, en vuelo real, existe una emisora radiocontrol, la cual en simulación no es posible. Se ha realizado un hilo aparte que simula una emisora radiocontrol.

Todos los hilos que realizan cambios en el UAV maestro, como precedencia a cualquier proceso y código en común, requieren establecer una espera hasta que el Simulador se encuentre en el estado adecuado para realizar las configuraciones necesarias.

```
1         while (!Tools.areUAVsReadyForSetup()) {
2       Tools.waiting(100);
3     }
```

7.6.1. Simulación de la emisora RC

Siguiendo la línea de la Sección 7.3, el hilo de simulación de una emisora radiocontrol sigue una serie de pasos que se tratan a continuación.

En tiempo de ejecución se calcula el tiempo que transcurre desde un registro al siguiente. El cálculo del tiempo se utiliza para realizar la espera adecuada en el envió de información. Para transmitir los valores de la traza se utiliza la función mencionada en la Subsección 6.1.3, *Copter.channelsOverride()*. Se trata de una función que necesita una serie de parámetros, los cuales se encuentran en la traza.

```
1     recurso = FollowMeParam.recurso.get();
2     long tfin = 0;
3     long tSysIni = System.nanoTime();
4     Nodo n = recurso.pop(type);
5
6     do {
7       Copter.channelsOverride(FollowMeParam.posMaster, n.chan1, n.chan2
            , n.chan3, n.chan4);
8       n = recurso.pop(type);
9       long espera = (n.time - (System.nanoTime() - tSysIni)) / 1000000;
10       if (espera > 0) {
11         Tools.waiting((int)espera);
12       }
13       if (n.next == null ) {
14         tfin = n.time;
15         Copter.setFlightMode(FollowMeParam.posMaster, FlightMode.
               LAND_ARMED);
16         FollowMeParam.uavs[FollowMeParam.posMaster] = FollowMeState.
               LANDING_MASTER;
```

```
17          GUI.updateprotocolState(FollowMeParam.posMaster, FollowMeParam.
                uavs[FollowMeParam.posMaster].getName());
18          break;
19        }
20      } while (n != null || n.next != null);
```

El cálculo de la variable espera es la diferencia de tiempo que existe; hay que realizar una espera por parte del hilo que envió de las trazas. Como se puede observar, al finalizar, el contenido de la traza en bucle finaliza, de manera que indica al dron maestro que debe aterrizar.

7.6.2. Listener del maestro

El hilo *Listener* realiza el cambio de estado en el maestro, estableciendo el estado *LISTEN_ID*.

El estado se utiliza para recibir todos los identificadores de los esclavos. El proceso que sigue es el siguiente, mientras no se cambie al estado *Setup* del simulador, éste espera mensajes tipo *MSG_ID*.

Los identificadores de los esclavos se almacenan en una variable tipo *ConcurrentHashMap*[25], por lo que es importante la utilización del tipo de datos correcto, ya que no acepta inserción de duplicados y se trata de un tipo de dato concurrente, ya que se escribe y lee desde distintos hilos.

A continuación realiza una espera corta, mientras se encuentra en el estado *WAIT_TAKE_OFF_MASTER*.

Durante el estado *WAIT_TAKE_OFF_MASTER* escucha mensajes del tipo *MSG_Ready* con el identificador de los esclavos, esperando a que se encuentren todos listos.

Por último realiza un cambio de estado al estado *READY_TO_START*.

```
1     FollowMeParam.uavs[idMaster] = FollowMeState.LISTEN_ID;
2     GUI.updateprotocolState(idMaster, FollowMeParam.uavs[idMaster].
          getName());
3
4     Input in = new Input();
5     byte[] message;
6     int idSender, typeMsg;
7
8     while (Tools.areUAVsReadyForSetup()) {
9       message = Copter.receiveMessage(idMaster); /// Espera bloqueante
10      in.setBuffer(message);
11      idSender = in.readInt();
12      typeMsg = in.readInt();
13      if (typeMsg == FollowMeParam.MsgIDs) {// Msg esperado
14        FollowMeParam.posformacion.put(idSender, idSender);
15        String m = "[";
16        for (int i : FollowMeParam.posformacion.values()) {
17          m += "" + i + ",";
18        }
19        GUI.log("IDs detectados: " + m + "]");
20      }
21
22    }
23
```

```
24      while (FollowMeParam.uavs[idMaster] != FollowMeState.
            WAIT_TAKE_OFF_MASTER) {
25        Tools.waiting(100);
26      }
27
28      int totUAVs = FollowMeParam.posformacion.size();
29
30      while (slaveReady.size() != totUAVs) {
31        message = Copter.receiveMessage(FollowMeParam.posMaster);
32        in.setBuffer(message);
33        idSender = in.readInt();
34        typeMsg = in.readInt();
35
36        if (typeMsg == FollowMeParam.MsgReady) {
37          slaveReady.put(idSender, idSender);
38
39          String msg = "[";
40          for (int i : slaveReady.values()) {
41            msg += "" + i + ",";
42          }
43          GUI.log("MasterListener Recibe Ready:" + msg + "]");
44        }
45      }
46
47      FollowMeParam.setupFinished = true;
48      FollowMeParam.uavs[FollowMeParam.posMaster] = FollowMeState.
            READY_TO_START;
49      GUI.updateprotocolState(FollowMeParam.posMaster, FollowMeParam.uavs
            [FollowMeParam.posMaster].getName());
50
51      GUI.log("MasterListener Finaliza");
```

7.6.3. Talker del maestro

Se realiza una espera mientras el estado del simulador no se encuentre en el estado *Setup*. Garantiza que no se avanza la ejecución hasta que no se presiona el botón de control.

Una vez presionado el botón, el maestro realiza un cambio de estado al estado *WAIT_TAKE_OFF_MASTER*.

A continuación, realiza una serie de pasos para obtener todos los identificadores de los esclavos, y construir un array indicando la posición que debe adoptar cada uno de los esclavos en la formación. El array se prepara para más tarde enviarlo a todos los esclavos, y éstos realizarán una búsqueda de su identificador.

Mientras el maestro se encuentre en el estado *WAIT_TAKE_OFF_MASTER* realizará envíos del mensaje de tipo *MSG_TAKEOFF*, cada medio segundo.

Más tarde, espera a que el estado del simulador cambie a *ExperimentInProgress*, lo cual significa que empieza el experimento, realizando un cambio de estado a *TAKE_OFF*.

Al realizar el despegue tiene que alcanzar una altura adecuada para poder empezar a realizar el cambio de estado a *SENDING*. Este estado permite realizar envíos de mensajes tipo *MSG_Coordenadas* 7.4.4, cada segundo.

Si el maestro cambia de modo de vuelo a los modos *LAND_ARMED* o *LAND*, el proceso de envío de coordenadas finaliza, dando lugar al aterrizaje. El aterrizaje envía indefinidamente el mensaje de tipo *MSG_Landing* 7.4.5, dando la orden a los esclavos para que aterricen.

```
1   GUI.log("MasterTalker run");
2
3   while (!Tools.isSetupInProgress()) {
4     Tools.waiting(100);
5   }
6
7   FollowMeParam.uavs[idMaster] = FollowMeState.WAIT_TAKE_OFF_MASTER;
8   GUI.updateprotocolState(idMaster, FollowMeParam.uavs[idMaster].
        getName());
9
10  for (int p : FollowMeParam.posformacion.values()) {
11    idsformacion[i++] = p;
12  }
13
14  while (FollowMeParam.uavs[idMaster] == FollowMeState.
        WAIT_TAKE_OFF_MASTER) {
15    buffer = new byte[Tools.DATAGRAM_MAX_LENGTH];
16    out.setBuffer(buffer);
17    out.writeInt(idMaster);
18    out.writeInt(FollowMeParam.MsgTakeOff);
19    out.writeInt(idsformacion.length);
20    out.writeInts(idsformacion);
21    out.flush();
22    byte[] message = Arrays.copyOf(buffer, out.position());
23    Copter.sendBroadcastMessage(idMaster, message);
24    out.clear();
25    Tools.waiting(500);
26  }
27  while (!Tools.isExperimentInProgress()) {
28    Tools.waiting(200);
29  }
30  FollowMeParam.uavs[FollowMeParam.posMaster] = FollowMeState.
        TAKE_OFF;
31  GUI.updateprotocolState(FollowMeParam.posMaster, FollowMeParam.uavs
        [FollowMeParam.posMaster].getName());
32
33  while (Copter.getZRelative(FollowMeParam.posMaster) < FollowMeParam
        .AlturaInitSend) {
34    Tools.waiting(200);
35  }
36
37  FollowMeParam.uavs[FollowMeParam.posMaster] = FollowMeState.SENDING
        ;
38  GUI.updateprotocolState(FollowMeParam.posMaster, FollowMeParam.uavs
        [FollowMeParam.posMaster].getName());
39
40  while (Copter.getFlightMode(idMaster) != FlightMode.LAND_ARMED
41      && Copter.getFlightMode(idMaster) != FlightMode.LAND) {
42    buffer = new byte[Tools.DATAGRAM_MAX_LENGTH];
43    out.setBuffer(buffer);;
44    out.writeInt(idMaster);
45    out.writeInt(FollowMeParam.MsgCoordenadas);
46    double lat, lon, heading, z, speedX, speedY, speedZ;
```

```
47        Point2D.Double utm = Copter.getUTMLocation(FollowMeParam.
              posMaster);
48        heading = Copter.getHeading(FollowMeParam.posMaster);
49        z = Copter.getZRelative(FollowMeParam.posMaster);
50        Triplet<Double, Double, Double> speed = Copter.getSpeeds(
              FollowMeParam.posMaster);
51        speedX = speed.getValue0();
52        speedY = speed.getValue1();
53        speedZ = speed.getValue2();
54        out.writeDouble(utm.x);
55        out.writeDouble(utm.y);
56        out.writeDouble(heading);
57        out.writeDouble(z);
58        out.writeDouble(speedX);
59        out.writeDouble(speedY);
60        out.writeDouble(speedZ);
61        out.flush();
62        byte[] message = Arrays.copyOf(buffer, out.position());
63        Copter.sendBroadcastMessage(idMaster, message);
64        out.clear();
65        Tools.waiting(1000);
66      }
67
68      while (true) {
69        buffer = new byte[Tools.DATAGRAM_MAX_LENGTH];
70        out.setBuffer(buffer);
71        out.writeInt(idMaster);
72        out.writeInt(FollowMeParam.MsgLanding);
73        msg = "Aterrizad todos";
74        out.flush();
75        byte[] message = Arrays.copyOf(buffer, out.position());
76        Copter.sendBroadcastMessage(idMaster, message);
77        out.clear();
78        Tools.waiting(1000);
79      }
```

7.7 Simulación de Esclavo

Al igual que ocurre con en el UAV maestro, y como se indica en la Sección 7.6, los hilos que realizan cambios en la configuración de los UAV esclavos contienen un código común antes de realizar cualquier tarea:

```
1         while (!Tools.areUAVsReadyForSetup()) {
2             Tools.waiting(100);
3         }
```

7.7.1. Listener del esclavo

El hilo que cambia el estado del protocolo en los esclavos es el *Listener*. El primer cambio de estado se realiza al estado *SEND_ID*. Durante este estado, los esclavos tienen la función de enviar mensajes del tipo *MSG_ID* (Subsección 7.4.1). Además, tienen que realizar la función de lectura de mensajes y su descarte (si

procede). De esta forma, el *buffer* de entrada de mensajes no quedará colapsado con demasiados mensajes, tal y como se detalla en la Subsección 7.2.3.

Los esclavos esperan recibir un mensaje del tipo *MSG_TAKEOFF*. Este tipo de mensaje contiene los datos adecuados para realizar los siguientes procesos, como se ha explicado en la Subsección 7.4.2.

Los datos que se extraen para realizar un orden en la formación son necesarios para crear el *offset*, y todas las formaciones tienen un *offset* precalculado en el tiempo que se recibe el mensaje con la información de las posiciones.

Para obtener la posición que tiene que adoptar, cada UAV hace una búsqueda en el array de posiciones. Cuando encuentra su propio identificador, se guarda la posición en el array de dicho identificador. Todos los esclavos reciben el mismo array, con lo cual todos los identificadores deben de tener distinta posición, la cual nunca estará solapada con otro UAV.

Como se observa, la latitud, longitud y heading no son necesarios en este paso, pero gracias a una mejora y ampliación, que se tratará en la Subsección 7.9, se han ampliado los datos en el mensaje *MSG_TakeOff* (Subsección 7.4.2).

El UAV esclavo realiza un cambio de estado a *TAKE_OFF*, mientras el UAV que se encuentra en este estado descarta todos los mensajes que recibe.

Una vez alcanzado los objetivos del despegue, el hilo *Listener* realiza una espera del mensaje del tipo *MSG_Coordenadas*. Al ser recibido el primero, cambia de estado a *FOLLOW*. Al realizar el cambio de estado, los multicópteros esclavos entran en un bucle de seguimiento. Esto permite realizar la formación al calcular donde colocarse mediante las coordenadas obtenidas en cada mensaje, y el *offset* calculado anteriormente.

La manera de salir de este bucle es recibir un mensaje del tipo *MSG_Landing* que ordena a los esclavos aterrizar.

```
1     FollowMeParam.uavs[numUAV] = FollowMeState.SEND_ID;
2     GUI.updateprotocolState(numUAV, FollowMeParam.uavs[numUAV].getName
          ());
3
4     while (!received) {
5       message = Copter.receiveMessage(numUAV); /// Espera bloqueante
6       in.setBuffer(message);
7       idSender = in.readInt();
8       typeRecibido = in.readInt();
9       if (typeRecibido == FollowMeParam.MsgTakeOff) {
10        idMaster = idSender;
11        int size = in.readInt();
12        idsformacion = in.readInts(size);
13        // Precalculo de la formacion
14        int posformacion = -1;
15        for (int i = 0; i < idsformacion.length; i++) {
16          if (idsformacion[i] == idSlave) {
17            posformacion = i;
18          }
19        }
20        switch (FollowMeParam.formacionUsada) {
21        case FollowMeParam.formacionLinea:
22          offset = formacion.getOffsetLineal(posformacion);
23          break;
```

```
24          case FollowMeParam.formacionMatriz:
25            offset = formacion.getOffsetMatrix(posformacion, size);
26            break;
27          case FollowMeParam.formacionCircular:
28            offset = formacion.getOffsetCircular(posformacion, size);
29            break;
30          default:
31            break;
32          }
33          FollowMeParam.uavs[numUAV] = FollowMeState.TAKE_OFF;
34          GUI.updateprotocolState(numUAV, FollowMeParam.uavs[numUAV].
                getName());
35          received = true;
36        }
37      }
38
39      while (FollowMeParam.uavs[numUAV] == FollowMeState.TAKE_OFF) {
40        message = Copter.receiveMessage(numUAV);
41      }
42
43      received = false;
44      while (!received) {
45        message = Copter.receiveMessage(numUAV); /// Espera bloqueante
46        in.setBuffer(message);
47        idSender = in.readInt();
48        typeRecibido = in.readInt();
49        if (idSender == idMaster && typeRecibido == FollowMeParam.
              MsgCoordenadas) {
50          double x, y, heading, z, speedX, speedY, speedZ;
51          x = in.readDouble();
52          y = in.readDouble();
53          heading = in.readDouble();
54          z = in.readDouble();
55          speedX = in.readDouble();
56          speedY = in.readDouble();
57          speedZ = in.readDouble();
58
59          double incX, incY;
60          incY = offset.y * Math.cos(heading) - offset.x * Math.sin(
                heading);
61          incX = offset.x * Math.cos(heading) + offset.y * Math.sin(
                heading);
62
63          x += incX;
64          y += incY;
65
66          GeoCoordinates geo = Tools.UTMToGeo(x, y);
67
68          try {
69            Copter.getController(numUAV).msgTarget(FollowMeParam.
                  TypemsgTargetCoordinates, geo.latitude,
70                  geo.longitude, z, 1.0, false, speedX, speedY, speedZ);
71          } catch (IOException e) {
72            e.printStackTrace();
73          }
74
75          received = true;
76        }
77      }
```

```
78
79      FollowMeParam.uavs[numUAV] = FollowMeState.FOLLOW;
80      GUI.updateprotocolState(numUAV, FollowMeParam.uavs[numUAV].getName
            ());
81
82      received = false;
83      while (!received) {
84        long t = System.currentTimeMillis();
85        message = Copter.receiveMessage(numUAV); /// Espera bloqueante
86
87        if (System.currentTimeMillis() - t > 3000) {
88          System.out.println(numUAV);
89        }
90
91        in.setBuffer(message);
92        idSender = in.readInt();
93        typeRecibido = in.readInt();
94        if (idSender == idMaster && typeRecibido == FollowMeParam.
              MsgCoordenadas) {
95          double x, y, heading, z, speedX, speedY, speedZ;
96          x = in.readDouble();
97          y = in.readDouble();
98          heading = in.readDouble();
99          z = in.readDouble();
100         speedX = in.readDouble();
101         speedY = in.readDouble();
102         speedZ = in.readDouble();
103
104         double incX, incY;
105         incY = offset.y * Math.cos(heading) - offset.x * Math.sin(
                heading);
106         incX = offset.x * Math.cos(heading) + offset.y * Math.sin(
                heading);
107
108         x += incX;
109         y += incY;
110
111         GeoCoordinates geo = Tools.UTMToGeo(x, y);
112
113         double speed = Copter.getSpeed(numUAV);
114         if (speed < 0.1) {
115           System.err.println("Slave " + numUAV + " parado");
116         }
117         try {
118           Copter.getController(numUAV).msgTarget(FollowMeParam.
                  TypemsgTargetCoordinates, geo.latitude,
119                 geo.longitude, z, 1.0, false, speedX, speedY, speedZ);
120         } catch (IOException e) {
121           e.printStackTrace();
122         }
123
124       } else if (idSender == idMaster && typeRecibido == FollowMeParam.
              MsgLanding) {
125
126         Copter.setFlightMode(numUAV, FlightMode.LAND_ARMED);
127
128         received = true;
129       }
130     }
```

```
131
132     FollowMeParam.uavs[numUAV] = FollowMeState.LANDING_FOLLOWERS;
133     GUI.updateprotocolState(numUAV, FollowMeParam.uavs[numUAV].getName
            ());
134
135     GUI.log("SlaveListener " + idSlave + " Finaliza");
```

7.7.2. Talker del esclavo

Las funciones del hilo *Talker* son simples, y permiten realizar esperas mientras el UAV esclavo no tiene que enviar ningún mensaje, así como enviar en los momentos necesarios.

El primer mensaje que envía es del tipo *MSG_Id* (Subsección 7.4.1). Esta acción se realiza en el estado *SEND_ID*.

A continuación, el UAV realiza el despegue al entrar en el estado *TAKE_OFF*. Una vez alcanzado el objetivo en el despegue, envía un último mensaje al maestro indicando que está listo para realizar el seguimiento y la formación establecida.

Como se puede observar, el funcionamiento de hilo *Talker* es simple, realiza dos tipos de envíos y finaliza, liberando así la CPU.

```
1     GUI.log("SlaveTalker " + numUAV + " run");
2
3     while (!Tools.areUAVsReadyForSetup()) {
4       Tools.waiting(100);
5     }
6
7     while (FollowMeParam.uavs[numUAV] == FollowMeState.START) {
8       Tools.waiting(100);
9     }
10
11    Output out = new Output();
12    byte[] buffer;
13    while (FollowMeParam.uavs[numUAV] == FollowMeState.SEND_ID) {
14        // Enviar mi idSlave
15      buffer = new byte[Tools.DATAGRAM_MAX_LENGTH];
16      out.setBuffer(buffer);
17      out.writeInt(numUAV);
18      out.writeInt(FollowMeParam.MsgIDs);
19      out.flush();
20      byte[] message = Arrays.copyOf(buffer, out.position());
21      Copter.sendBroadcastMessage(numUAV, message);
22      out.clear();
23      Tools.waiting(1000);
24    }
25
26    if (FollowMeParam.uavs[numUAV] == FollowMeState.TAKE_OFF) {
27      Copter.takeOff(numUAV, FollowMeParam.AlturaInitFollowers);
28      FollowMeParam.uavs[numUAV] = FollowMeState.WAIT_MASTER;
29      GUI.updateprotocolState(numUAV, FollowMeParam.uavs[numUAV].
            getName());
30    }
31
32    while (FollowMeParam.uavs[numUAV] == FollowMeState.WAIT_MASTER) {
33      buffer = new byte[Tools.DATAGRAM_MAX_LENGTH];
```

```
34          out.setBuffer(buffer);
35          out.writeInt(numUAV);
36          out.writeInt(FollowMeParam.MsgReady);
37          out.flush();
38          byte[] message = Arrays.copyOf(buffer, out.position());
39          Copter.sendBroadcastMessage(numUAV, message);
40          out.clear();
41          Tools.waiting(1000);
42      }
43
44      GUI.log("SlaveTalker " + numUAV + " Finaliza");
```

7.8 Simulación del protocolo en su conjunto

Se han realizado a modo de ejemplo algunas simulaciones con varios UAVs, y se registraran los datos obtenidos en las siguientes tablas.

Los experimentos se realizaron con tres formaciones distintas, y con dos cantidades diferentes de UAVs.

Tabla 7.1: Mensajes enviados en los experimentos de simulación

Total Mensajes enviados	9 UAVs	25 UAVs
Formación Línea	724	1613
Formación Matriz	629	1170
Formación Círculo	705	1067

Tabla 7.2: Mensajes recibidos en los experimentos de simulación

Total Mensajes recibidos	9 UAVs	25 UAVs
Formación Línea	5792	38712
Formación Matriz	5032	28080
Formación Círculo	5640	25608

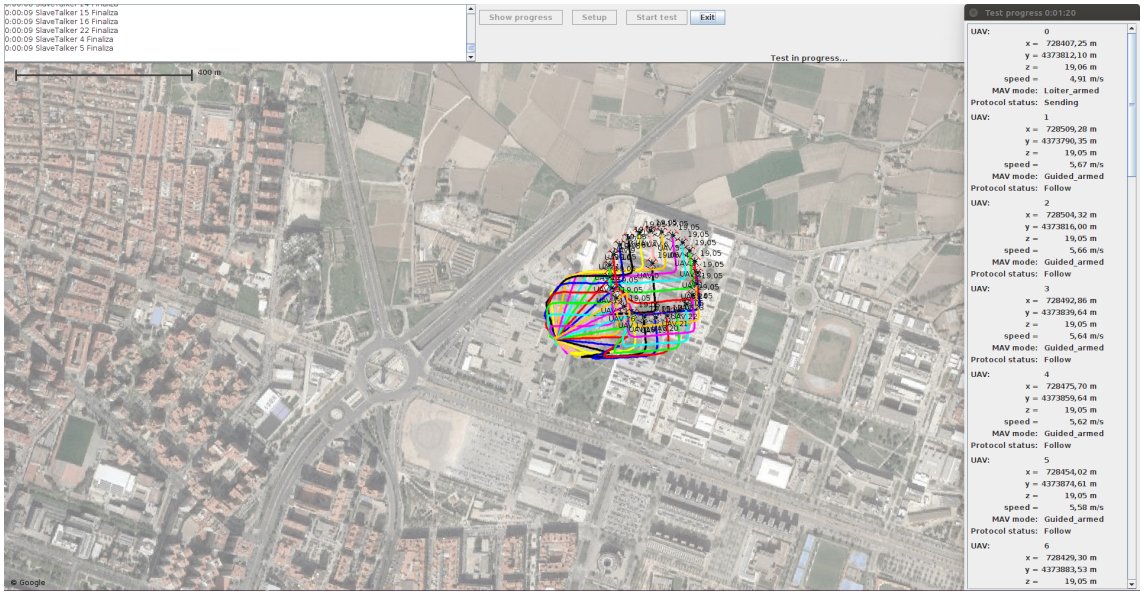

Figura 7.16: Experimento 25 UAVs formación circular

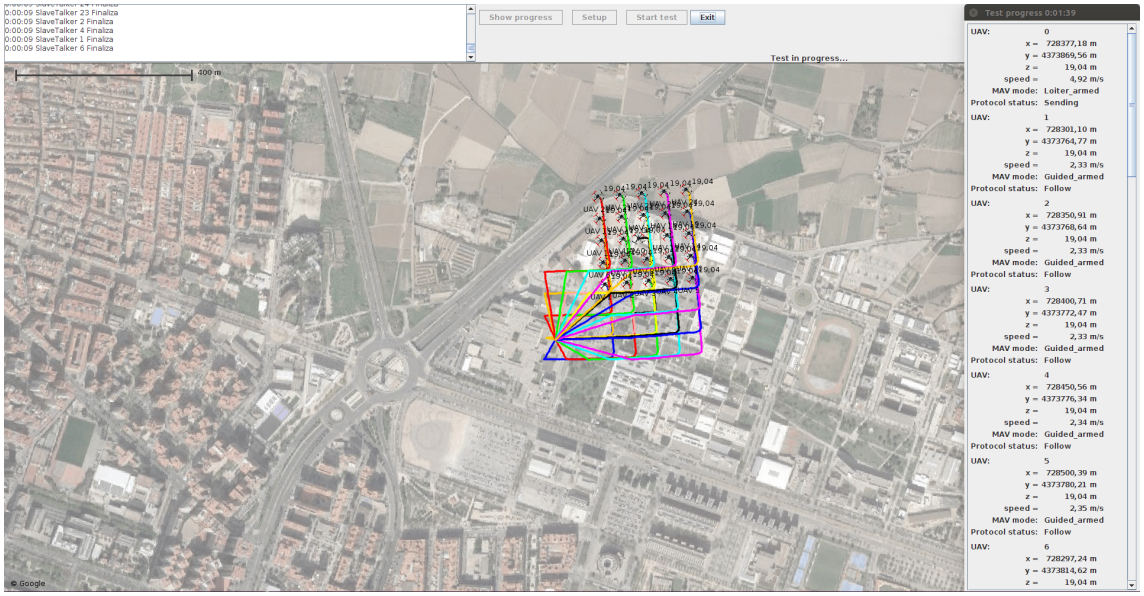

Figura 7.17: Experimento 25 UAVs formación Matriz

7.9 Mejoras realizadas

Se ha realizado alguna mejora para que el objetivo del proyecto se alcance de la manera más rápida, segura y eficiente. La principal mejora que se ha desarrollado es la de realizar un despegue en formación.

El despegue en formación implica que los UAV esclavos realizan el despegue y se sitúan en la posición donde tienen que permanecer. El despegue ordenado se tratará en la Sección 8.2.

Para realizar esta función, se necesita modificar el mensaje de tipo *MSG_TakeOff* 7.11, como se ha explicado. El maestro envía los datos adecuados para que los es-

clavos puedan calcular las coordenadas hacia donde se tiene que ubicar después de despegar.

El despegue se realiza en dos fases, primero se eleva el UAV hasta una cierta altura, y luego realiza el movimiento hacia las coordenadas calculadas previamente.

Una vez alcanzadas las coordenadas de despegue, los UAVs envían un mensaje del tipo *MSG_Ready*, de manera idéntica a como se realiza sin dicha mejora.

CAPÍTULO 8
Conclusiones y trabajos futuros

En el capitulo final se detallaran las conclusiones obtenidas tras finalizar el proyecto y posibles trabajos futuros que ofrece este proyecto una vez finalizado.

8.1 Conclusiones

Una vez finalizado el trabajo, es posible obtener unas conclusiones respecto al proyecto desarrollado. En el inicio del proyecto se han detallado un objetivo principal, además de una serie de objetivos secundarios que permiten alcanzar el objetivo principal.

El primero objetivo secundario era recopilar la información adecuada sobre un vuelo real. El vuelo fue realizado con un multicóptero de fabricación propia. Se obtuvieron trazas de tres vuelos de prueba distintos, aunque solo uno cumplía con todas las condiciones deseadas. La conclusión fue que el software desarrollado para esta finalidad, para obtención de trazas, funcionaba bastante bien, ya que de lo contrario mantener un control del dron estabilizado y preciso no es del todo fácil.

El segundo objetivo secundario trataba de replicar el vuelo realizado en el simulador ArduSim. Con la ayuda de la traza, se implementó una emisora radiocontrol simulada. Esta emisora radiocontrol reproduce el contenido de la traza, indicando al dron simulado los movimientos que tiene que realizar. El correcto funcionamiento de estos dos objetivos es beneficioso para aportar más funcionalidades al simulador.

El tercer objetivo secundario trata las comunicaciones que se tienen que realizar entre los distintos drones que han adoptado los roles de maestro o esclavo. Estas comunicaciones son esenciales para el control de la situación en todo momento, y de las esperas en el protocolo y correspondiente actuación. Las conclusiones obtenidas, dados los problemas que se ocasionaron, son las siguientes: los hilos que tienen que recibir mensajes nunca pueden quedar a la espera cuando solamente interesa la información mas actualizada que llega al buffer de recepción. Si no se leen los mensajes recibidos en el buffer, este se llena de información obsoleta y se colapsa, ocasionando un uso de CPU excesivo, debido al modo en que se trata las comunicaciones en el simulador. Lo ocurrido se detalla en la documentación de ArduSim para ayudar a otros desarrolladores.

El cuarto objetivo secundario era realizar cálculos de manera eficiente para la colocación en formación alrededor de las coordenadas indicadas. La conclusión es que, cuando se van a realizar cálculos, conviene realizar los cálculos repetitivos una sola vez. De hecho, precalcular algunas de las operaciones hace que las operaciones se realicen en menor tiempo y con mayor fluidez.

El objetivo inicial era proporcionar un software que permitiera a varios drones realizar vuelos en enjambre manteniendo una formación, y siendo liderados por un dron maestro, el cual se controla manualmente por parte de un piloto desde una emisora radiocontrol. Como se puede observar, el conjunto de los subobjetivos anteriores se complementan perfectamente de cara a permitir lograr alcanzar el objetivo principal satisfactoriamente.

8.2 Trabajos futuros

El proyecto se ha realizado como una línea de investigación, la cual obviamente puede ser ampliada.

Las propuestas de ampliación de cara al futuro son las siguientes:

- Realizar ligeros cambios para poder adaptar el programa desarrollado a UAVs reales.

- Aplicar el protocolo de despegue ordenado, desarrollado con ayuda del simulador.

- Dar robustez UDP y probar con modelo de comunicación realista

8.2.1. UAV Reales

El simulador Ardusim, permite realizar, con pequeñas variaciones del código, una implementación en drones reales. Para lograrlo, tan solo es necesario aplicar algunos cambios en los tipos de variable que contienen el identificador, y realizar cambios en las comunicaciones.

Igualmente será necesario modificar la creación de los hilos, ya que tan solo será necesario crear los hilos que van asociados al rol que tome cada uno de los UAV. El maestro será fácil de detectar, ya que es el único que dispone de una emisora radiocontrol.

El mayor inconveniente de realizar un vuelo en multicópteros reales es realmente disponer de varios UAVs. La zona donde realizar el vuelo también es un factor importante.

8.2.2. Aplicar el protocolo de despegue

En el protocolo no se tiene en cuenta de donde se inicia el despegue, y los multicópteros no tienen en cuenta si otro a su alrededor está despegando. Estos factores se tienen en cuenta en un protocolo que actualmente se encuentra en desarrollo.

Una vez finalizados los dos protocolos, éstos se podrían fusionar. De esta forma se lograría un despegue ordenado y sin colisiones.

Bibliografía

[1] Simulador de Drones Consultado el 27 de junio de 2018 https://www.dronethusiast.com/simulador-de-drones/

[2] Simulador Microsoft Consultado el 27 de junio de 2018 https://github.com/Microsoft/AirSim

[3] Origen y desarrollo de los drones. Consultado el 18 junio 2018. Cristina Cuerno Rejado http://drones.uv.es/origen-y-desarrollo-de-los-drones/.

[4] Web oficial Sistema Operativo Raspbian. Consultado el 18 junio 2018 http://www.raspbian.org/.

[5] Hacienda rastrea con drones obras sin declarar. Consultado el 18 junio 2018. Victoria Salinas, Valencia 02.01.2017 | 04:15 https://www.levante-emv.com/comunitat-valenciana/2017/01/02/hacienda-rastrea-drones-obras-declarar/1510742.html.

[6] Drones, los nuevos vigilantes de la playa. Consultado el 18 junio 2018. María Alcaraz Mayor,Actualizado: 08/05/2018 21:28h http://www.abc.es/sociedad/abci-drones-nuevos-vigilantes-playa-201805082124_noticia.html.

[7] Real Decreto 1036/2017. Consultado el 18 junio 2018. MINISTERIO DE LA PRESIDENCIA Y PARA LAS ADMINISTRACIONES TERRITORIALES. https://www.boe.es/boe/dias/2017/12/29/pdfs/BOE-A-2017-15721.pdf.

[8] Nueva ley sobre el uso de drones en España. Consultado el 18 junio 2018. Dronair. http://www.dronair.es/nueva-ley-sobre-el-uso-de-drones-en-espana-2.

[9] Drone Hopper Consultado el 18 junio 2018. Pablo Flores. https://elpais.com/elpais/2017/07/18/talento_digital/1500372485_447085.html.

[10] Lesson 4: Throttle, Yaw, Pitch and Roll Consultado el 19 junio 2018. August 7, 2016 | In CoDrone Basic Lessons | Justin Chang https://www.robolink.com/lesson-b04-flight-part-ii/.

[11] Enum Types Consultado el 19 junio 2018. Oracle https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

[12] BufferedWriter Consultado el 19 junio 2018. Oracle https://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html.

[13] MAVLink Developer Guide Consultado el 20 junio 2018. MAVLink https://mavlink.io/en/.

[14] MAVLink Common Message Set Consultado el 20 junio 2018. MAVLink https://mavlink.io/en/messages/common.html.

[15] RC_CHANNELS_RAW Consultado el 20 junio 2018. MAVLink http://mavlink.org/messages/common#RC_CHANNELS_RAW.

[16] GLOBAL_POSITION_INT Consultado el 20 junio 2018. MAVLink hthttp://mavlink.org/messages/common#GLOBAL_POSITION_INT.

[17] HEARTBEAT Consultado el 20 junio 2018. MAVLink http://mavlink.org/messages/common#HEARTBEAT.

[18] Ejecutar un programa automáticamente al iniciar la Raspberry Pi. Consultado el 20 junio 2018. http://nideaderedes.urlansoft.com/2013/12/20/como-ejecutar-un-programa-automaticamente-al-arrancar-la-raspberry-pi/.

[19] Java SE Runtime Environment 8 Downloads Consultado el 21 junio 2018. Oracle http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html.

[20] Consultado el 21 junio 2018. Ardupilot http://ardupilot.org/dev/docs/setting-up-sitl-on-linux.html.

[21] Consultado el 21 junio 2018. ArduSim https://bitbucket.org/frafabco/ardusim/src/a6b168b566a7946e65a097b67267ec7bf71e020f/help/setup.md.

[22] ArduSim: Accurate and real-time multicopter simulation Francisco Fabra, Carlos T. Calafate, Juan Carlos Cano, Pietro Manzoni Department of Computer Engineering (DISCA), Universidad Politécnica de Valencia(UPV), Camino de Vera s/n, Valencia, 46022 Spain. En desarrollo.

[23] A. D. Team, SITL Simulator (Software in the Loop) Consultado el 22 de junio 2018. http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[24] A. D. Team, SITL Simulator (Software in the Loop) Consultado el 22 de junio 2018. https://bitbucket.org/frafabco/ardusim.

[25] ConcurrentHashMap Consultado el 1 de julio 2018. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html

APÉNDICE A
Características de los dispositivos

Características multicóptero de obtención de las trazas.

Motores:

Descripción	Datos
KV	920KV
Rpm	13616rpm
Amperios	15A
Peso	54g
Alimentación	14.8V (4S)
Empuje	1100g
Consumo	217W

Variadores:

Descripción	Datos
Corriente máxima continua	30A
Corriente máxima (10s)	40A
Peso	28g
Baterías aceptadas	2-4S

Hélices:

Hélice	Consumo	Empuje
8045	195W	900g
9450	210W	1034g
1045	225W	110g

Batería:

Descripción	Datos
Capacidad	9000mAh
Tasa descarga	25C
Voltaje	14.8V (4S)

Características PC utilizado para la simulacion.

Noteboook	CX62 6QD
Placa base	MS-16J6
CPU	Intel Core i7-6700HQ @ 2.60GHz
RAM	8GB
Grafica	GeForce 940MX