



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

 etsinf

Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Extensión del protocolo MUSCOP para tolerar fallos en el dron líder durante el vuelo.

TRABAJO DE SISTEMAS BASADOS EN REDES MÓVILES

Máster Universitario en Computadores y Redes

Autores: Manel Lurbe Sempere
Izan Catalán Gallach

Curso 2019-2020

Resum

El protocol MUSCOP és un protocol per a gestionar comunicacions en vols conjunts de UAVs anomenats eixams amb un node líder o mestre que coordina als altres. En aquest treball es planteja la idea de implementar la tolerància a fallades d'aquest node líder.

Paraules clau: ArduSim, Reds FANET, Drons, MUSCOP, Protocol

Resumen

El protocolo MUSCOP es un protocolo para gestionar comunicaciones en vuelos conjuntos de UAVs llamados enjambres con un nodo líder o maestro que coordina a los demás. En este trabajo se plantea la idea de implementar la tolerancia a fallos de este nodo líder.

Palabras clave: ArduSim, Redes FANET, Drones, MUSCOP, Protocolo

Abstract

The MUSCOP protocol is a protocol to manage communications on joint flights of UAVs called swarms with a leader or master node that coordinates the others. In this work, the aim is to implement the fault tolerance of this leading node.

Key words: ArduSim, FANET Networks, Drones, MUSCOP, Protocol

Índice general

Índice general	III
Índice de figuras	IV
Índice de tablas	IV
<hr/>	
1 Introducción	1
2 Protocolo MUSCOP	3
3 Propuesta de ampliación del protocolo MUSCOP	5
3.1 Código de la ampliación	6
4 Entorno de desarrollo y pruebas	8
4.1 Requisitos del sistema	8
4.1.1 Sistema operativo	8
4.1.2 Hardware	8
4.1.3 Java Versión 13	8
4.1.4 Eclipse IDE	8
4.1.5 Google Earth	9
4.1.6 Cuenta Microsoft para mapas de Bing (opcional)	9
4.2 Instalación del entorno	9
5 Pruebas de simulación	10
5.1 Definir una misión con Google Earth	10
5.2 Como lanzar las simulaciones	11
5.3 Simulación de fallos en el dron maestro	14
5.4 Resultados de las simulaciones	14
6 Conclusiones	18
Bibliografía	19

Índice de figuras

2.1 Diagrama de funcionamiento del protocolo MUSCOP original.	4
3.1 Diagrama de funcionamiento de la propuesta de ampliación al protocolo MUSCOP.	5
3.2 Paquete donde se ubican todas las clases que componen la implementación del MUSCOP en ArduSim.	6
5.1 Selección de la ruta con Google Earth.	10
5.2 Guardar la misión en formato <i>.kml</i>	11
5.3 Crear configuración para la clase <i>main</i>	12
5.4 Establecer los argumentos para poder lanzar el simulador.	12
5.5 Fichero de velocidades <i>.csv</i> empleado.	13
5.6 Pestaña de configuración inicial del simulador.	13
5.7 Pestaña de configuración del protocolo MUSCOP.	14
5.8 Pestaña de simulación, cuando la simulación esta preparada hay que pulsar <i>Setup</i>	15
5.9 Pestaña de simulación, cuando los drones están en el aire hay que pulsar <i>Start test</i>	15
5.10 En el WP2 el dron maestro falla, sigue hasta el WP3 donde supera el <i>timeout</i> de espera de sus esclavos y vuelve a la zona de despegue. Los demás drones siguen la ruta con el nuevo maestro, UAV3.	16
5.11 Fin de simulación, el dron Maestro original ha vuelto a la zona de despegue, los esclavos originales llegan al final de la misión gracias al nuevo Maestro UAV 3.	16

Índice de tablas

CAPÍTULO 1

Introducción

Hoy en día hay un auge y propagación del uso de los vehículos aéreos no tripulados cuyas siglas son UAVs (*Unmanned Aerial Vehicles (UAVs)*) y que comúnmente se conocen como drones. Ejemplos de esto son sus aplicaciones en la agricultura para el uso de pesticidas [3] o en grabaciones científicas de investigación [4]. Además de estas situaciones, también se ha avanzado en el beneficio que puede tener el vuelo conjunto de los mismos, que se conoce como enjambres de drones (agrupaciones de varios dispositivos que colaboran en equipo).

Estos vuelos requieren muchas veces de automatizaciones para evitar que se tenga que controlar individualmente a cada dron. No obstante, dirigir el enjambre presenta dificultades en las comunicaciones dado que la distancia de seguridad que separa a los drones entre sí para evitar su choque puede suponer una pérdida de señal y paquetes perdidos. En consecuencia, se han investigado soluciones específicas para este tipo de vuelos.

En [5] se avanzó en el tema investigándose un protocolo de comunicaciones para enjambres autónomos que se centran en una misión de búsqueda. Esta propuesta combina la comunicación entre UAVs con enrutamiento geográfico para mejorar la eficiencia de búsqueda.

Posteriormente, en [6], se comenta el uso de redes ad-hoc como mecanismo para controlar la movilidad de enjambres de drones. Sus características principales se centran en el área de conectividad y cobertura, de ahí que surgiera de manera posterior [7], donde se realizaron amplios estudios sobre el uso de las Redes Flying Ad-Hoc (FANET) y como describir los principales problemas de despliegue de redes ad-hoc basadas en UAVs.

En este trabajo se va a hacer uso del protocolo MUSCOP, el cual propone definir y mantener la formación de UAVs en un enjambre mientras sigue una misión previamente planificada. Una misión es una secuencia de *waypoints* o puntos de paso que el dron debe recorrer para llegar a destino.

La comunicación inalámbrica permitirá a un dron líder coordinar al resto de UAVs cada vez que el enjambre llega a un *waypoint* de la misión. También como punto destacado, se implementa en este trabajo la tolerancia a fallos del líder o maestro del enjambre, pudiendo ser sustituido por los otros UAVs de manera que se pueda seguir con la misión planificada al principio. Esta solución se podría plantear mediante un protocolo distribuido, lo cual ralentizaría el proceso dado que se necesitaría una negociación entre los UAVs para determinar cual es el nuevo líder y consumiría las baterías en este proceso, que es un punto crítico. Debido a estas razones se ha optado por la redundancia en el UAV líder, lo que permite determinar al nuevo maestro a partir de un orden preestablecido.

Además, la propuesta ha sido validada utilizando la plataforma de simulación ArduSim [1], que nos permite realizar experimentos fidedignos, validando las formaciones con diferentes números de UAVs.

CAPÍTULO 2

Protocolo MUSCOP

El protocolo MUSCOP ha sido desarrollado para la coordinación de vuelos de UAVs. El objetivo del protocolo es que los UAVs mantengan una formación de vuelo estable mientras realizan una misión o ruta planeada. El protocolo se basa en un modelo maestro-esclavo, donde el UAV que es el maestro sincroniza a todos los UAVs esclavos cuando alcanzan el *waypoint*. Los UAVs esclavos reciben una variante de la misión o ruta principal (que solo la tiene el maestro), que consiste básicamente en una ruta paralela pero separada por unos metros de distancia para evitar cualquier colisión, de este modo los esclavos pueden alcanzar el waypoint, cada uno con su propia misión.

El protocolo mantiene unido al enjambre con este procedimiento, pero para sincronizar los dispositivos, éstos necesitan comunicarse. Para ello se usan dos hilos: *Talker Thread* y *Listener Thread*. El primero se encarga de comunicar los mensajes entre drones y el segundo los escucha y actúa en base al tipo de mensaje recibido. Para entender estas comunicaciones conviene observar a la maquina de estados de la figura 2.1.

Las flechas curvadas encima de los estados son los mensajes enviados por el hilo *Talker* mientras que las flechas por debajo representan los mensajes recibidos por el hilo *Listener*. Por último, las letras «M» y «S» se refieren al maestro y a los esclavos respectivamente. También hay que matizar que la letra «C» representa el UAV localizado en el centro de la formación, que posteriormente se convertirá en el maestro, dado que al estar localizado en el centro tiene mejor posición para emitir y recibir mensajes con éxito. Por contra, «NC» significa que un UAV no está en el centro.

Todos los UAVs comienzan en el estado *Start*. Los esclavos envían al maestro un mensaje *hello* para informarle de que están vivos y listos, cuando el maestro haya reconocido a todos, les envía la ruta a seguir con un mensaje tipo *data*. Cuando todos los esclavos hayan recibido este mensaje, el maestro entra en estado *Ready To Fly* y así lo transmite a todos los esclavos mediante un mensaje de tipo *readyToFly*. A continuación, el maestro espera el Ack de confirmación por parte de los esclavos, cuando lo recibe entra en estado de *Take Off* y retransmite un mensaje tipo *takeOff*, lo que significa que todos despegan y se sitúan en formación de salida en el aire a una distancia unos de otros. Cuando todos llegan a sus posiciones el proceso de inicio termina.

La posición inicial del UAV durante el vuelo es considerado como el primer waypoint de la misión, por lo que todos los UAVs comienzan en el estado de *Waypoint Reached*. Cuando el maestro UAV recibe el mensaje de tipo *reachedWPack* de los esclavos, comienza a moverse al siguiente waypoint (estado *Moving to Waypoint*) y fuerza a los esclavos a moverse también al siguiente waypoint mediante el mensaje *moveToWP*.

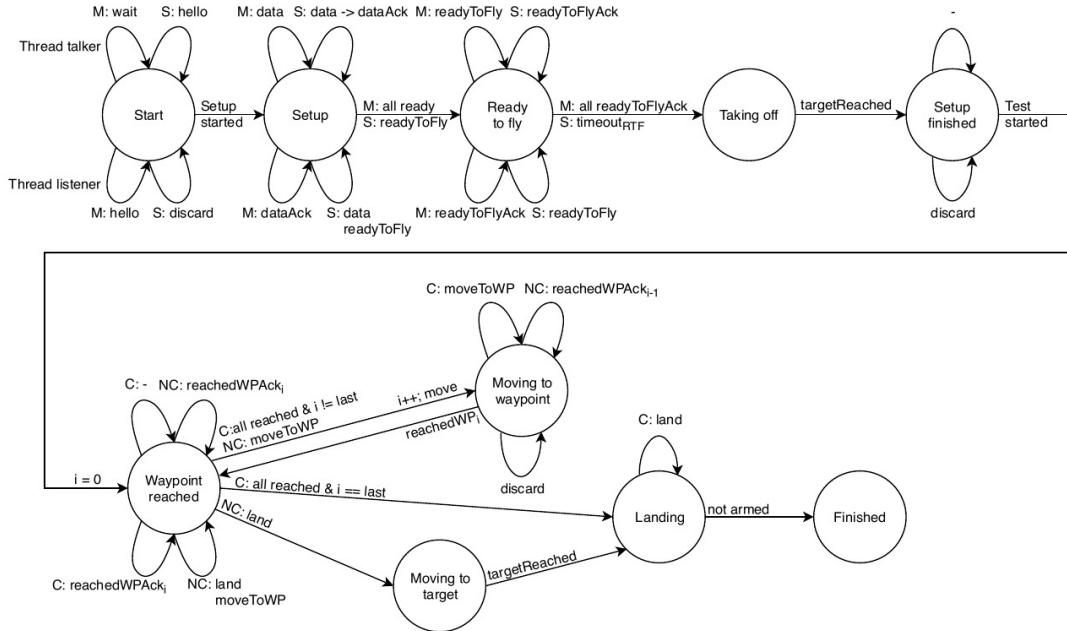


Figura 2.1: Diagrama de funcionamiento del protocolo MUSCOP original.

Todos los UAVs permanecen en el estado *Moving to waypoint* hasta que alcanzan el siguiente waypoint. Durante ese proceso, el UAV maestro está enviando continuamente un comando para moverse al siguiente waypoint, mientras que los UAV esclavos devuelven un reconocimiento de haber alcanzado el waypoint previo anteriormente. Este comportamiento redundante aumenta la fiabilidad del protocolo, ya que los mensajes enviados entre los UAV podrían perderse. Cuando todos los UAV alcanzan el último waypoint de la misión, el UAV maestro aterriza en su ubicación actual, y transmite el mensaje de *land* para que aterricen.

Ahora bien, una vez visto este complejo protocolo, falta por hacerse la pregunta de qué hacer si el maestro (que es el UAV que hace de coordinador) durante la misión falla. Este es el principal problema a tratar en este trabajo, así como los métodos que se aplican para ello y que se detallarán en el siguiente capítulo.

CAPÍTULO 3

Propuesta de ampliación del protocolo MUSCOP

Como se ha mencionado en el capítulo anterior, MUSCOP es un protocolo de control de drones basado en UAVs esclavos (Slaves) que siguen a un UAV maestro (Center) en el recorrido de una misión aérea. El problema actual de este protocolo es que si el dron maestro deja de funcionar, los demás drones no terminan la misión. Para ello se ha decidido ampliar este protocolo para tolerar fallos en el dron maestro. El diagrama que sigue esta ampliación del protocolo es el presentado en la figura 3.1

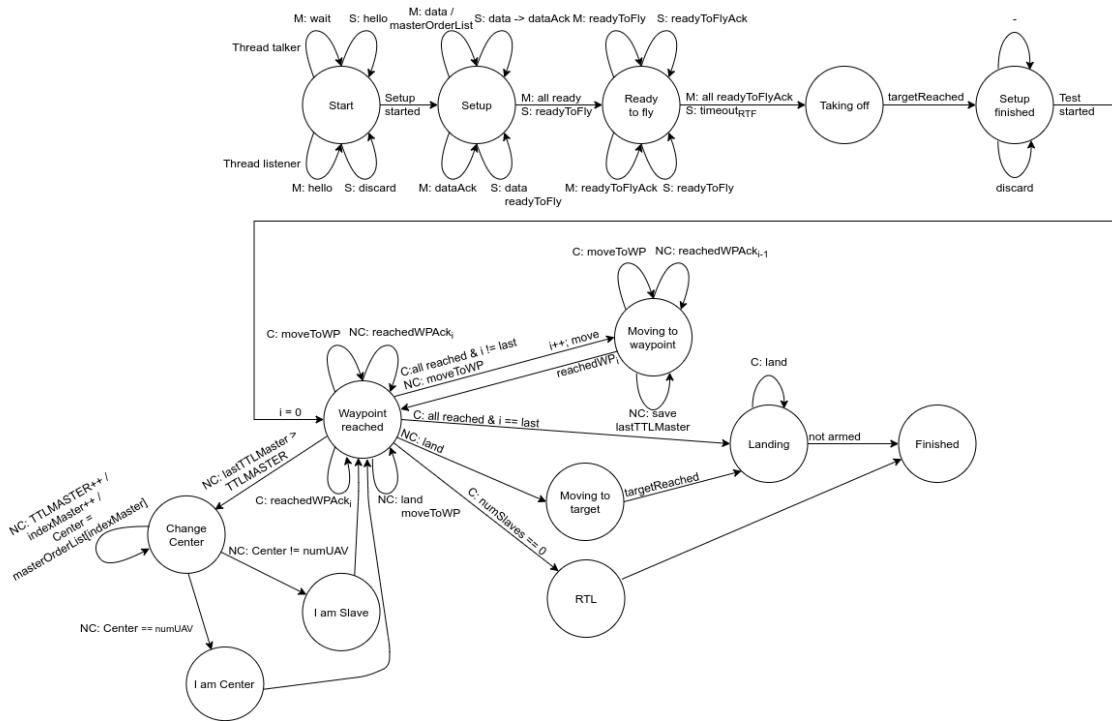


Figura 3.1: Diagrama de funcionamiento de la propuesta de ampliación al protocolo MUSCOP.

Para el desarrollo de esta ampliación se han modificado algunas funcionalidades originales del simulador así como el protocolo MUSCOP. Primeramente se ha procedido a realizar un cálculo inicial para establecer el orden en el que se asignará el rol maestro en caso de fallo. Se envía una lista ordenada a los esclavos con los identificadores de los UAVs ajustándose al orden establecido en el cálculo anterior. Dicho orden estará deter-

minado según la formación de vuelo, para establecer siempre como maestro el dron que se encuentre más cerca del centro.

Durante el vuelo los esclavos almacenan el último instante de tiempo en el que recibieron un mensaje del UAV maestro (variable *lastTTLMaster* en la figura 3.1).

Cuando los UAVs llegan a un *Waypoint* (WP), el nodo maestro sigue enviando mensajes de *moveToWP* aunque ya hayan llegado a éste, para que los esclavos puedan saber que el maestro activo sigue vivo. En el caso de que el maestro falle en este punto o hubiese fallado de camino al WP, los UAVs esclavos habrán superado el *timeout* (variable *TTLMASTER*), con lo que pasarán a cambiar de maestro (Center). Cada esclavo determina que se haya dado esta situación de manera independiente. Para ello accederán a la siguiente posición de la lista de maestros (actualizando el índice, variable *indexMaster*) que recibieron al inicio y establecerán al UAV de esta posición como su maestro. Adicionalmente, el *timeout* para considerar que el maestro ha caído (*TTLMASTER*) se aumentará al doble cada vez que un maestro falle. En el caso de ser ellos mismos el UAV maestro, tendrán que realizar un cambio de contexto en el código, para pasar a ejecutar el código correspondiente al maestro. El dron maestro sabe el número de esclavos a los que debe esperar en el WP a partir de la longitud de la lista de drones recibida antes de iniciar el vuelo.

Si llegasen a fallar todos los UAVs excepto el último (donde el número de esclavos a esperar sería 0, *numSlaves == 0*), este directamente activaría el RTL para volver a la zona de despegue para aterrizar, abortando la misión.

3.1 Código de la ampliación

En el figura 3.2 se muestra el contenido del paquete o módulo del simulador, donde se pueden encontrar el protocolo y las modificaciones comentadas anteriormente.

El protocolo consta de tres paquetes. El primero, *gui*, y el tercero, *pojo*, son necesarios para ejecutar el protocolo con el simulador y no ha sido necesario modificarlos. El paquete más relevante es el *logic*, que como se puede deducir, incluye toda la lógica del protocolo. En este tenemos 4 clases importantes (marcadas en azul) dónde se ha incluido todo el código necesario para la ampliación.

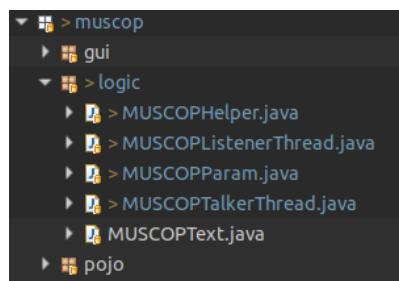


Figura 3.2: Paquete donde se ubican todas las clases que componen la implementación del MUSCOP en ArduSim.

La clase *MUSCOPHelper* es la que integra el protocolo en el simulador. En nuestro caso, solamente ha hecho falta inicializar las listas de los índices de maestros y una lista de booleanos donde los hilos de cada UAV que escuchan mensajes (*listener*) escriben en el índice de su UAV si están activos o caídos para que el hilo que habla (*talker*) lo sepa.

Seguidamente, la clase *MUSCOPListenerThread* corresponde al hilo *listener* encargado de leer los mensajes de los demás UAVs. En nuestra propuesta es el encargado de actuali-

zar el *timeout* de la última vez que se escuchó al maestro y comprobar si ha fallado. En el caso de que el maestro falle, el esclavo deberá identificar a su nuevo maestro, y si resulta ser él mismo deberá realizar un cambio de contexto en el código y empezar a actuar como tal. Adicionalmente deben ampliar el tiempo para decidir si un maestro ha caído al doble del tiempo actual. En última instancia tendrán que notificar al hilo *talker* cuál es el nuevo maestro, para que pueda cambiar de contexto en el código.

En tercer lugar, tenemos la clase *MUSCOPParam* que básicamente es donde se declaran las variables compartidas entre los hilos *talker* y *listener*, así como algunos parámetros.

Por último, la clase *MUSCOPTalkerThread* corresponde al hilo *talker*, encargado de enviar los mensajes correspondientes ya sea en contexto de maestro o esclavo. Para esta propuesta, se ha modificado la constante que identifica el dron maestro por una referencia a una variable compartida con el hilo *listener*. De esta forma ambos hilos identifican al mismo tiempo el maestro actual, pudiendo ejecutar el código correspondiente al rol asignado.

El código de la extensión del protocolo desarrollado en este trabajo se puede descargar desde nuestro repositorio de git [12].

CAPÍTULO 4

Entorno de desarrollo y pruebas

Para desarrollar la propuesta del protocolo se ha empleado el simulador ArduSim desarrollado por Francisco Jose Fabra Collado como Tesis doctoral [1]. Este simulador esta pensado para usarse en JAVA por lo que exige algunas instalaciones previas antes de poner en marcha. Todo lo necesario está explicado en el repositorio del simulador [11], pero a continuación se muestra en resumen lo que se ha realizado en este trabajo.

4.1 Requisitos del sistema

4.1.1. Sistema operativo

El simulador funciona tanto en Linux como en Windows, aunque el desarrollador recomienda usarlo preferiblemente en Linux. Para estas pruebas se ha usado un sistema basado en GNU/LINUX, concretamente la versión de Ubuntu 18.04.

4.1.2. Hardware

ArduSim es un simulador de vuelo en tiempo real con lo cual emplea muchos recursos hardware. En concreto emplea dos hilos por UAV más el main o hilo principal. El desarrollador recomienda usar un Intel Core i7 u otro similar con mínimo 8GB de RAM para no demorar mucho con las simulaciones. Para estas pruebas se ha empleado un portátil con un i7 de 4a generación con 16GB de RAM.

4.1.3. Java Versión 13

Aunque el simulador pueda funcionar con versiones anteriores de JAVA, con la versión más actual de *ORACLE JAVA 13* [8] el simulador compila y se ejecuta sin problemas.

4.1.4. Eclipse IDE

Para desarrollar y ejecutar el simulador se ha empleado el entorno de programación abierto y gratuito de JAVA conocido como *ECLIPSE IDE* [9], concretamente la versión 4.13.0.

4.1.5. Google Earth

Esta aplicación gratuita de Google [10] compatible con todas las plataformas Windows, MAC y Linux, es imprescindible para generar de forma fácil archivos .kml que incluirán la información de las misiones a seguir por el dron.

4.1.6. Cuenta Microsoft para mapas de Bing (opcional)

Como parte opcional, si se quieren realizar simulaciones con imágenes por satélite dentro del simulador, hay que activar una cuenta Microsoft y generar una clave Bing. Esta clave debe escribirse al final del fichero de configuración de ArduSim (*ardusim.ini*) que se encuentra en la carpeta raíz del proyecto.

```
1 # Optional simulation parameters:  
2 ## Bing key used to show aerial, hybrid, or street maps from Microsoft Bing.  
3 BINGKEY= PEGAR AQUI LA CLAVE
```

4.2 Instalación del entorno

Una vez tenemos todos los requisitos instalados y funcionales, hay que clonar el repositorio de Git donde se encuentra el simulador. Para ello desde el entorno de desarrollo, en el menú superior hay que abrir un proyecto desde git:

File –> Import... –> Git –> Projects_from_Git –> Clone_URI

Desde esta ventana se rellenan los parámetros de la URL del repositorio [11] para que el entorno realice un *git clone* y abra el proyecto.

Una vez en este punto se puede empezar a usar el simulador, aunque se puede recompilar el programa principal siguiendo el tutorial disponible con más detalle en el repositorio oficial [11]. Para estas pruebas no hace falta, ya que el repositorio incluye una versión compilada que nos sirve para este trabajo.

CAPÍTULO 5

Pruebas de simulación

5.1 Definir una misión con Google Earth

Para crear una simulación se necesita definir una ruta o misión de vuelo. Para ello este simulador acepta ficheros creados con Google Earth .kml que incluyen la información sobre una ruta en la superficie terrestre.

Para obtener una fichero con dicha información hay acceder a la aplicación de Google Earth y crear una ruta con puntos en la superficie del planeta allí donde interese realizar las pruebas.

Add -> Path

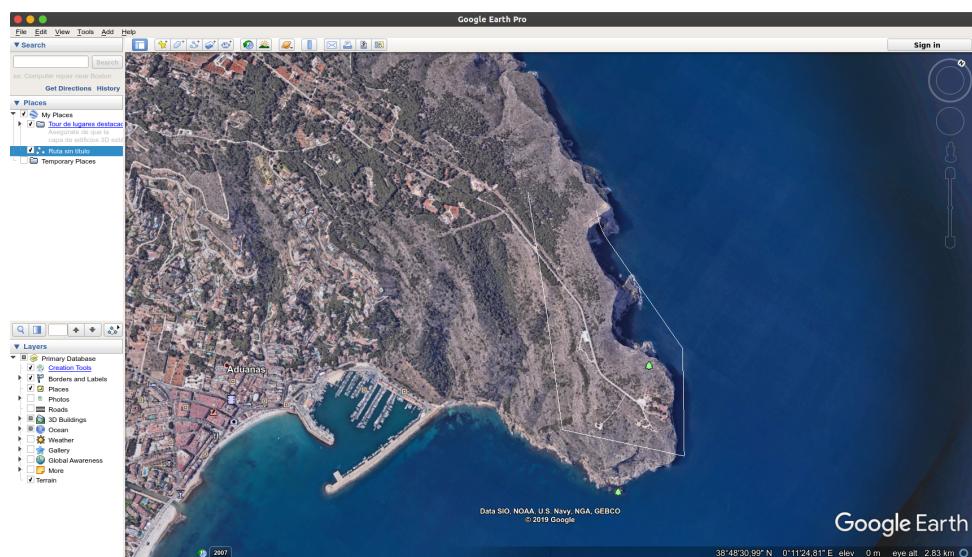


Figura 5.1: Selección de la ruta con Google Earth.

Cuando se tenga la ruta creada, aparecerá en el menú lateral *Places* una *Ruta sin título*, como en la figura 5.1. En este punto se puede guardar haciendo click derecho encima con el ratón y seleccionando *Save Place As...* y guardándola como .kml (figura 5.2).

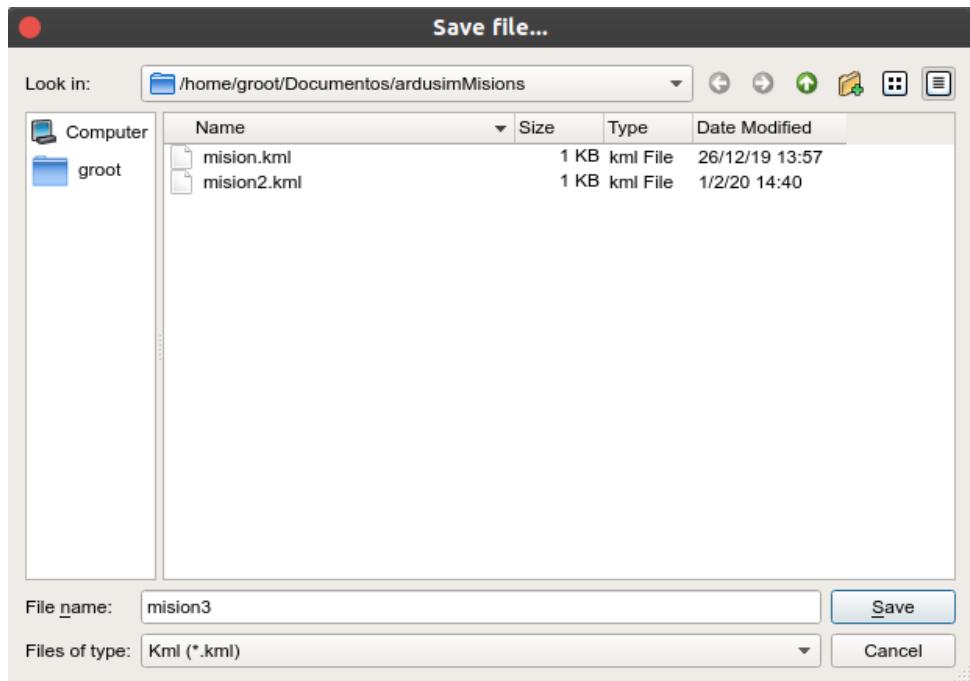


Figura 5.2: Guardar la misión en formato .kml.

5.2 Como lanzar las simulaciones

Las siguientes indicaciones son las empleadas para las pruebas de este trabajo en específico, si se desea ir más allá se pueden encontrar instrucciones más detalladas en el repositorio del simulador [11].

Desde el entorno de desarrollo con el simulador abierto, hay que abrir la configuración del RUN de ECLIPSE:

Run -> Run_Configurations... -> New_Launch_Configuration

Se tienen que llenar los campos *Project* y *Main class* como en la figura 5.3 en la pestaña principal.

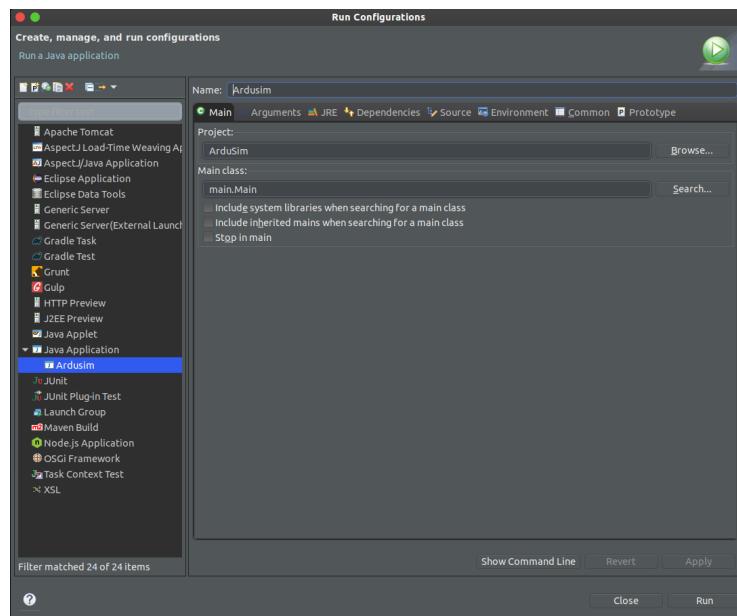


Figura 5.3: Crear configuración para la clase main.

Luego en la pestaña *Arguments* hay que poner el argumento "*simulator*" en el campo *Program arguments*, figura 5.4

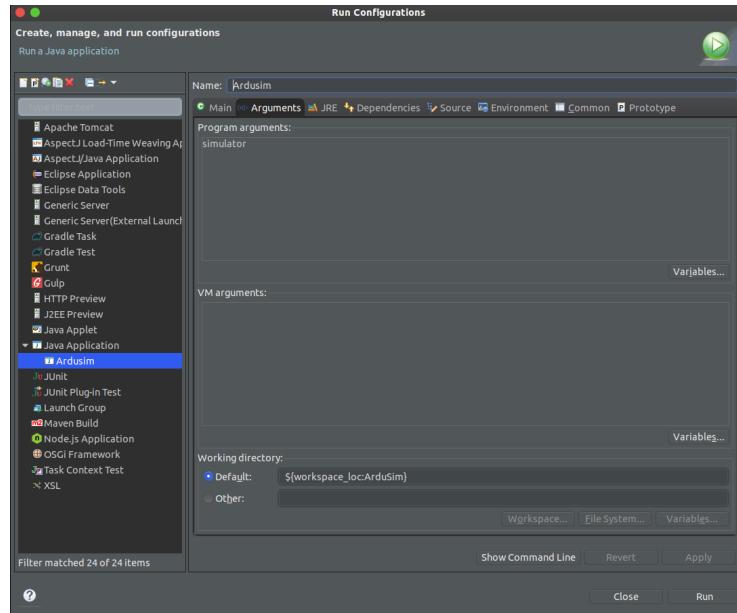


Figura 5.4: Establecer los argumentos para poder lanzar el simulador.

Una vez configurados todos los parámetros hay que darle en *Run* y se abrirá una pestaña como la de la figura 5.6. En ella se debe seleccionar un archivo de velocidades para los drones. Básicamente es un archivo *.csv*, con tantas líneas al menos como drones se vayan a simular, con valores numéricos que representan la velocidad de los mismos. En la figura 5.5 se observa un ejemplo de este fichero.

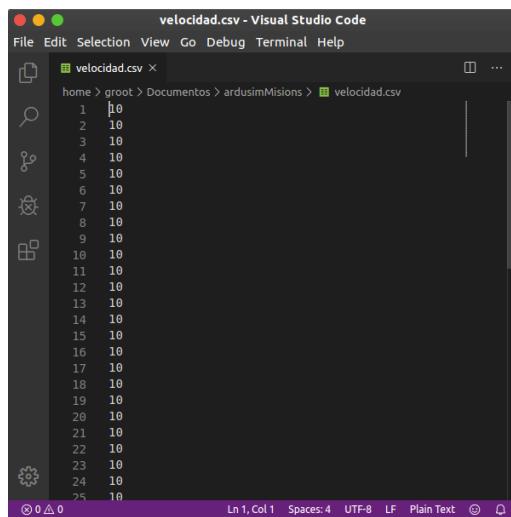


Figura 5.5: Fichero de velocidades .csv empleado.

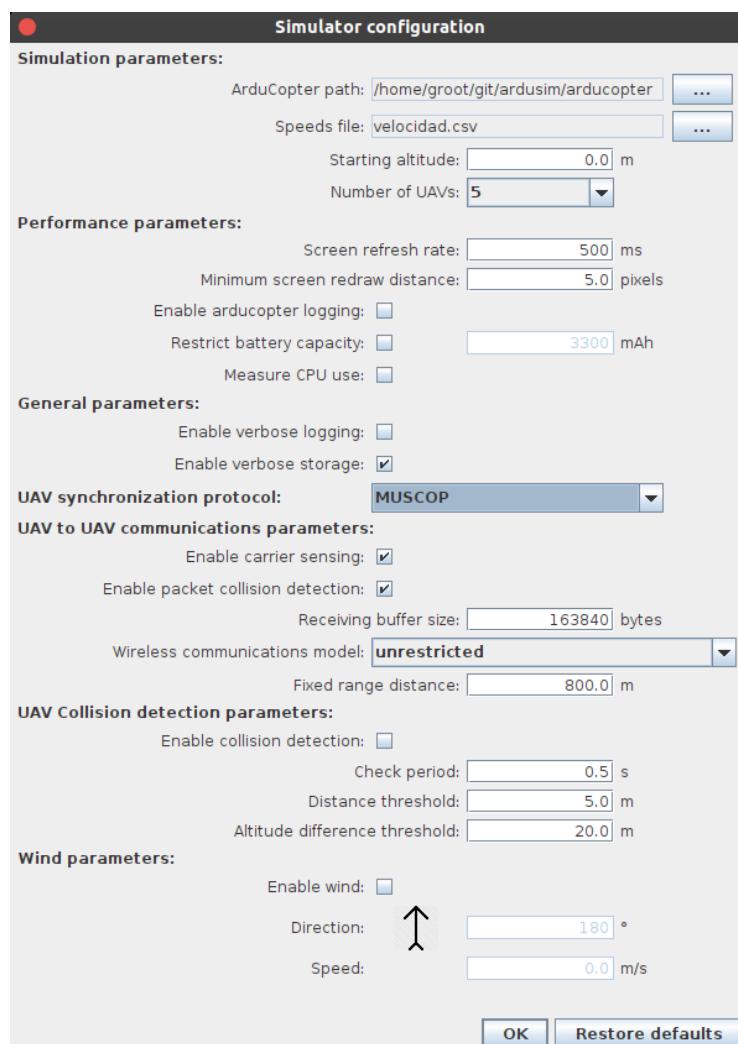


Figura 5.6: Pestaña de configuración inicial del simulador.

Para no alargar las simulaciones excesivamente se han empleado únicamente cinco drones (campo *Number of UAVs* en la figura 5.6). Por último en esta misma ventana hay que elegir el campo *UAV synchronization protocol* que básicamente establece el protocolo

a usar en la simulación, en este caso MUSCOP. Cuando se haga click en *OK* aparecerá la ventana de configuración del protocolo MUSCOP, figura 5.7. En esta pestaña hay que elegir la misión en primer lugar (campo *Flight mission*) donde se seleccionará desde el explorador el fichero *.kml* que se ha obtenido previamente con el Google Earth.

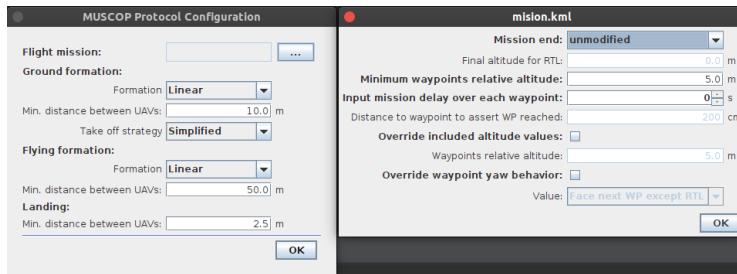


Figura 5.7: Pestaña de configuración del protocolo MUSCOP.

Seguidamente hay que elegir una formación para los drones (campo *Formation* en tierra y en vuelo), en esta simulación se ha seleccionado la versión lineal por simplicidad, aunque cualquiera sería valida. Por último se elige la estrategia de despegue simplificada (campo *Take off strategy*), para no demorar mucho el inicio del test.

5.3 Simulación de fallos en el dron maestro

A continuación se expone un ejemplo de código para bloquear la comunicación del dron maestro (en el hilo *talker*). Si el identificador del maestro es el 2 y llega al WP 2, el hilo sale del código maestro, impidiendo la ejecución del código que envía mensajes.

```

1 /*
2 * Inicio de inyección de fallo en el UAV maestro
3 * Para la formación de vuelo lineal donde el primer maestro es el UAV2
4 */
5 if (selfId==2 && currentWP == 2) {
6     gui.logUAV("El maestro 2 cae");
7     return; //Para terminar las comunicaciones del maestro
8 }
9 /*
10 * Fin de inyección de fallo en el UAV maestro
11 */

```

Este código es el empleado para la siguiente simulación y se encuentra dentro del código del maestro de la clase *MUSCOPTalkerThread*.

5.4 Resultados de las simulaciones

Una vez configurado el simulador se abrirá la pestaña de simulación y los hilos de los drones empezarán a enviarse datos y configurar diversos parámetros como el orden de despegue y la lista de maestros. Cuando esto finalice se activará un botón (*Setup*) en la parte superior que hay que pulsar para iniciar el despegue, figura 5.8.

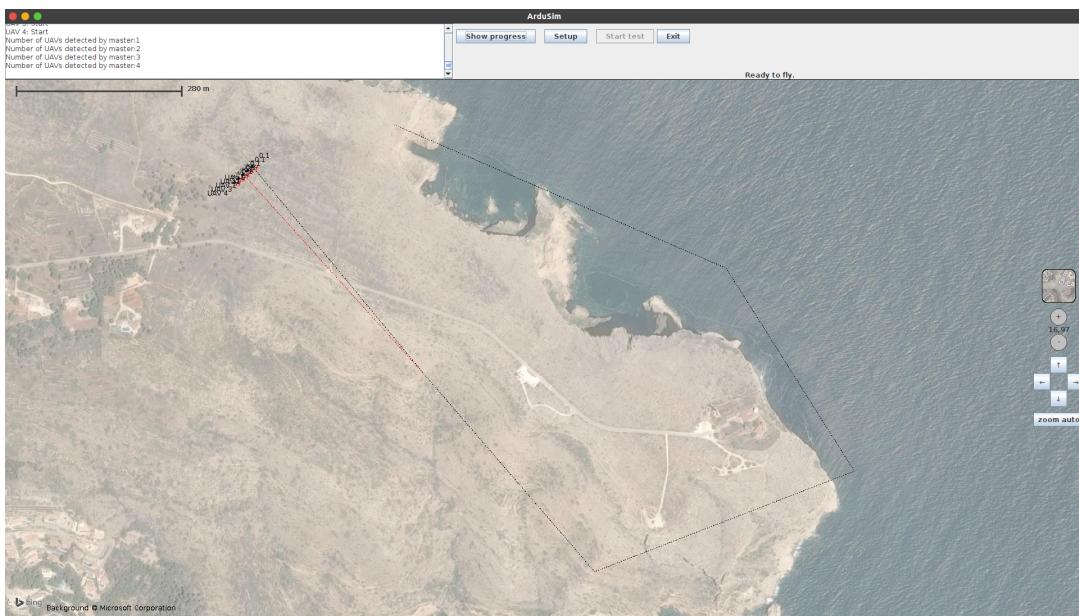


Figura 5.8: Pestaña de simulación, cuando la simulación esta preparada hay que pulsar *Setup*.

Cuando el proceso de despegue termina, se activará un segundo botón (*Start test*), que al pulsarlo iniciará la simulación, figura 5.9.

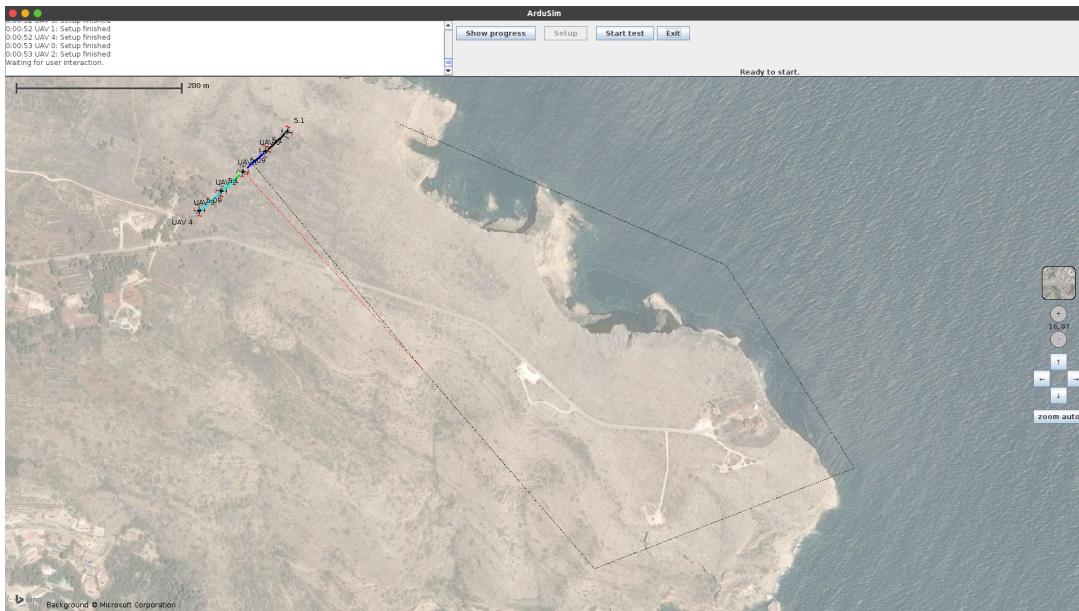


Figura 5.9: Pestaña de simulación, cuando los drones están en el aire hay que pulsar *Start test*.

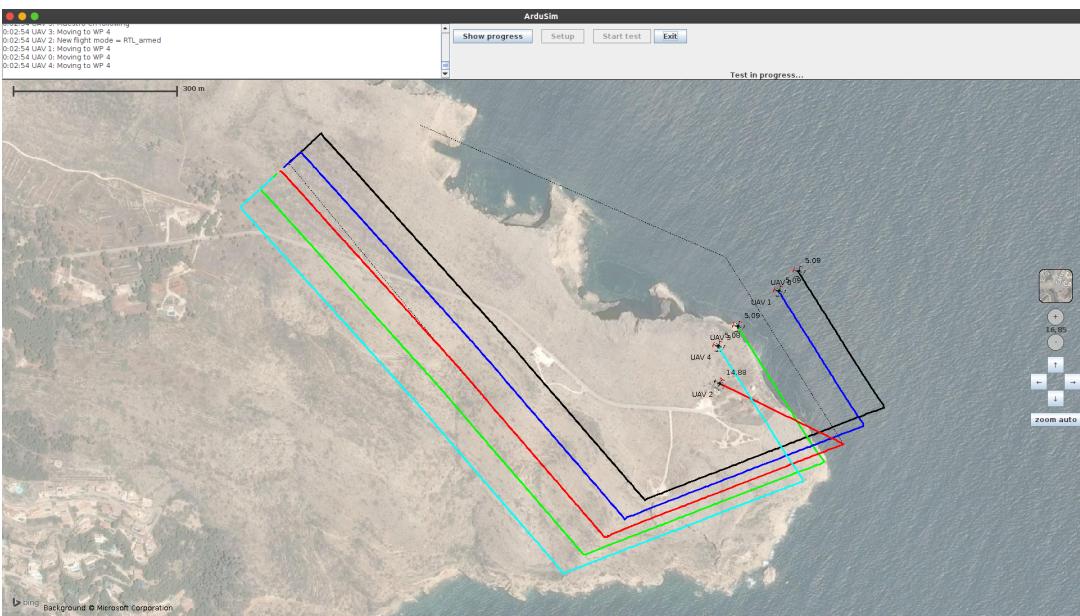


Figura 5.10: En el WP2 el dron maestro falla, sigue hasta el WP3 donde supera el *timeout* de espera de sus esclavos y vuelve a la zona de despegue. Los demás drones siguen la ruta con el nuevo maestro, UAV3.

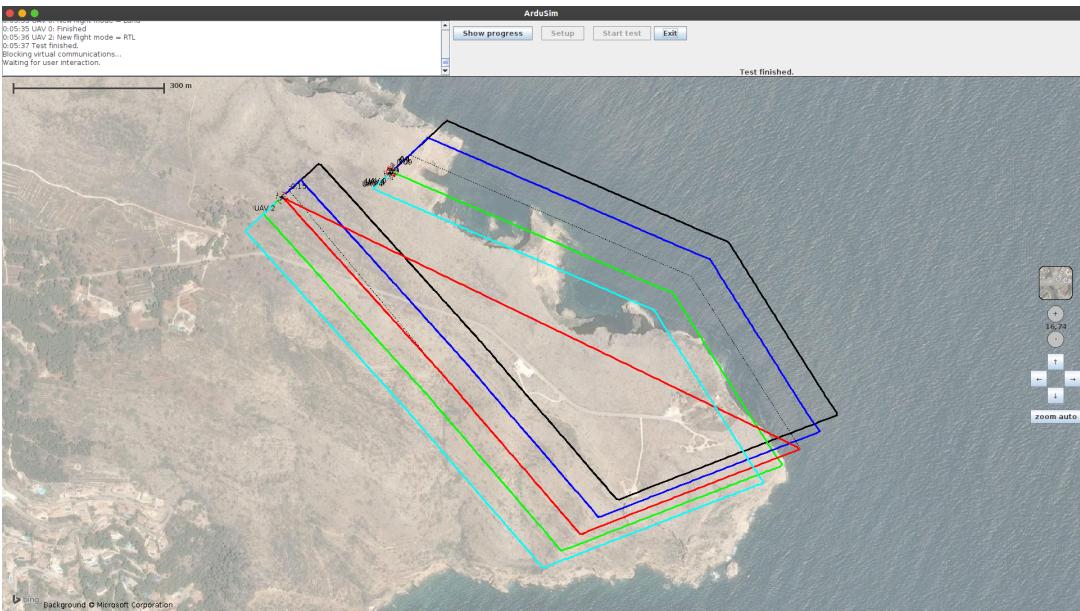


Figura 5.11: Fin de simulación, el dron Maestro original ha vuelto a la zona de despegue, los esclavos originales llegan al final de la misión gracias al nuevo Maestro UAV 3.

En las imágenes anteriores se ha expuesto una simulación controlada de la extensión del protocolo MSUCOP propuesta en este trabajo.

En el *Waypoint* (WP) 2 (primera curva pronunciada en las figuras presentadas anteriormente), el UAV2 (UAV con la línea de color rojo) deja de enviar mensajes, simulando que su antena ha dejado de transmitir. Este UAV sigue escuchando a sus esclavos pero no transmite ningún mensaje, por esta razón para los demás UAVs vence el *timeout* del maestro y deciden cambiar el líder al siguiente en la lista de maestros, en este caso el UAV3. Debido a esto el UAV2 sigue moviéndose de forma no sincronizada con respecto a sus esclavos desde el WP2 al WP3, porque sigue creyendo que él es el maestro, aunque

el resto de UAVs no le escuchen. En el WP3 el UAV2 espera un *timeout* a sus esclavos, pero como no recibe todos los ACKs, ya que el UAV3 es ahora el nuevo maestro y no envía ACK, el UAV2 realiza un RTL para volver a la zona de despegue. Mientras tanto, los demás drones siguen la misión gracias al nuevo líder, UAV3.

Esta simulación se puede visualizar de forma más detallada en la demostración en vídeo [2].

CAPÍTULO 6

Conclusiones

Los objetivos de este trabajo eran de plantear una solución óptima y válida a una carencia del protocolo MUSCOP. Esta deficiencia concretamente consiste en el fallo del UAV líder o maestro durante el proceso de la misión. En este caso anteriormente en el protocolo, si fallaba dicho UAV, el resto de esclavos se quedaban parados hasta que sus baterías se agotasen, con lo que al final todos caerían.

Este problema se ha analizado como se ha mencionado en los apartados anteriores, y se ha modificado gran parte del diagrama de estados previo. El objetivo planteado pues es que ahora se tiene una lista de posibles maestros desde el principio, que serán los más próximos geográficamente al UAV central (que es el maestro) haciendo que sea más factible que cuando emitan un mensaje, la señal llegue al resto de UAVs del enjambre.

Cuando un maestro falla, ahora se pone en funcionamiento el siguiente en la lista comunicándolo al resto de esclavos y permitiendo que todos acaben su misión y lleguen al destino con éxito. Con todo ello, se ha conseguido añadir tolerancia a fallo utilizando la base ya operativa de MUSCOP y con resultados contrastados positivamente, con lo que se puede obtener una buena dado que los objetivos planteados se han cumplido.

Bibliografía

- [1] *ArduSim: Accurate and real-time multicopter simulation*, Francisco Jose Fabra Collado, <https://riunet.upv.es/handle/10251/121381>.
- [2] *ArduSim: MUSCOP Protocol extension - Demo*, Manel Lurbe Sempere e Izan Catán Gallach, <https://youtu.be/RGRlsm1MN9c>.
- [3] *Fine-tuning of UAV control rules for spraying pesticides on crop fields*, in: 2014 IEEE 26th International Conference on Tools with Artificial Intelligence, 2014, B.S.Faiçal, G.Pessin, G.P.R. Filho, A.C.P.L.F. Carvalho, G. Furquim,J. Ueyama.
- [4] *Lightweight unmanned aerial vehicles will revolutionize spatial ecology*, *Frontiers in Ecology and the Environment* 11 (3) (2013), K. Anderson, K. J. Gaston.
- [5] *A novel communications protocol using geographic routing for swarming UAVs performing a search mission*, in: 2009 IEEE International Conference on Pervasive Computing and Communications, 2009, R. L. Lidowski, B. E. Mullins, R. O. Baldwin.
- [6] *Topology control and mobility strategy for UAV ad-hoc networks: A survey*, in: *Joint ERCIM eMobility and MobiSense Workshop*, Citeseer, 2012, Z. Zhao, T. Braun.
- [7] *Flying ad-hoc networks (FANETs): A survey*, *Ad Hoc Networks* 11 (3) (2013), I. Bekmezci, O. K. Sahingoz, Samil Temel.
- [8] Oracle JAVA JDK 13 <https://www.oracle.com/technetwork/java/javase/downloads/jdk13-downloads-5672538.html>
- [9] Eclipse IDE <https://www.eclipse.org/downloads/>
- [10] Google Earth Pro <https://www.google.es/earth/versions/#download-pro>
- [11] Repositorio Git de ArduSim <https://bitbucket.org/frafabco/ardusim/src/master/>
- [12] Repositorio git de la extensión propuesta en este trabajo <https://github.com/mlurbe97/ArduSim-MUSCOP-Protocol-SRM>

