



INGENIERÍA TÉCNICA INFORMÁTICA EN SISTEMAS

Curso Académico 2011/2017

Proyecto Fin de Carrera

ROBÓTICA AÉREA CON AVIONES

Autor : José Antonio Fernández Casillas

Tutor : José María Cañas Plaza

Proyecto Fin de Carrera

Robótica aérea con aviones

Autor : José Antonio Fernández Casillas **Tutor :** José María Cañas Plaza

La defensa del presente Proyecto Fin de Carrera se realizó el día de
de 20XX, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 20XX

*A mis padres y a mi mujer,
sin vuestra perseverancia,
apoyo y paciencia
nada de esto
hubiese sido posible.
A mi hijo Pablo*

Resumen

Los UAV o Drones se han popularizado en los últimos años hasta es punto de formar parte de nuestro día a día con aplicaciones en muchos ámbitos de nuestra vida. El objetivo de este trabajo final es desarrollar tecnología que permita la programación de aplicaciones robóticas con un avión autónomo, que se ha adaptado a partir de un avión comercial a radiocontrol y se ha simulado. Este proyecto se ubica por lo tanto en el contexto de robótica y drones, en robótica aérea. Como plataforma robótica se ha empleado JdeRobot 5.3. El lenguaje de programación ha sido Python y se han empleado las bibliotecas OpenCV y PyQt, para tratamiento de imágenes y para construir la GUI respectivamente

Summary

Here comes a translation of the “Resumen” into English. Please, double check it for correct grammar and spelling. As it is the translation of the “Resumen”, which is supposed to be written at the end, this as well should be filled out just before submitting.

Índice general

1. Introducción	1
1.1. Robótica Aérea	1
1.2. Tipos de Aeronaves	6
1.3. Aplicaciones	7
1.4. Robótica aérea en el laboratorio de Robótica de la URJC	13
2. Objetivos	19
2.1. Problema a abordar	19
2.2. Requisitos	20
2.3. Metodología y plan de trabajo	22
3. Infraestructura utilizada	25
3.1. Hardware	25
3.2. JdeRobot	27
3.2.1. Interfaces	28
3.3. Protocolo MAVLink	30
3.4. Biblioteca pymavlink	31
3.5. Lenguaje Python y biblioteca PyQt5	32
3.6. Mapas y Geo-referenciación	33
3.7. Simulador SITL	34
4. APM Server	37
4.1. Diseño	37
4.2. Bloque de conexión MAVLink	39
4.2.1. Listado de mensajes MAVLink	40

4.2.2. Conexión y configuración de la comunicación con el APM	46
4.2.3. Lectura de datos del APM	47
4.2.4. Envío de misiones al APM	48
4.3. Capa de conexión con aplicaciones JdeRobot	50
4.4. Bloque de interpretación	52
5. AUV Commander	55
5.1. Diseño	55
5.2. Bloque de conexión con aplicaciones JdeRobot	57
5.3. Core	59
5.3.1. Obtención y actualización de mapas	59
5.3.2. Utilidades de mapas	61
5.3.3. Proyección	61
5.3.4. Procesamiento de los datos de vuelo	62
5.3.5. Control de misiones	63
5.4. Interfaz de usuario	63
5.4.1. Creación y envío de misiones	65
5.4.2. Control de misiones	67
6. Experimentos	71
6.1. Experimentos simulados	71
6.1.1. Experimento de integración de todo el software	71
6.1.2. Experimento envío de misión con avión en vuelo	72
6.1.3. Experimento de envío de misión con despegue y varios waypoints . . .	73
6.1.4. Experimento de envío de misión con despegue y aterrizaje	73
6.1.5. Experimento de autozoom forzando la salida del avión del mapa . . .	74
6.2. Experimentos con el avión real	74
6.2.1. Pruebas de envío recepción del sistema de radiofrecuencias	75
6.2.2. Experimento de integración con APM Server y UAV Commander .	75
6.2.3. Experimento de seguimiento del avión a través del UAV Commander .	76
6.2.4. Experimento de envío de misión desde el avión en vuelo	76

ÍNDICE GENERAL	IX
7. Conclusiones	77
7.1. Consecución de objetivos	77
7.2. Trabajos futuros	78
7.2.1. Futuros evolutivos de APM Server	78
7.2.2. Futuros evolutivos de UAV Commander	78
7.2.3. Futuros evolutivos generales	79
Bibliografía	81

Índice de figuras

1.1.	Infografía de la Agencia EFE con la regulación	5
1.2.	Drone grabando en el interior de un crater	8
1.3.	Drone Yamaha RMAX fumigando	9
1.4.	Drone Volt Z18 UF	10
1.5.	AR Drone 2	14
1.6.	Ventanas de UAV Viewer	14
1.7.	AR Drone Simulado en Gazebo	15
1.8.	SOLO Drone de 3D Robotics	16
1.9.	Drone teleoperado con teléfono móvil	17
2.1.	Evolución de nuestra	23
3.1.	Placa estabilizadora ardupilot	26
3.2.	Imagen geo-referenciada del club de aeromodelismo Icaro	33
3.3.	Arquitectura de SITL	35
4.1.	Diseño de APM Server	38
4.2.	Diseño de APM Server	39
4.3.	Conexión con el APM	46
4.4.	configuración de la conexión	47
4.5.	Hilo de alimentación de mensajes MAVLink	47
4.6.	Protocolo de envío de misiones MAVLink	48
4.7.	Creación y arranque de los hilos encargados de servir los interfaces ICE	51
4.8.	Servidor ICE encargado de servir objetos de la interfaz Pose3D	52
4.9.	Listener encargado de detectar misiones entrantes	53

4.10. Código de ejemplo de <code>refreshAPMPose3D()</code> encargado de recorrer los mensajes y obtener los datos de actitud que se van a servir a través de Pose3D .	54
5.1. Diseño de caja negra de UAV Commander	56
5.2. Diseño de UAV Commander	57
5.3. Creación de los servicios ICE y lincado de los mismos a sus variables	58
5.4. Pase de parámetros al interfaz gráfico	58
5.5. Método <code>retrieve_new_map(lat, lon, radius, width, height)</code>	
60	
5.6. Método <code>getBoundingBox(lat, lon, distance)</code> encargado de calcular esquina inferior izquierda y esquina superior derecha de un cuadrado de ancho <code>distance</code> . .	61
5.7. Método <code>distance_to(valores, other)</code> , que calcula distancia entre puntos	62
5.8. Método <code>posCoords2Image(lonMin, latMin, lonMax, latMax, lat, lon, tamImageX, tamImageY)</code> , encargado de calcular la proyección de un punto en coordenadas geográficas a la matrix de píxeles de la imagen que representa el mapa	63
5.9. Método <code>sendWP(self, send2APM)</code> , encargado de enviar la misión creada por el usuario al driver	64
5.10. Pantalla principal de UAV Commander	66
5.11. Método <code>set_waypoints(self, wayPoints, current=True)</code> , encargado de pintar los puntos de paso en el mapa	67
5.12. Métodos encargados de actualizar la posición y el recorrido efectuado en el mapa	69

Capítulo 1

Introducción

Los UAV o Drones se han popularizado en los últimos años hasta el punto de formar parte de nuestro día a día con aplicaciones en muchos ámbitos de nuestra vida. Si bien se están utilizando ya de forma habitual en sectores como el cine o la ingeniería civil, aún se están explorando muchas de las posibles utilidades que estos robots pueden llegar a ofrecer.

El objetivo de este trabajo final es desarrollar tecnología que permita la programación de aplicaciones robóticas con un avión autónomo. Este proyecto se ubica por lo tanto en el contexto de robótica y drones, en robótica aérea.

1.1. Robótica Aérea

Los orígenes de la robótica aérea tienen origen militar y su avance ha estado intrínsecamente ligado a este ámbito durante todo el siglo XX. Se consideran el origen de los aviones no tripulados los experimentos llevados a cabo a principios del siglo XX durante la 1^a guerra mundial como el “*Aerial Target*” desarrollado por el capitán A. H. Low para su uso como blanco aéreo. Si bien eran vehículos no tripulados (*Unmanned Aereal Vehicles*) no eran autónomos y eran manejados desde tierra a través de una radio. No es hasta el final del siglo XX cuando bajo el escenario de la guerra de Vietnam y ante la creciente pérdida de vidas de los pilotos estos vehículos vuelven de nuevo a ser objeto de desarrollo y se convierten en vehículos autónomos.

Desde ese momento y hasta nuestros días se utilizan de forma habitual en el ámbito militar en misiones de reconocimiento, bombardeos o apoyo sin arriesgar vidas humanas.

A lo largo de los primeros años de este siglo debido al abaratamiento de los componentes

electrónicos y a su minituarización y potencia, la robótica aérea se ha 'desmilitarizado' y está experimentado un enorme crecimiento en el ámbito de las aplicaciones civiles.

Hoy en día es común encontrar en cualquier juguetería quadracópteros radio-pilotados por poco menos de 30 euros y en tiendas especializadas podemos encontrarlos ya con el hardware y software integrados que les permiten seguir una serie de puntos de control y comportarse de forma autónoma por poco más de 200 €.

En la actualidad el uso de AUV o drones se ha popularizado tanto que es una de las industrias en las que más ha crecido su inversión, y es que según la empresa analista especializada en drones Droneii con sede en Hamburgo, en un estudio sobre la inversión en el sector[5] en europa se invirtió en proyectos domésticos en 2016 cerca de 65 millones de dólares incrementándose esta cifra hasta los 314 millones si atendemos al mercado norteamericano.

Estos datos se asientan en un mercado cada vez más extendido y con una gran proyección de crecimiento. La publicación BI Intelligence¹[3] espera que las ventas de drones alcancen los 12.000 millones en 2021. La venta de drones es sólo una de las piezas de éste negocio incipiente, empresas como DJI, Xiaomy o 3DR estan en la punta de lanza de la innovación desarrollo de éstos, pero es tán solo la punta del iceberg. Esta industria está potenciando otras como la de las videocámaras deportivas GoPRO e incluso estan surgiendo nuevos puestos de trabajo como el de operador de drones. Las grandes empresas tecnológicas ven el potencial económico que pueden aportar a sus balances y se está produciendo una pugna por adquirir las principales empresas esencializadas en drones, Verizon compró en febrero Skyward, FaceBook compró Acenta y Google Titan Aerospace.

La enorme aceptación y expansión de los drones se ha producido de un modo tan explosivo que en ciertos aspectos de la sociedad no se ha avanzado lo suficiente como para que su utilización se haga de forma segura.

- Problemas de seguridad. Los drones y en especial los drones de gran envergadura, más allá de su uso profesional o lúdico, pueden ser muy peligrosos. Sus palas giran a más de 1000RPM y pueden producir cortes o amputaciones o producir daños personales y/o materiales en el caso de una pérdida de control del mismo o una caída. Esto se hizo patente cuando durante la filmación de un concierto del cantante Enrique Iglesias en Tijuana (Méjico) éste agarró el drone que le grababa de forma espontánea y ésto le produjo severos

¹<http://www.businessinsider.com>

cortes en su mano derecha que hizo que sangrara profusamente.

También pueden utilizarse con fines terroristas como se sospechó en Francia cuando se detectaron en octubre de 2014 volando en las proximidades de varias centrales nucleares. Además, en la noche del 19 al 20 de enero de 2015 otro 'drone' sobrevoló el Palacio del Elíseo, donde tiene su residencia el presidente de la república francesa. Y a ellos hay que sumar los 19 'drones' que han sido avistados sobrevolando 17 centrales nucleares francesas de octubre a febrero de 2015. Su bajo coste y su versatilidad producen que proliferen muchos drones construidos para fines oscuros.

Algunas empresas del sector se han hecho eco de estos ataques y han deshabilitado sus drones en zonas de conflicto para evitar que se utilicen como armas de guerra como es el caso de la empresa china DJI, líder en el sector.²

A estos problemas de seguridad hay que unir el problema del *hacking*, muchos de los más extendidos drones comerciales no tienen protección alguna contra el ataque de un ciberdelincuente. Por poner un ejemplo, el modo en el que el Parrot AR-Drone se empareja con el móvil es una conexión plana sin contraseña, haciendo posible conectarse a él por un tercero con unas pocas líneas de código.

- Guía sencilla de hacking del AR-Drone 2.0 - <http://www.xdrones.es/guia-para-hackear-el-ar-drone-2-0/>
- Cómo atacar el AR-Drone 2.0 con nodecopter - <http://www.nodecopter.com/hack>
- Entorno para *hackear* drones a través de WIFI - <https://github.com/samyk/skyjack>

Existen técnicas como el GPS Spoofing que pueden "secuestrar" cualquier drone que se guíe por GPS como ocurrió en Irán en 2011 dónde hackers iraníes se hicieron con el control de un drone militar americano Lockheed Martin RQ-170 Sentinel³ mediante ésta técnica que engaña a la antena GPS del UAV haciéndole pensar que se encuentra en otro lugar. Si bien para este ataque se necesitó un hardware caro que haría el ataque prohibitivo, hoy es posible realizarlo por unos pocos cientos de euros con placas como hackrf o braderf como se demostró por la compañía china Qihoo 360 en las conferencias de Hacking DEFCON de 2015.[10]

²<http://clipset.20minutos.es/dji-bloquea-drones-guerra-irak-siria/>

³https://en.wikipedia.org/wiki/Iran%E2%80%99S._RQ-170_incident

- Problemas regulatorios. Otro problema importante que apenas se está empezando a abordar es el regulatorio. Con el fin de minimizar el riesgo para las personas se está llevando a cabo una regulación del sector en todo el mundo. En España por ejemplo hasta abril de 2014, volar un drone para uso civil estaba absolutamente prohibido. La no regulación no indicaba que se pudiese volar, indicaba más bien todo lo contrario. En este marco regulatorio no se puede por ejemplo, volar un drone en núcleos urbanos, sólo debe hacerse en zonas previstas a tal efecto. Tampoco es posible volar un drone de más de 2Kg más allá de donde puedas verlo (500m), y es que antes era muy usual ponerle una cámara y volar varios kilómetros con él.

Para hacernos una idea clara del marco regulatorio actual para su uso no profesional basta con fijarnos en la siguiente Figura 1.1 publicada por la agencia EFE.

Para poder utilizar los drones con fines profesionales la ley es aún más restrictiva. La Agencia Española de Seguridad Aérea AESA exige el registro de las aeronaves civiles pilotadas por control remoto cuya masa máxima al despegue exceda de 25 kg, que deberán estar inscritas en el Registro de matrícula de aeronaves y disponer de certificado de aeronavegabilidad. Para drones de menor peso, el piloto deberá presentar ante la Agencia Estatal de Seguridad Aérea una comunicación previa y declaración responsable) con una antelación mínima de cinco días al día del inicio de la operación y éste ha de estar habilitado como operador de drones. La ley completa se puede consultar el BOE⁴

Con su uso ya ampliamente extendido en sectores como el cine, la televisión, fotografía, agropecuario, forestal, ingeniería civil y presencia en sectores como el de salvamento o seguridad y protección, el nicho de mercado de los UAV esta lejos de su cima. Día a día se investigan nuevos usos en sectores como el de la logística o las telecomunicaciones, como veremos más adelante, suponiendo un reto constante para investigadores, desarrolladores, ingenieros, inversores y entidades regulatorias.

⁴http://www.seguridadaerea.gob.es/media/4243006/rdl_8_2014_4julio.pdf

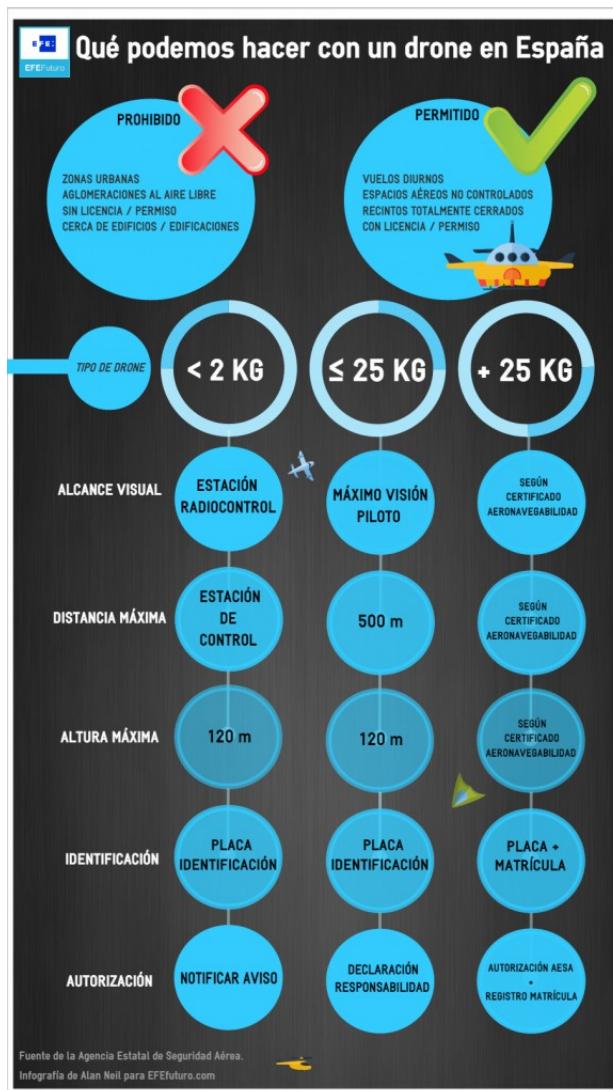


Figura 1.1: Infografía de la Agencia EFE con la regulación

1.2. Tipos de Aeronaves

Las aeronaves son la base que permite que nuestro robot vuele, de ahí que convenga dedicar unas líneas a entender la base de las mismas y en particular las que son objeto de estudio y desarrollo en este PFC, los llamados aerodinos.

Los aerodinos son aquellas aeronaves que vuelan a pesar de pesar más que el aire, son capaces de generar sustentación por sus propios medios a diferencia de los aerostatos como por ejemplo los globos aerostáticos. Existen principalmente 2 tipos de aerodinos si atendemos al modo en que generan su sustentación con sus alas: de ala fija y las de ala rotatoria.

Dentro de los primeros tenemos aquellos aerodinos que tienen sus alas fijas al fuselaje, y que comúnmente conocemos como aviones. Según la OACI, un avión es un «Aerodino impulsado por motor, que debe su sustentación en vuelo principalmente a reacciones aerodinámicas ejercidas sobre superficies que permanecen fijas en determinadas condiciones de vuelo». Algunos ejemplos de aerodinos de ala fija son los aeroplanos, planeadores/veleros, aladeltas, parapentes, paramotores y ultraligeros.

Este tipo de aerodinos tienen como principal ventaja que la carga de aire que necesitan en sus alas puede ser producida de muchas formas distintas (los veleros no tienen ningún tipo de propulsión). Esta carga es variable en función de la superficie alar del mismo y permite por tanto cargas más grandes que si instalásemos el mismo propulsor en un ala rotatoria. Pongamos como ejemplo el A380 de Airbus, es el avión de pasajeros más grande del mundo y cuenta con 4 motores que producen un empuje de entre 70.000 y 80.000lbs, unas 32-36 toneladas de empuje cada uno generando por tanto entre los 4 a máximo rendimiento y optimas condiciones alrededor de 144 toneladas de empuje. Este avión tiene un peso máximo al despegue⁵ de entre 560 y 590 toneladas. Tenemos por tanto que necesitamos en este caso $\frac{1}{4}$ del peso total en empuje para despegar este avión. Si hiciésemos este mismo ejercicio con un aerodino de ala rotatoria como el Boing AH-64 o Apache con un peso máximo al despegue de 9,5 toneladas necesitaríamos que la combinación que realizan empuje y palas superase esos 9,5 toneladas para siguiendo levantar del suelo. Este tipo de aerodinos son por tanto más eficientes, rápidos, con mayor carga de pago, mayor alcance debido a su menor consumo y más estables.

Dentro de la tipificación de ala rotatoria tenemos aquellas aerodinos que producen su sus-

⁵Peso máximo que es capaz de soportar un avión en su maniobra de despegue

tentación con el movimiento (rotación) de sus alas. En este tipo de aerodinos las alas, también llamadas 'palas', que giran en torno a un eje produciendo con este giro la sustentación necesaria para despegar del suelo. Algunos ejemplos de este tipo de aerodinos son los helicópteros, autogiros, convertibles o los ampliamente conocidos en robótica aerea los quadricópteros. Este tipo de aerodino tiene como principal ventaja frente a los ala fija en su versatilidad a la hora de realizar las maniobras de despegue y aterrizaje que pueden realizarse de forma vertical (VTOL⁶), además de la capacidad de realizar vuelo estacionario⁷ que le hacen imprescindible en escenarios poco accesibles o donde no es posible aterrizar como el rescate marítimo.

1.3. Aplicaciones

La robótica aérea ha experimentado un crecimiento exponencial en los últimos años, se ha popularizado su uso y se ha extendido la comercialización de drones.

El sector donde más rápida acogida ha tenido la robótica aérea ha sido el sector audiovisual, se usa de forma habitual en cine, grabación de espectáculos en directo televisión y fotografía. El porqué de tal acogida se basa principalmente en dos factores, los costes y la viabilidad técnica. Los costes de realizar una toma aérea en una producción antes pasaban por el alquiler de un helicóptero, dependiendo de la toma, así como del material y la contratación de los medios humanos a bordo para realizarlas podía ascender a entre 4000 y 6000 euros la hora. Hoy en día basta con un pequeño dron de entre 400 y 1800 euros⁸ por jornada, e incluirían en el tramo más elevado piloto y operador de cámara. Como se puede constatar fácilmente, el ahorro incurrido no es desdeñable y ofrece a pequeñas productoras acceder a éste tipo de grabaciones que de otro modo serían privativas. Aunque el aspecto económico es especialmente importante para decantarse por el uso de este tipo de medios, existen en el mundo audiovisual trabajos que no hubiesen sido posibles sin los drones. Grabaciones en las que el riesgo humano y material que habría que asumir es tan alto que tan solo son factibles de este modo. Debemos por tanto a los drones fotografías como la de la Figura 1.2 que si bien nos se tomó como parte de ningún proyecto cinematográfico nos hace una idea de lo que la robótica aérea puede ofrecernos.

⁶Vertical take off and landing

⁷Mantenerse estáticamente en un punto elevado

⁸Precios aproximados extraídos de la empresa especializada World Aviation Helicopters <https://www.worldaviation.es/es/servicio-drones.aspx>



Figura 1.2: Drone grabando en el interior de un crater

Otro sector que ha adaptado rápidamente el uso de éstas pequeñas aeronaves es el agropecuario donde se utiliza para medir las condiciones del terreno, con el fin de recoger información sobre la hidratación, la temperatura o el ritmo de crecimiento de los cultivos. Con el punto de vista e imágenes de los drones el agricultor puede tener en tiempo real una foto del estado de su viñedo localizando con facilidad zonas afectadas por plagas o incluso vigilar el cultivo para evitar la entrada de intrusos. Controlan el riego e incluso esparcen los pesticidas de manera eficiente siendo un arma eficaz contra las plagas, se utilizan incluso como espantapájaros. La aplicación de drones en este sector se remonta a 1983 cuando el Ministerio de Agricultura de Japón preocupado por el envejecimiento de su población rural encargó a Yamaha el desarrollo de una aeronave no tripulada capaz de realizar varias tareas de las anteriormente descritas, a fin de atraer más gente al medio rural. En 1990 se entregaron las primeras unidades del Yamaha RMAX y actualmente el 40 % de los arrozales japoneses cuentan con un drone sobrevolándolos. Como método de riego, el drone tiene dos contenedores que pueden albergar una cantidad de 8 litros cada uno y en caso de utilizarse para esparcimiento de semillas o material granulado, posee otros dos contenedores que se pueden adaptar los cuales tienen una capacidad de 13 litros cada uno. Puede elevarse a una altura de 400 metros y dispone de un sistema de GPS que no solo le permite mayor estabilidad durante el vuelo, sino que también le proporciona al usuario

la posibilidad de programar la ruta desde antes de ser lanzado. Tiene una autonomía de una hora durante el vuelo.



Figura 1.3: Drone Yamaha RMAX fumigando

Los drones están empezando a popularizarse en las empresas de seguridad privada. El punto de vista y alcance que proporcionan hacen de los drones herramientas muy potentes y versátiles. Se utilizan principalmente en vigilancia perimetral, dónde un drone puede recorrer una ruta de forma autónoma proporcionando imágenes de la misma a los vigilantes del centro de cámaras o a los *smartphones* o *tablets* de los guardias de seguridad. Se les utiliza también como apoyo a la vigilancia tradicional, sobrevolando con ellos zonas de grandes aglomeraciones o apoyando desde las alturas a las patrullas de vigilancia proporcionando, por ejemplo, imágenes térmicas de una zona durante la noche. Empresas como la francesa Drone Volt, líder en el sector de los drones profesionales, ha realizado un avance en materia de seguridad con el modelo *Z18 UF (Unlimited Flight)* un drone umbilical capaz monitorizar 24 horas de forma ininterrumpida. Esto es posible gracias a que va unido con un cable a una fuente de alimentación en tierra y por tanto se reduce la principal limitación de los drones, el tiempo de vuelo que proporcionan baterías pequeñas y ligeras.



Figura 1.4: Drone Volt Z18 UF

En España varias empresas de vigilancia privada ofrecen sus servicios con drones, algunas de ellas son:

- Aero Cámaras - <http://aerocamaras.es/servicios-drones-profesionales/drones-vigilancia>
- Prosegur - <https://www.prosegur.es/newsdetails/drone-vigilancia-interiores>
- SkyDron - <https://www.skydron.es/seguridad-aerea-privada-drones/>

Son utilizados también por los servicios de rescate tras una catástrofe para evaluar los daños experimentados y ayudas a encontrar supervivientes entre los escombros. Algunos son capaces de enviar a los supervivientes paquetes de supervivencia con salvavidas, alimentos o agua mientras esperan su rescate⁹

Es muy extendido en el mantenimiento de infraestructuras, como su uso en trabajos como:

- La inspección de tendidos eléctricos, dónde antes se utilizaban helicópteros. Ahora con los drones se obtienen mejores imágenes e inspecciones mas precisas y meticulosas sin arriesgar vidas humanas y por una mínima parte del coste que se asumía.

⁹<http://www.elmundo.es/economia/2015/09/15/55f8239546163fc6598b45c3.html>

- Estos mismos drones también se utilizan para la revisión de las palas de los aerogeneradores, evitando que el personal de mantenimiento tenga que descolgarse desde ellos para su análisis.
- Se utilizan también drones para revisar oleoductos, especialmente en zonas de difícil acceso.
- Conservación de carreteras, detectando el estado de conservación de las calzadas, obstáculos o potenciales situaciones de peligro debido a accidentes o condiciones meteorológicas adversas.
- Supervisión y mantenimiento de presas y embalses.

La empresa española Ferrovial fue la pionera en España en obtener la licencia para este tipo de trabajos¹⁰

Es muy importante su uso en topografía principalmente para obtener topografía aérea mediante técnicas de fotogrametría¹¹, la empresa española OHL dispone de varios drones a tal efecto y tiene varios proyectos de i+d+i para seguir profundizando en esta materia. De esta manera se pueden estudiar obras en su fase de licitación, realizar cálculos de volúmenes y superficies en acopios, control de certificaciones, estudio de patologías como deslizamiento de taludes y realizar seguimientos. OHL Se utilizan en minas a cielo abierto para entre otras muchas funciones:

- Control y monitorización de explotación de minerales y su impacto ambiental
- Observación de operaciones con necesidad de supervisión aérea y realizar seguimientos de movimientos de tierra y acumulación de residuos.
- Seguimiento de movimientos de tierra, residuos, balsas, control de relaves,control de pilas,transporte de implementos de un punto a otro.
- Realizar la reconstrucción de una mina mediante fotogrametría para medir los volúmenes extraídos.

¹⁰<http://www.ferrovial.com/es/prensa/noticias/ferrovial-servicios-obtiene-la-licencia-para-utilizar-drones-en-su-operacion-minera>

¹¹Técnica para obtener mapas y planos de grandes extensiones de terreno por medio de la fotografía aérea.

- Analizar la variación de alturas entre periodos para determinar los volúmenes explotados por periodo.

E incluso se han empezado a utilizar por el Ministerio de Hacienda de España con fines recaudatorios, sobrevuelan las viviendas o fincas localizando edificaciones no registradas por el catastro con el fin de regularizar su situación.

El uso de drones es especialmente importante en el ámbito científico para por ejemplo tomar medidas de temperatura y CO₂ en zonas peligrosas¹², estudiar las nubes volcánicas¹³ o volar dentro de una grieta en un glaciar¹⁴.

El futuro del uso de los drones se está escribiendo en este mismo momento y es que poco a poco se investigan con nuevos usos o cómo mejorar los ya actuales.

Uno de los usos más prometedores que está siendo investigado es el de logística y paquetería, Amazon estudió en 2015 la viabilidad de utilizar drones para el reparto, especialmente en zonas de difícil acceso o bien alejadas de las zonas habituales de reparto. A finales de dicho año realizó pruebas de entregas en el Reino Unido y desarrolló un prototipo de alrededor de 25Kg de peso. El objetivo de Amazon, en este experimento, era realizar entregas de hasta 3Kg de peso en menos de 30 minutos y ver la viabilidad de la entrega de paquetes en zonas pobladas.

Esto ha dado pie a las principales empresas de logística y paquetería a realizar sus propios desarrollos y construir sus propios prototipos. DHL ha realizado sendas pruebas con drones de cerca de 5 Kg para la entrega de paquetes de hasta 1,2 Kg. La empresa francesa GeoPost ha comprado recientemente la empresa Atechsys, especializada en el desarrollo de sistemas autónomos para aeronaves no tripuladas. Operación que ha dado como resultado un drone que cuenta con seis motores eléctricos y estructura de fibra de carbono, con capacidad para llevar paquetes de un peso aproximado de hasta 2 kilos. UPS ha realizado un experimento en Tampa con un octocóptero que despegaría desde el techo de la furgoneta de reparto para entregar los paquetes en zonas rurales y ahorrar en kilometraje el drone tendría una carga de pago de unos 4,5Kg.

Otro de los estudios que más llama la atención es el de Google y Facebook que compiten en llevar internet a zonas aisladas a través de una red de drones y satélites. Facebook presentó

¹²<http://www.igepn.edu.ec/servicios/noticias/1395-medidas-de-temperatura-y-co2-de-las->

¹³<https://www.nasa.gov/topics/earth/earthmonth/volcanic-plume-uavs.html>

¹⁴http://tn.com.ar/tecnologia/recomendados/increible-un-drone-volo-dentro-de-un-glaciar_648661

en su F8 en Marzo de 2015 su prototipo de drone a tal efecto, Aquila, fruto de la adquisición de la empresa especializada en robótica aérea Acenta. Este proyecto se enmarca dentro del plan internet.org y liderado Connectivity Lab¹⁵ que pretende ofrecer internet con un coste reducido a todo aquel que no lo tenga.

En 2014 Google por su parte compró Titan Aerospace para crear una flota de drones impulsados por energía solar, capaces de volar más de una semana mientras tomaban fotos de la superficie y proveían de acceso a Internet a lugares remotos y proveer nueva información para sus mapas. Este desarrollo sin embargo se ha abandonado en enero de este año en pro de la utilización de globos aerostáticos para tal propósito.

1.4. Robótica aérea en el laboratorio de Robótica de la URJC

Dentro del laboratorio de robótica de la Universidad Rey Juan Carlos cabe destacar varios trabajos realizados para profundizar investigar y experimentar con drones y que han servido de antecedentes directos y base para este Proyecto Fin de Carrera. Todos ellos han usado como plataforma software JdeRobot http://jderobot.org/Main_Page. Se resumen a continuación los más importantes.

Trabajos como el de Alberto Martínez[6] permitió controlar un AR-Drone 2 de Parrot real desde una aplicación cliente, desarrollando para ello un driver para el dron, `ardrone_server` y una aplicación cliente `UAV Viewer`. El driver `ardrone_server` es capaz de conectar con el AR-DRONE a través de comandos AT tanto para obtener los datos de todos sus sensores como para controlarlo y expone todo ello en interfaces JdeRobot para su interconexión con todo el software disponible en el entorno.

El AR-Drone es un quadricóptero que en su versión actual cuenta con acelerómetros, giroscopos y magnetómetros en 3 ejes que determinan junto con su barómetro y un sensor de ultrasonidos la actitud del mismo, dispone además de 2 cámaras, una ventral y otra frontal, y algunos modelos traen sensor GPS.

¹⁵Un equipo formado por 50 expertos en aeronáutica y ciencia espacial



Figura 1.5: AR Drone 2

Alberto desarrolló también el software de control UAV Viewer, esta aplicación cliente es capaz de controlar con cualquier drone y mostrar de forma visual su actitud, lo que captan sus cámaras.

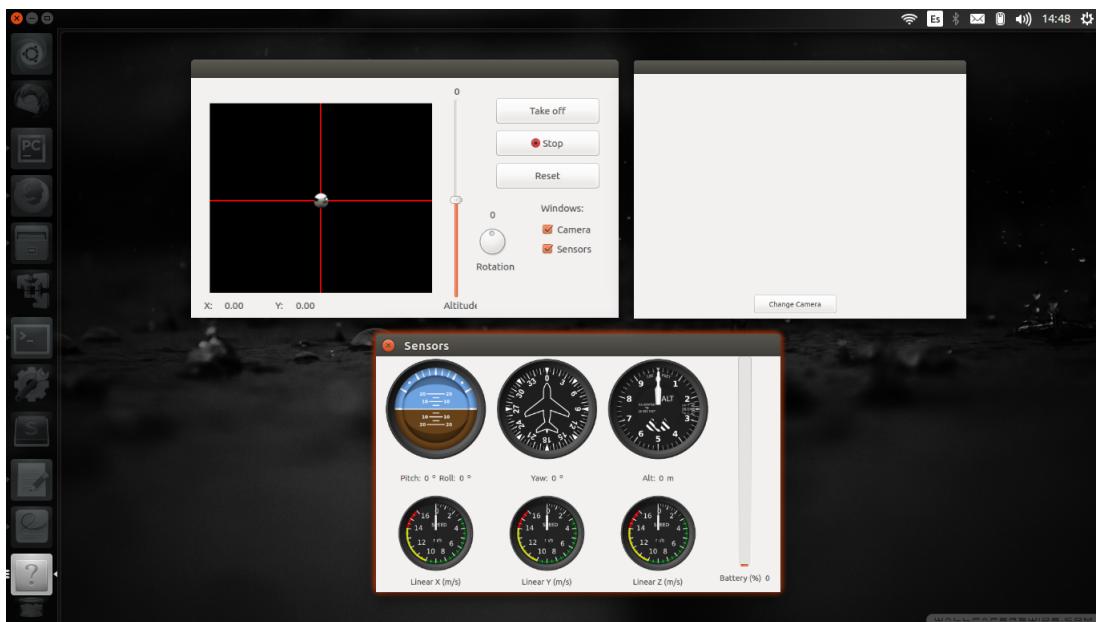


Figura 1.6: Ventanas de UAV Viewer

Daniel Yague desarrolló un driver (plugin) para simular el comportamiento del AD-Drone en el simulador Gazebo, de referencia en el Laboratorio de Robótica de la universidad y contenido dentro de la suite JdeRobot[1]. Con este driver se hizo posible probar, anticipar los problemas que puedan surgir en el vuelo del drone antes siquiera de volarlo físicamente. De esta forma es posible el desarrollo de aplicaciones de navegación complejas sin disponer de él, sin arriesgarlo e independientemente de factores externos como la climatología. Para mostrar las posibilidades

que se abría desarrolló varias aplicaciones en el ar-drone simulado como un gato-ratón donde un drone teleoperado trataba de huir de otro que le seguía o una aplicación donde el drone seguía una carretera.

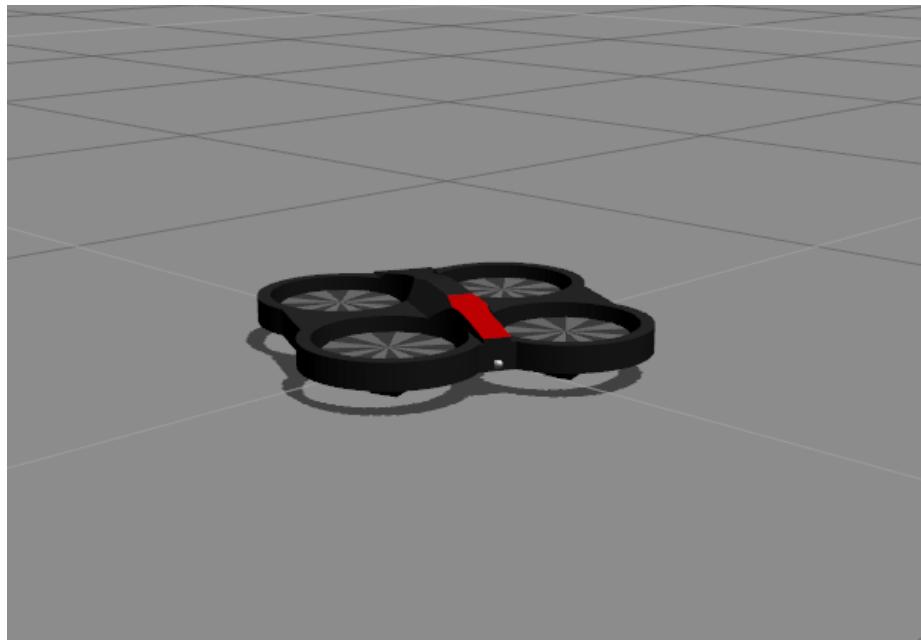


Figura 1.7: AR Drone Simulado en Gazebo

Jorge Cano construyó propio su drone utilizando como base un quadricóptero con un procesador Intel Compute Stick (ICS) STCK1A8LFC como ordenador de abordo y una placa Pixhawk como placa estabilizadora/piloto automático[4]. Esta placa utiliza como protocolo de comunicación MAVLink¹⁶ y dispone de acelerómetros, giróscopos y magnetómetros para determinar la actitud. Además de la construcción del drone Jorge desarrolló el driver MAVLinkServer donde apoyándose en MAVProxy adaptó los comandos MAVLink a interfaces JdeRobot permitiendo acceder a la actitud y controlar el drone con comandos tipo GotoXY enviados a través del interfaz Pose3D¹⁷

En la actualidad se están desarrollando nuevos trabajos sobre este tipo de placas estabilizadoras que tiene como software base ArduCopter/Ardupilot o son compatibles. Por un lado Diego Jiménez[7] trabaja en controlar un Solo Drone de la empresa 3DR¹⁸ mediante el interfaz

¹⁶Micro Air Vehicle Link <https://en.wikipedia.org/wiki/MAVLink>

¹⁷Ver capítulo 3 Arquitectura utilizada

¹⁸Empresa norteamericana con sede en California especializada en robótica aérea. Se sitúa en 2017 como la 3^a empresa del sector



de velocidades CMDVel¹⁹, este quadracótero tiene como placa de control una Pixhawk como la utilizada por Jorge Vela en el drone que se construyó y utiliza también comandos MAVLink.

Por otro lado Jorge Vela se encuentra desarrollando cómo realizar la maniobra de aterrizaje de forma automática al localizar un patrón o baliza localizado visión[11]. Como drone de referencia está utilizando un Solo Drone con una Intel Stick como ordenador de abordo.



Figura 1.8: SOLO Drone de 3D Robotics

Crear clientes web para las principales aplicaciones JdeRobot fue tarea de Aitor Martínez[8].

¹⁹Ver capítulo 3 Arquitectura utilizada

En desarrolló 6 clientes: CameraViewJS, que recibe datos de CameraServer RgbdViewerJS, que recibe datos de Openni1Server KobukiViewerJS, que permite teleoperar robots Kobuki Uav-ViewerJS: Creación del cliente web similar a la herramienta UavViewer para teleoperar drones tanto reales como simulados y ver los datos de sus sensores. IntrorobKobukiJS, que permite programar comportamientos como introrob en una interfaz Web IntrorobUavJS, permite programar comportamientos autónomos en una en una interfaz Web similar a UAV Viewer Gracias a éstos desarrollos, se podía teleoperar un drone desde cualquier dispositivo que tenga un navegador web, como un móvil o una tablet.



Figura 1.9: Drone teleoperado con teléfono móvil

En este contexto este PFC aborda el soporte de un nuevo robot áereo, diferentes a todos los usados en el Laboratorio de Robótica hasta la fecha: un avión de ala fija. Además se integrará con las herramientas ya existentes para robots aéreos dentro de la plataforma JdeRobot.

La memoria de este PFC se ha vertebrado en 7 capítulos. En el segundo se fijan los objetivos concretos que se persiguen y los requisitos marcados. A continuación se describe la infraestructura, tanto hardware como software, utilizada en el desarrollo de este trabajo. En el capítulo 4 se detalla el desarrollo del driver APM Server y se desgrana la aplicación UAV Commander en el capítulo 5. En el sexto se muestran los experimentos realizados y en el séptimo las conclusiones alcanzadas.

Capítulo 2

Objetivos

2.1. Problema a abordar

Los objetivos de este proyecto final de carrera es dar un soporte en la plataforma software JdeRobot para drones de ala fija que utilicen como interfaz de comunicación MAVLink. Para abordar el problema lo hemos dividido en 5 grandes Milestones:

1. Preparación del hardware necesario para abordar el problema con un avión real.
2. Desarrollo de un driver que acceda a los sensores y actuadores de drones de ala fija que utilicen MAVLink y dar soporte a la actuación de misiones que hasta ahora no soportaba JdeRobot el driver se llamará APM Server siglas de Ardupilot Mega Server.
3. Desarrollo de una aplicación GCS Ground Control Station que permita al operador introducir misiones y seguir el cumplimiento de las mismas a través de él. Permitiendo el acceso a toda la actitud y a las cámaras de abordo. Se llamará UAV Commander.
4. Experimentos en simulación. Conectaremos nuestro driver APM Server al simulador SITL y el UAV Commander al APM Server para poder simular el seguimiento de misiones.
5. Experimentos en el avión real. Montaremos la aviónica en el avión real y lo llevaremos al campo de vuelo para validar su comportamiento.

2.2. Requisitos

Para abordar con éxito los milestones expuestos anteriormente debemos cubrir los siguientes requisitos:

1. Preparación del hardware necesario.

a) Compra del hardware necesario:

- Raspberry Pi, se empezó por la Raspberry 2 y se ha adquirido posteriormente la 3.
- Cámara Picam.
- Ardupilot Mega junto con GPS.
- Avión de radiocontrol Bix3.
- Kit de motorización más potente.

b) Instalación de Raspbian y JdeRobot en Raspberry Pi.

c) Pruebas de vídeo en Raspberry Pi con cameraserver.

2. APM Server

a) Dar acceso a los sensores del APM y servir en forma de Pose3D y NavData:

- 1) Conectar con el APM
- 2) Leer los mensajes que envían sus sensores.
- 3) Implementar los interfaces Pose3D y NavData con esta información.
- 4) Servir a través de ICE todos los interfaces.

b) Dar acceso a instrucciones de actuación a través del interfaz mission.

- 1) Recibir a través de ICE un objeto de misión.
- 2) Construir los comandos MAVLink necesarios para que el APM sepa interpretar y realizar la misión.
- 3) Recibir los comandos de despegue y aterrizaje a través del interfaz Extra.
- 4) Construir los comandos MAVLink necesarios para que el APM sepa interpretarlos y añadirlos a la misión.

3. UAV Commander

a) Recibir toda la información sensorial a partir de los interfaces ICE Pose3D y NavData.

b) Funcionalidad de creación de misiones:

- 1) Recuperar mapa georeferenciado de un servicio WMS.
- 2) Capacidad de actuar sobre él para añadir puntos de misión.
- 3) Añadir botones de despegue y aterrizaje que interactúen con el mapa.
- 4) Borrado de misión actual.
- 5) Envío de misión al APM.

c) Funcionalidad de seguimiento de misiones:

- 1) Pintar estela del avión en el mapa.
- 2) Recuperar nuevo mapa con mayor zoom si se prevé la salida del actual mapa.

d) Acceso a la información de actitud de forma visual.

e) Acceso a las cámaras de abordo.

4. Experimentos en simulación.

a) Montaje de máquina virtual para el simulador

b) Instalación de SITL.

c) Conectar APM Server a SITL.

d) Ejecución de plan de pruebas simuladas:

- 1) Prueba de integración de todo el software.
- 2) Prueba envío de misión con avión en vuelo.
- 3) Prueba de envío de misión con despegue y varios waypoints.
- 4) Prueba de envío de misión con despegue y aterrizaje.
- 5) Prueba de autozoom forzando la salida del avión del mapa.

5. Experimentos en el avión real.

a) Pruebas de envío recepción del sistema de radiofrecuencias.

- b) Pruebas de conexión con APM Server y UAV Commander.
- c) Ejecución de plan de pruebas:
 - 1) Pruebas de seguimiento del avión a través del UAV Commander.
 - 2) Prueba de envío de misión desde el avión en vuelo.

Cómo requisitos no funcionales debemos:

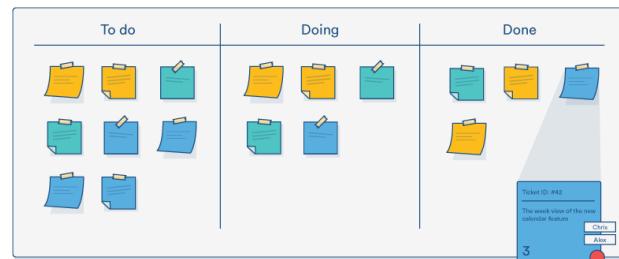
1. Ser multiplataforma.
2. Utilizar únicamente librerías de software libre.
3. Ser 100 % compatibles con los actuales interfaces JdeRobot.

2.3. Metodología y plan de trabajo

Este proyecto se ha abordado en con 2 metodologías de trabajo distintas que combinadas, la primera metodología de desarrolló aplicada es el desarrollo en espiral que se ha utilizado como metodología principal, y la segunda que se ha aplicado la metodología ágil Kanban, para un control de tareas y subtareas más eficiente.



(a) Representación gráfica del desarrollo en espiral.



(b) Representación de una pizarra kanban

El modelo de desarrollo en espiral define una serie de ciclos que se repiten en un bucle hasta el final del proyecto, dividiéndolo en varias subtareas más sencillas y estableciendo puntos de control al final de cada iteración en los que se evalúa el trabajo realizado y se enfocan las nuevas tareas para continuar. Esta metodología recibe su nombre por la forma de espiral que tiene su representación gráfica o diagrama de flujo, que podemos ver en la figura 2.1. En cada iteración se llevan a cabo las siguientes actividades: Determinar los objetivos, dividir en subobjetivos y fijar

requisitos. Analizar los riesgos y factores que impidan o dificulten el trabajo y las consecuencias negativas que este pueda ocasionar.

Para desarrollar las tareas especificadas en cada incremento, fijadas en cada reunión de seguimiento, se utilizó la metodología kanban. Esta metodología de desarrollo no es más que una adaptación de su versión industrial que surgió en Toyota. A finales de los años 40, Toyota empezó a optimizar sus procesos de ingeniería a partir del modelo que empleaban los supermercados para llenar los estantes. Los supermercados almacenan los productos suficientes para suprir la demanda del cliente, una práctica que optimiza el flujo entre el supermercado y el cliente. En su versión TIC ésta metodología se centra, al igual que su versión industrial, en el just in time y permite ver de una forma muy visual las tareas que hay en vuelo, desarrolladas, pendientes o bloqueadas pudiendo anticiparnos a cuellos de botella o bloqueos de forma sencilla. Esta metodología nos permite visualizar tareas que a priori no tienen relación pero se relacionan a un nivel más profundo y agruparlas para optimizar el tiempo.



Figura 2.1: Evolución de nuestra

Durante el ciclo de vida del proyecto se han llevado a cabo reuniones semanales de seguimiento.

miento con el tutor. En ellas se evaluaban las tareas realizadas y se marcaba qué dirección tomar para la siguiente iteración o incremento. Si los puntos marcados en la anterior reunión no se habían alcanzado se ampliaba el plazo o se discutían otras vías para avanzar. En caso contrario se proponían nuevos subobjetivos.

Para apoyarnos en nuestro desarrollo hemos utilizado principalmente 4 herramientas:

- GitHub como forja y control de versiones. En el repositorio <https://github.com/RoboticsURJC-students/2014-pfc-JoseAntonio-Fernandez> se almacenan todos los desarrollos que son objetivo de éste PFC así como ésta memoria. También se encuentran subproductos de desarrollo que han ido surgiendo como apoyo o pruebas a los desarrollos principales.
- Contamos también con un mediawiki en JdeRobot dónde hemos actualizado periódicamente nuestros avances acompañados con explicaciones, vídeos e imágenes. Aquí se puede contemplar con más detalle la construcción del UAV.
- Como apoyo principalmente al mediawiki dispusimos de una carpeta en el FTP de JdeRobot hasta que colgamos los videos en Youtube.
- Youtube. Muchos de los vídeos del mediawiki han sido compartidos en youtube.com.

Capítulo 3

Infraestructura utilizada

Se describen en este capítulo las herramientas, ingredientes hardware y software en las que se apoya este PFC.

3.1. Hardware

Este PFC utiliza principalmente tres componentes hardware: un avión de radio control, un procesador Raspberry PI 3 y una placa estabilizadora.

Como el avión de radio-control hemos elegido el avión Bix3 distribuido por la empresa china www.hobbyking.com. Hemos elegido este modelo por ser un avión muy estable debido principalmente a sus cualidades como velero y su alta superficie alar. Otro aspecto importante de la construcción del avión es que el propulsor no se encuentra en el frontal del avión lo que nos permitirá aprovechar al máximo esa zona pudiendo poner incluso una cámara frontal.



(a) Bix3

(b) Vista de cerca

(c) Bix3 en vuelo

Para poder cargar con el equipo necesario a bordo ha sido necesario cambiar el motor, el variador y las baterías de serie por otras de mayor rendimiento que nos permitan volar con tanta

carga de pago.

Una raspberry PI3 que es el ordenador de abordo y en el que se ha instalado la infraestructura necesaria para dotarle de inteligencia. Hemos elegido éste dispositivo debido al compromiso peso/potencia que otorga, así como porque es de un hardware muy asequible y extendido. Conectado a esta PI3 va una PiCam que nos permitira ver lo que el avión vea.

Para este PFC hemos utilizado una placa Ardupilot Mega¹ como estabilizador/piloto automático. Este dispositivo tiene como base una placa Arduino Mega a la que se le han incorporado giróscopos y acelerómetros en 3 ejes para su estabilización y que trae de serie un receptor GPS con brújula. El kit que adquirimos incluye también todo lo necesario para integrarlo en nuestro avión y proporcionarle la alimentación necesaria. Esta placa ataca directamente sobre los actuadores del avión (los servos y el motor) a través de señales PWM.

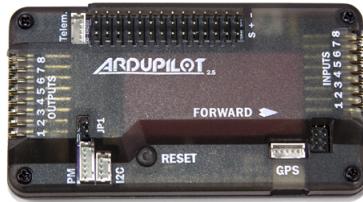


Figura 3.1: Placa estabilizadora ardupilot

Esta placa ha de *flashear*se con un *firmware* en función del tipo de aeronave que utilicemos como base, Arducopter para alas rotatorias o Ardupilot para alas fijas. Esta y otras placas similares como PixHawk, se comunican y aceptan comandos a través de un protocolo llamado MAVLink. A través de comandos MAVLink se puede acceder a los sensores, a los que incorpora la placa o a los que le añadamos a la misma, como el GPS o sensores de velocidad del aire. A través de estos comandos se le puede también enviar órdenes al piloto automático, quien las ejecutará. Más adelante trataremos el protocolo MAVLink en profundidad. Cabe destacar que algunos de los comandos que mandemos tendrán distinto comportamiento en la placa en función del *firmware* escogido, y que hay un conjunto de comandos que no pueden ser utilizados en uno o en otro. El objetivo de este PFC es crear un driver que nos permita comunicarnos con esta placa y una aplicación en JdeRobot para el manejo de un avión robotizado.

¹<http://www.ardupilot.co.uk/>

3.2. JdeRobot

JdeRobot es un entorno de software creado por el laboratorio de robótica de la Universidad Rey Juan Carlos, para el desarrollo de aplicaciones de robótica. Su última versión, la 5.5, se liberó el 15 de Marzo de 2017 pudiendo ver los detalles de ésta en el github oficial². JdeRobot se compone de interfaces, drivers, utilidades y aplicaciones para el desarrollo de cualquier proyecto de robótica. Se apoya en estos interfaces para interconectar entre sí todos los componentes de una aplicación robótica y en Zeroc ICE para la comunicación entre ellos. Algunos de los drivers más importantes que contiene y mas relacionado con este trabajo son:

1. **Cameraserver.** Un driver para enviar imágenes y video a través del interfaz `camera`
2. **Gazeboserver.** Driver para conectar las aplicaciones con distintos robots simulados (con ruedas, cuadricópteros, etc.)
3. **MAVLinkServer.** Driver desarrollado para intercomunicar JdeRobot con placas de estabilización y autopiloto que utilicen el protocolo de comunicación MAVLink.
4. **Ardrone_server.** Driver que conecta el Parrot Ar-Drone a JdeRobot. Este driver escrito en c++ transforma el conjunto de comandos AT del drone en interfaces ICE y viceversa. Implementa los interfaces `camera`, `cmdvel`, `navdata`, `extra` y `pose3D` y permite acceder a la actitud del drone así como a sus 2 cámaras. Sirve también datos como el nivel de la batería y permite grabar vídeo o tomar fotos.

Algunas de las herramientas contenidas en JdeRobot y más relacionadas con este PFC son:

1. **Cameraview.** Se trata de una aplicación desarrollada en c++ capaz de recibir vídeo a través del interfaz `camera`.
2. **UAV viewer.** Aplicación desarrollada para teleoperar robots aéreos. Esta aplicación permite teleoperar cualquier tipo de robot aéreo y ofrece de forma visualmente atractiva datos como la actitud, velocidades lineales y angulares, ofrece también la posibilidad de visualizar imágenes servidas por el interfaz `camera`.

²<https://github.com/JdeRobot/JdeRobot/wiki/JdeRobot-5.5.0>

3.2.1. Interfaces

JdeRobot expone más de 30 interfaces pero en este capítulo explicaremos los que durante nuestro desarrollo hemos implementado en el driver para el avión o usado en la aplicación:

- **Pose3D.** Utilizado para recoger los datos de actitud y la posición de la aeronave. Devuelve la posición del robot y su orientación en el espacio, empleando para la orientación cuaterniones. Comparados con los ángulos de Euler, son más simples de componer y evitan el problema del bloqueo del cardán³, pymavlink nos permite transformar de uno a otro de forma sencilla.

```
Pose3DData
{
    float x; //latitude
    float y; //longitude
    float z; //altitude
    float h; //not used now
    float q0; //quaternion component 1
    float q1; //quaternion component 2
    float q2; //quaternion component 3
    float q3; //quaternion component 4
};
```

- **Camera.** Utilizado para servir imágenes.

```
class CameraDescription
{
    string name;
    string shortDescription;
    string streamingUri;
    float fdistx; //focal distance in x
    float fdisty; //focal distance in y
    float u0;
    float v0;
    float skew;
    float posx; //position in x of the camera
    float posy; //position in y of the camera
    float posz; //position in z of the camera
    float foax; //position of a focused object
    float foay; //position of a focused object
    float foaz; //position of a focused object
    float roll; //roll of the camera
};
```

³El bloqueo del cardán consiste en la pérdida de un grado de libertad en una suspensión cardán de tres motores, que ocurre cuando los ejes de dos de los tres motores se colocan en paralelo, bloqueando el sistema en una rotación en un espacio bidimensional degenerado.

Los parámetros del interfaz se pueden dividir en 2 bloques: intrínsecos y extrínsecos.

- Intrínsecos.
 - fdistx
 - fdisty
 - u0
 - v0
 - skew
- Extrínsecos.
 - posx
 - posy
 - posz
 - foax
 - foay
 - foaz
 - roll
- NavData. Utilizado para servir datos secundarios de actuación como velocidades lineales o angulares o el estado de la batería.

```
class NavdataData
{
    int vehicle; //0-> ArDrone1, 1-> ArDrone2
    int state; // landed, flying, ...
    float batteryPercent; //The remaing charge of baterry %

    //Magnetometer Ardrone2.0
    int magX;
    int magY;
    int magZ;

    int pressure; //Barometer Ardrone2.0
    int temp; //Temperature sensor Ardrone2.0
    float windSpeed; //Estimated wind speed Ardrone2.0

    float windAngle;
    float windCompAngle;
```

```

float rotX; //rotation about the X axis
float rotY; //rotation about the Y axis
float rotZ; //rotation about the Z axis

int altd; //Estimated altitude (mm)

//linear velocities (mm/sec)
float vx;
float vy;
float vz;

//linear accelerations (unit: g) ;Ardrone2.0?
float ax;
float ay;
float az;

//Tags in Vision Detectoion
//Should be unsigned
int tagsCount;
arrayInt tagsType;
arrayInt tagsXc;
arrayInt tagsYc;
arrayInt tagsWidth;
arrayInt tagsHeight;
arrayFloat tagsOrientation;
arrayFloat tagsDistance;

float tm; //time stamp
};

```

- Extra. Utilizado principalmente para las órdenes de despegue y aterrizaje.

```

void land() - land drone.
void takeoff() - takeoff drone.
void reset()
void recordOnUsb(bool record)
void ledAnimation(int type, float duration, float req)
void flightAnimation(int type, float duration)
void flatTrim()
void toggleCam() - switch camera.

```

3.3. Protocolo MAVLink

MAVLink siglas de Micro Air Vehicle Link es un protocolo de comunicación desarrollado para comunicar las placas estabilizadoras dotadas de piloto automático con los GCS o *Ground*

control station, es decir, con las aplicaciones desde las que se podía enviar misiones y seguir el cumplimiento de las mismas desde tierra. MAVLink se publicó en 2009 por Lorenz Meier, publicado bajo licencia LGPL. Aspira a convertirse en el protocolo standard en robótica aérea y se ha probado su funcionamiento en PX4, PIXHAWK, APM⁴ y Parrot AR.Drone.

Un ejemplo de comando MAVLink sería este mensaje trae la información del GPS y se envía periódicamente en ciclos que decidimos en parámetros de conexión con el dispositivo.

```
type GpsStatus struct {
    SatellitesVisible uint8      Número de satélites visibles
    SatellitePrn      [20]uint8   Id Global de cada satélite
    SatelliteUsed     [20]uint8   Lista con el uso de cada sat\'elite
    SatelliteElevation [20]uint8  Elevación, nos da el ángulo sobre el horizonte.
    SatelliteAzimuth  [20]uint8  Dirección del satélite, 0: 0 grados, 255: 360 grados.
    SatelliteSnr      [20]uint8  Señal/ruido de cada uno de los satélites
}
```

Otro mensaje, esta vez vinculado a la actuación sería:

```
type MissionItem struct {
    Param1          float32    parámetro variable en función del comando.
    Param2          float32    parámetro variable en función del comando.
    Param3          float32    parámetro variable en función del comando.
    Param4          float32    parámetro variable en función del comando.
    X               float32    latitud
    Y               float32    longitud
    Z               float32    altitud
    Seq             uint16    Número del item en la misión
    Command         uint16    Tipo de comando de navegación.
    TargetSystem    uint8     ID del sistema
    TargetComponent uint8
    Frame           uint8     Sistema de coordenadas que se utiliza.
    Current         uint8     Misión actual no:0, si:1
    Autocontinue    uint8     Autocontinuar al siguiente objeto de misión.
}
```

Este mensaje representa un item de misión, una misión se compone de uno o varios de estos items. Un item de misión en MAVLink puede ser un punto de paso o *waypoint*, una maniobra de despegue o aterrizaje, o instrucciones de cambio de altura o velocidad del robot.

3.4. Biblioteca pymavlink

Pymavlink trae la definición de los comandos a utilizar en python. Gracias a esta biblioteca no se tienen que construir los comandos MAVLink a mano, evitando cometer errores y sim-

⁴Ardupilot Mega

plificando la creación y envío de los mismos. Su uso facilita también la identificación de los comandos recibidos. Para instalarla tan sólo se necesita ejecutar:

```
sudo pip2 install -U pymavlink
```

Y tiene como dependencias:

- future. Se necesita future como soporte en Python 3.X de Python 2.7 (<http://python-future.org/>)
- lxml. Para parsear xml (<http://lxml.de/installation.html>)
- python-dev
- MavLink. (<http://qgroundcontrol.org/mavlink/start>)

3.5. Lenguaje Python y biblioteca PyQt5

Python es un lenguaje de programación interpretado y multiplataforma que nació en los años 80 en los países bajos con idea de hacer más legible el código. El lenguaje de programación que inicialmente se utilizaba principalmente para scripting, ha sabido crecer con los años y con la publicación de Python3 en 2009 ha recibido el impulso que necesitaba para ser hoy en día el quinto lenguaje más utilizado, por encima de PHP, .NET y JavaScript (que baja hasta el 8º puesto según TIOBE⁵ en un estudio de Abril de 2017).

Para este PFC se eligió Python, porque mantiene el carácter multiplataforma de JdeRobot, su código es simple y legible y trabaja muy bien con dependencias muy utilizadas en robótica como OpenCV.

Para nuestro desarrollo hemos utilizado PyQt5 para desarrollar el interfaz gráfico. PyQt5 es un *binding* de Qt5 en forma de librería Python que nos permite acceder a toda la funcionalidad de Qt5. Qt, propiedad de Nokia, es un conjunto de librerías escritas en C++ para interfaces gráficas.

⁵<https://www.tiobe.com/tiobe-index/>



Figura 3.2: Imagen geo-referenciada del club de aeromodelismo Icaro

3.6. Mapas y Geo-referenciación

En la aplicación de navegación autónoma hemos tenido que trabajar con mapas geo-referenciados. Estos mapas, que tanto se han popularizado gracias a Google, son una herramienta imprescindible en robótica aérea si se quiere trabajar con largas distancias. Un mapa geo-referenciado es aquel en el que conocemos o podemos calcular la posición en el planeta que representa cada píxel del mismo.

La forma más común de obtener estos mapas geo-referenciados es a través de WMS, siglas de *Web Map Service*. Un WMS no es más que un servicio web que recibe como entrada unas coordenadas y una serie de parámetros y devuelve una imagen encuadrada en los datos enviados.

En nuestro desarrollo hemos utilizado dos WMS: el del IGN⁶ y el de Google. A continuación vamos a desgranar un WMS, el de PNOA del IGN. PNOA son las siglas de Plan Nacional de Ortofotografía Aérea y contiene los mapas más actuales del territorio español. El WMS de PNOA requiere como entrada cuatro atributos principales:

⁶Instituto Geográfico Nacional de España

1. La posición GPS.
2. *bounding box*. El *bounding box* son dos puntos que corresponden con la posición GPS que queremos que sea el extremo inferior izquierdo de nuestra imagen y con el punto superior derecho de la misma.
3. Datum. El datum es el sistema de referencia o proyección de la tierra a utilizar. En el caso de IGN pese a que soporta varias utilizamos WGS84 que es el estandar.
4. Tamaño de la imagen, el tamaño que queremos que tenga la imagen obtenida.

El resultado de la consulta es un mapa del que conocemos el punto central, y los extremos inferior izquierdo y superior derecho y los píxeles que tiene a lo ancho y alto del mismo, es decir, un mapa geo-referenciado.

Con el fin de facilitar el montaje y envío de las peticiones al WMS de PNOA utlizamos la librería ⁷ en su versión 0.14.0.

3.7. Simulador SITL

SITL⁸ siglas de *Software In The Loop* es un simulador que permite a cualquier programa que envíe o reciba comandos MAVLink ejecutar pruebas sin necesidad de tener ninguna placa estabilizadora real y evitando la pérdida de la aeronave en caso de error del programa. SITL se conecta con JSBSim, un simulador de vuelo de software libre para ejecutar el aspecto físico de la simulación y con MAVProxy para el envío de comandos y seguimiento de misiones. Se trata de una compilación en C++ de ardupilot y se podría asemejar al driver para cuadricópteros existente en JdeRobot gazeboserver, pero para aviones de ala fija. SITL puede conectarse también con Gazebo o a FlightGear para hacer más completa su simulación mostrando el vuelo en un entorno desarrollado para pruebas con elementos del mundo real.

⁷<https://geopython.github.io/OWSLib/>

⁸<http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html>

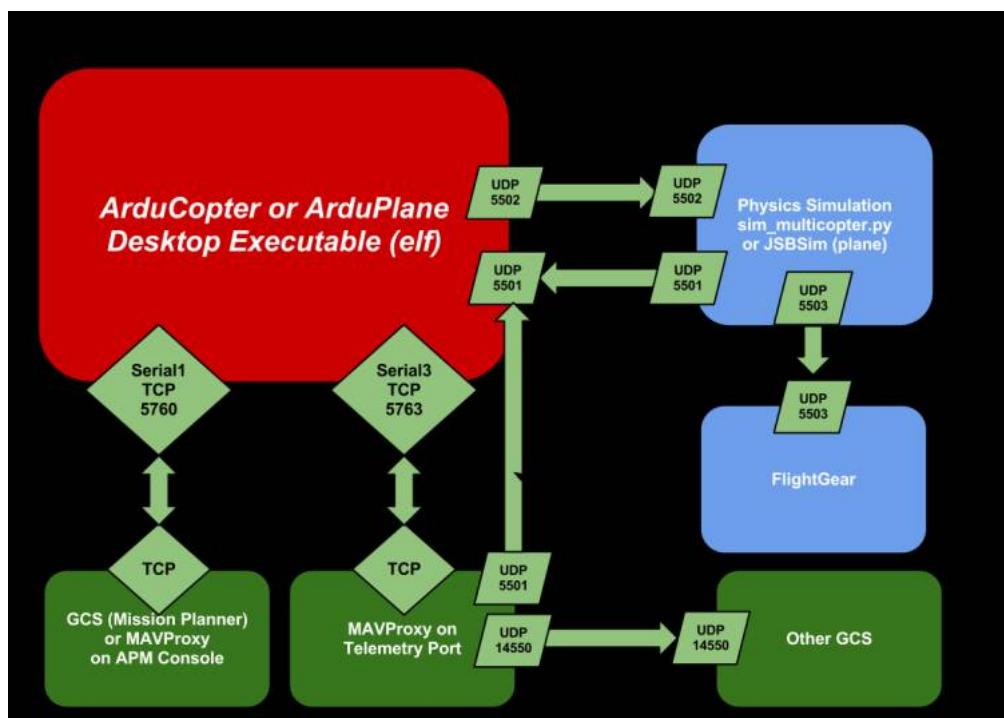


Figura 3.3: Arquitectura de SITL

Capítulo 4

APM Server

El driver APM Server es quien va a mediar entre las aplicaciones de JdeRobot y el robot aéreo con sus sensores y actuadores físicos o simulados a través de SITL. De esta manera las aplicaciones pueden correr en máquinas distintas, no obligatoriamente a bordo, y pueden estar implementadas en distintos lenguajes de programación. Estas ventajas vienen de utilizar la división habitual en JdeRobot entre componentes drivers y componentes aplicación.

En nuestro caso este driver se ejecuta sobre una Raspberry Pi3 con raspbian a la que hemos instalado el entorno JdeRobot y el driver APM Server, e irá abordo del robot físico conectado por puerto serie.

4.1. Diseño

Los requerimientos hablados en el capítulo 2 son a grandes rasgos:

- El driver debe ser capaz de conectar con dispositivos físicos de estabilización como APM 2.8 o PixHawnk así como al simulador SITL
- El driver debe acceder a los sensores y actuadores del robot aéreo, interpretarlos y servirlos en forma de interfaz ICE a las aplicaciones de control.
- EL driver debe ser capaz de recibir a través de interfaces JdeRobot interfaces que éste interprete y envíe a los actuadores del robot aéreo.

Se ha diseñado y programado un componente de JdeRobot en Python que hace de driver

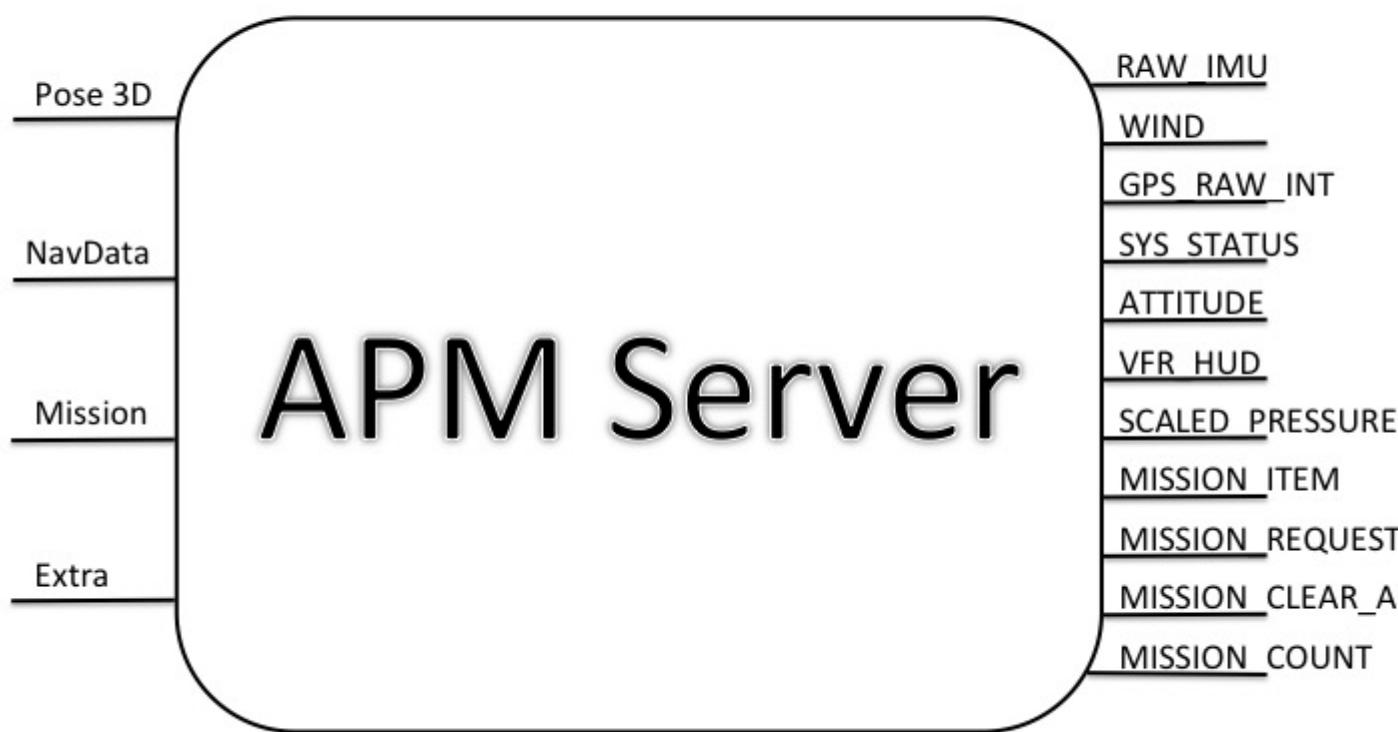


Figura 4.1: Diseño de APM Server

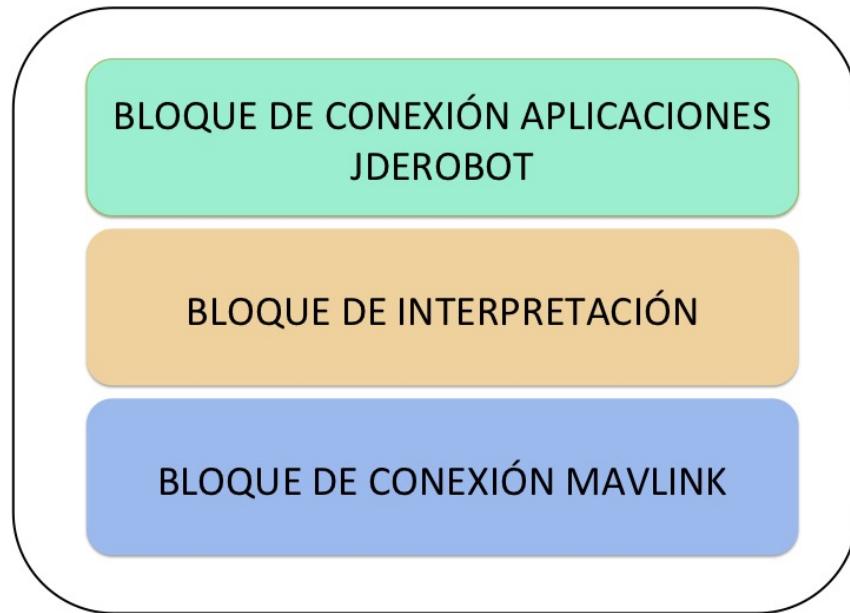


Figura 4.2: Diseño de APM Server

y que satisface esos requerimientos. Se ha organizado en tres capas desde la más cercana al hardware hasta la más cercana a JdeRobot:

- Capa de comunicación con el dispositivo APM. Esta capa se encarga de la comunicación del driver con el dispositivo APM a través del protocolo de comunicación MavLink.
- Capa de negocio e interpretación. En esta capa el driver transforma los comandos MavLink y los interfaces JdeRobot interpretando la información de ambas capas y haciendo ésta legible en ambos sentidos.
- Capa de comunicación con aplicaciones JdeRobot. En ésta capa se materializan los servicios ICE necesarios para la recepción y envío de los interfaces ICE que implementa.

4.2. Bloque de conexión MAVLink

En la capa de comunicación con el APM es dónde se van a recibir comandos MAVLink desde la placa estabilizadora y autopiloto y se van a consumir y enviar los siguientes mensajes. Para cada uno de ellos se detalla su nombre, su sintaxis y su semántica.

4.2.1. Listado de mensajes MAVLink

- RAW_IMU donde recibimos la información de los acelerómetros giróscopos y magnetómetros en crudo par el NavData.

```
<message id="27" name="RAW_IMU">
    <description>The RAW IMU readings for the usual 9DOF sensor setup.
        This message should always contain the true raw values without any
        scaling to allow data capture and system debugging.</description>
    <field type="uint64_t" name="time_usec" units="us">Timestamp
        (microseconds since UNIX epoch or microseconds since system boot)</field>
    <field type="int16_t" name="xacc">X acceleration (raw)</field>
    <field type="int16_t" name="yacc">Y acceleration (raw)</field>
    <field type="int16_t" name="zacc">Z acceleration (raw)</field>
    <field type="int16_t" name="xgyro">Angular speed around X axis (raw)</field>
    <field type="int16_t" name="ygyro">Angular speed around Y axis (raw)</field>
    <field type="int16_t" name="zgyro">Angular speed around Z axis (raw)</field>
    <field type="int16_t" name="xmag">X Magnetic field (raw)</field>
    <field type="int16_t" name="ymag">Y Magnetic field (raw)</field>
    <field type="int16_t" name="zmag">Z Magnetic field (raw)</field>
</message>
```

Un ejemplo de mensaje recibido sería: RAW_IMU {time_usec : 480794000, xacc : 244, yacc : 0, zacc : -968, xgyro : 1, ygyro : 1, zgyro : 1, xmag : 335, ymag : 60, zmag : -452}

- VFR_HUD Dónde se presentan los típicos datos de navegación en ala fija como, la velocidad en el aire, velocidad en tierra o la velocidad de ascenso. De aquí obtendremos la altitud

```
<message id="74" name="VFR_HUD">
    <description>Metrics typically displayed on a HUD for fixed wing aircraft</description>
    <field type="float" name="airspeed" units="m/s">Current airspeed in m/s</field>
    <field type="float" name="groundspeed" units="m/s">Current ground speed in m/s</field>
    <field type="int16_t" name="heading" units="deg">Current heading in degrees,
        in compass units (0..360, 0=north)</field>
    <field type="uint16_t" name="throttle" units="%">Current throttle setting in
        integer percent, 0 to 100</field>
    <field type="float" name="alt" units="m">Current altitude (MSL), in meters</field>
    <field type="float" name="climb" units="m/s">Current climb rate in meters/second</field>
</message>
```

Un ejemplo de mensaje recibido sería: VFR_HUD {airspeed : 0.022061701864004135, groundspeed : 0.08534427732229233, heading : 356, throttle : 0, alt : 584.1099853515625, climb : -0.33149993419647217}

- ATTITUDE donde recibimos la actitud del avión.

```

<message id="30" name="ATTITUDE">
  <description>The attitude in the aeronautical frame
    (right-handed, Z-down, X-front, Y-right).</description>
  <field type="uint32_t" name="time_boot_ms" units="ms">Timestamp
    (milliseconds since system boot)</field>
  <field type="float" name="roll" units="rad">Roll angle (rad, -pi..+pi)</field>
  <field type="float" name="pitch" units="rad">Pitch angle (rad, -pi..+pi)</field>
  <field type="float" name="yaw" units="rad">Yaw angle (rad, -pi..+pi)</field>
  <field type="float" name="rollspeed" units="rad/s">Roll angular speed (rad/s)</field>
  <field type="float" name="pitchspeed" units="rad/s">Pitch angular speed (rad/s)</field>
  <field type="float" name="yawspeed" units="rad/s">Yaw angular speed (rad/s)</field>
</message>

```

Un ejemplo de mensaje recibido sería: ATTITUDE {time_boot_ms : 480794, roll : -0.005003694910556078, pitch : 0.2430807501077652, yaw : -0.06844368577003479, rollspeed : -0.0010073177982121706, pitchspeed : -0.0008516315137967467, yawspeed : -0.0006909647490829229}

- SYS_STATUS donde se obtiene el nivel de batería.

```

<message id="1" name="SYS_STATUS">
  <description>The general system state. If the system is following the MAVLink standard,
    the system state is mainly defined by three orthogonal states/modes: The system mode,
    which is either LOCKED (motors shut down and locked), MANUAL (system under RC control),
    GUIDED (system with autonomous position control, position setpoint controlled manually)
    or AUTO (system guided by path/waypoint planner).

    The NAV_MODE defined the current flight state:
      LIFTOFF (often an open-loop maneuver),
      LANDING
      WAYPOINTS
      VECTOR.

    This represents the internal navigation state machine. The system status shows wether
    the system is currently active or not and if an emergency occured.

    During the CRITICAL and EMERGENCY states the MAV is still considered to be active,
    but should start emergency procedures autonomously. After a failure occured it should
    first move from active to critical to allow manual intervention and then move to
    emergency after a certain timeout.</description>
  <field type="uint32_t" name="onboard_control_sensors_present" enum="MAV_SYS_STATUS_SENSOR"
    display="bitmask" print_format="0x%04x">
    Bitmask showing which onboard controllers and sensors are present. Value of 0: not
    present. Value of 1: present. Indices defined by ENUM MAV_SYS_STATUS_SENSOR</field>
  <field type="uint32_t" name="onboard_control_sensors_enabled" enum="MAV_SYS_STATUS_SENSOR"
    display="bitmask" print_format="0x%04x">
    Bitmask showing which onboard controllers and sensors are enabled: Value of 0:
    not enabled. Value of 1: enabled. Indices defined by ENUM MAV_SYS_STATUS_SENSOR</field>
  <field type="uint32_t" name="onboard_control_sensors_health" enum="MAV_SYS_STATUS_SENSOR">

```

```

display="bitmask" print_format="0x%04x">
Bitmask showing which onboard controllers and sensors are operational or have an error:
Value of 0: not enabled. Value of 1: enabled.
Indices defined by ENUM MAV_SYS_STATUS_SENSOR</field>
<field type="uint16_t" name="load" units="d%">Maximum usage in percent of the mainloop
time, (0%: 0, 100%: 1000) should be always below 1000</field>
<field type="uint16_t" name="voltage_battery" units="mV">Battery voltage, in millivolts
(1 = 1 millivolt)</field>
<field type="int16_t" name="current_battery" units="cA">Battery current, in
10*milliamperes (1 = 10 milliampere), -1: autopilot does not measure the current</field>
<field type="int8_t" name="battery_remaining" units="%">Remaining battery energy:
(0%: 0, 100%: 100), -1: autopilot estimate the remaining battery</field>
<field type="uint16_t" name="drop_rate_comm" units="c%">Communication drops in percent,
(0%: 0, 100%: 10'000), (UART, I2C, SPI, CAN), dropped packets on all links
(packets that were corrupted on reception on the MAV)</field>
<field type="uint16_t" name="errors_comm">Communication errors (UART, I2C, SPI, CAN),
dropped packets on all links (packets that were corrupted on reception on the MAV)</field>
<field type="uint16_t" name="errors_count1">Autopilot-specific errors</field>
<field type="uint16_t" name="errors_count2">Autopilot-specific errors</field>
<field type="uint16_t" name="errors_count3">Autopilot-specific errors</field>
<field type="uint16_t" name="errors_count4">Autopilot-specific errors</field>
</message>

```

Un mensaje recibido sería: SYS_STATUS {onboard_control_sensors_present : 23198783, onboard_control_sensors_enabled : 23198783, onboard_control_sensors_health : 24247359, load : 0, voltage_battery : 12587, current_battery : 0, battery_remaining : 100, drop_rate_comm : 0, errors_comm : 0, errors_count1 : 0, errors_count2 : 0, errors_count3 : 0, errors_count4 : 0}

- SCALED_PRESSURE donde obtendremos la presión absoluta y la temperatura.

```

<message id="29" name="SCALED_PRESSURE">
<description>The pressure readings for the typical setup of one absolute and differential
pressure sensor. The units are as specified in each field.</description>
<field type="uint32_t" name="time_boot_ms" units="ms">Timestamp
(milliseconds since system boot)</field>
<field type="float" name="press_abs" units="hPa">Absolute pressure (hectopascal)</field>
<field type="float" name="press_diff" units="hPa">Differential pressure 1 </field>
<field type="int16_t" name="temperature" units="cdegC">Temperature measurement
(0.01 degrees celsius)</field>
</message>

```

Un mensaje recibido sería: SCALED_PRESSURE {time_boot_ms : 480794, press_abs : 945.0001831054688, press_diff : 0.021015625447034836, temperature : 2600}

- WIND donde se obtienen las lecturas del viento estimadas.

```
<message id="168" name="WIND">
  <description>Wind estimation</description>
  <field name="direction" type="float">wind direction that wind is coming from </field>
  <field name="speed" type="float">wind speed in ground plane (m/s)</field>
  <field name="speed_z" type="float">vertical wind speed (m/s)</field>
</message>
```

Un mensaje recibido sería: WIND {direction : -179.99998474121094, speed : 0.0, speed_z : 0.0}

- GLOBAL_POSITION_INT donde obtendremos la posición GPS.

```
<message id="33" name="GLOBAL_POSITION_INT">
  <description>The filtered global position (e.g. fused GPS and accelerometers).  

    The position is in GPS-frame (right-handed, Z-up). It is designed as scaled integer  

    message since the resolution of float is not sufficient.</description>
  <field type="uint32_t" name="time_boot_ms" units="ms">Timestamp  

    (milliseconds since system boot)</field>
  <field type="int32_t" name="lat" units="degE7">Latitude, expressed as degrees * 1E7</field>
  <field type="int32_t" name="lon" units="degE7">Longitude, expressed as degrees * 1E7</field>
  <field type="int32_t" name="alt" units="mm">Altitude in meters, expressed as  

    * 1000 (millimeters), AMSL (not WGS84 – note that virtually all GPS modules provide  

    the AMSL as well)</field>
  <field type="int32_t" name="relative_alt" units="mm">Altitude above ground in meters,  

    expressed as * 1000 (millimeters)</field>
  <field type="int16_t" name="vx" units="cm/s">Ground X Speed (Latitude, positive north),  

    expressed as m/s * 100</field>
  <field type="int16_t" name="vy" units="cm/s">Ground Y Speed (Longitude, positive east),  

    expressed as m/s * 100</field>
  <field type="int16_t" name="vz" units="cm/s">Ground Z Speed (Altitude, positive down),  

    expressed as m/s * 100</field>
  <field type="uint16_t" name="hdg" units="cdeg">Vehicle heading (yaw angle) in degrees  

    * 100, 0.0..359.99 degrees. If unknown, set to: UINT16_MAX</field>
</message>
```

Un mensaje recibido sería: GLOBAL_POSITION_INT {time_boot_ms : 480614, lat : -353632612, lon : 1491652301, alt : 584110, relative_alt : -179, vx : 0, vy : 0, vz : 0, hdg : 35608}

- MISSION_ITEM Son objetos de misión, en ellos se mandan los puntos de paso o comandos como fijar una velocidad o una altitud o bien aterrizar o despegar en función del parámetro command.

```

<message id="39" name="MISSION_ITEM">
  <description>Message encoding a mission item. This message is emitted to announce the
  presence of a mission item and to set a mission item on the system. The mission item
  can be either in x, y, z meters (type: LOCAL) or x:lat, y:lon, z:altitude.
  Local frame is Z-down, right handed (NED), global frame is Z-up, right handed (ENU).
  See also http://qgroundcontrol.org/mavlink/waypoint\_protocol.</description>
  <field type="uint8_t" name="target_system">System ID</field>
  <field type="uint8_t" name="target_component">Component ID</field>
  <field type="uint16_t" name="seq">Sequence</field>
  <field type="uint8_t" name="frame" enum="MAV_FRAME">The coordinate system of the MISSION.
    see MAV_FRAME in mavlink_types.h</field>
  <field type="uint16_t" name="command" enum="MAV_CMD">The scheduled action for the MISSION.
    see MAV_CMD in common.xml MAVLink specs</field>
  <field type="uint8_t" name="current">false:0, true:1</field>
  <field type="uint8_t" name="autocontinue">autocontinue to next wp</field>
  <field type="float" name="param1">PARAM1, see MAV_CMD enum</field>
  <field type="float" name="param2">PARAM2, see MAV_CMD enum</field>
  <field type="float" name="param3">PARAM3, see MAV_CMD enum</field>
  <field type="float" name="param4">PARAM4, see MAV_CMD enum</field>
  <field type="float" name="x">PARAM5 / local: x position, global: latitude</field>
  <field type="float" name="y">PARAM6 / y position: global: longitude</field>
  <field type="float" name="z">PARAM7 / z position: global: altitude (relative or absolute,
    depending on frame.</field>
  <extensions/>
  <field type="uint8_t" name="mission_type" enum="MAV_MISSION_TYPE">Mission type,
    see MAV_MISSION_TYPE</field>
</message>
```

Un mensaje enviado sería: MISSION_ITEM {target_system : 1, target_component : 1, seq : 0, frame : 3, command : 16, current : 0, autocontinue : 0, param1 : 0, param2 : 10, param3 : 0, param4 : 0, x : 40.33024215698242, y : -3.8008816242218018, z : 40.0}

- MISSION_REQUEST mensaje donde el APM nos requiere el siguiente MISSION_ITEM. Este comando se recibe o bien una vez enviado al APM nuestra intención de enviarle mensajes de misión y cuando vamos a enviarle, en un comando MISSION_COUNT o bien tras la recepción de alguno de ellos como ACK y solicitando el siguiente

```

<message id="40" name="MISSION_REQUEST">
  <description>Request the information of the mission item with the sequence number seq.
  The response of the system to this message should be a MISSION_ITEM message.
  http://qgroundcontrol.org/mavlink/waypoint\_protocol</description>
  <field type="uint8_t" name="target_system">System ID</field>
  <field type="uint8_t" name="target_component">Component ID</field>
  <field type="uint16_t" name="seq">Sequence</field>
  <extensions/>
  <field type="uint8_t" name="mission_type" enum="MAV_MISSION_TYPE">Mission type,
```

```
see MAV_MISSION_TYPE</field>
</message>
```

Un mensaje recibido sería: MISSION_REQUEST {target_system : 0, target_component : 0, seq : 2} En este mensaje el APM nos estaría solicitando el item de misión 3, ya que empieza solicitando el 0. Este mensaje termina al llegar al máximo de mission item que se ha indicado que le se le va a mandar.

- MISSION_COUNT éste comando se envía para comenzar una comunicación con el APM con el objetivo de enviarle una misión.

```
<message id="44" name="MISSION_COUNT">
  <description>This message is emitted as response to MISSION_REQUEST_LIST by the MAV
  and to initiate a write transaction. The GCS can then request the individual
  mission item based on the knowledge of the total number of MISSIONs.</description>
  <field type="uint8_t" name="target_system">System ID</field>
  <field type="uint8_t" name="target_component">Component ID</field>
  <field type="uint16_t" name="count">Number of mission items in the sequence</field>
  <extensions/>
  <field type="uint8_t" name="mission_type" enum="MAV_MISSION_TYPE">Mission type,
    see MAV_MISSION_TYPE</field>
</message>
```

Un mensaje enviado sería: MISSION_COUNT {target_system : 0, target_component : 0, count : 3} Donde comunicamos al APM nuestra intención de enviarle 3 objetos de misión.

- MISSION_CLEAR_ALL éste comando sirve para limpiar el APM de misiones, si éste tenía alguna misión previa se desecha.

```
<message id="45" name="MISSION_CLEAR_ALL">
  <description>Delete all mission items at once.</description>
  <field type="uint8_t" name="target_system">System ID</field>
  <field type="uint8_t" name="target_component">Component ID</field>
  <extensions/>
  <field type="uint8_t" name="mission_type" enum="MAV_MISSION_TYPE">Mission type,
    see MAV_MISSION_TYPE</field>
</message>
```

Un mesaje enviado sería MISSION_CLEAR_ALL {target_system : 0, target_component : 0}

```

1 class Server:
2
3     def __init__(self, port, baudrate):
4         self.master = mavutil.mavlink_connection(port, baudrate, autoreconnect=True)
5         print('Connection established to device')
6         self.master.wait_heartbeat()
7         print("Heartbeat Received")
8
9 #test = Server("/dev/ttyUSB0", 57600) # Connection to the real APM device
10 test = Server("udp:192.168.1.133:14558",57600) # Connection to SITL

```

Figura 4.3: Conexión con el APM

4.2.2. Conexión y configuración de la comunicación con el APM

De cara a facilitar la implementación del software, ahorrarnos crear los comandos MAVLink a mano y hacer el código más legible utilizaremos la librería Pymavlink.

Los import que se utilizan son los siguientes:

```

1 from pymavlink import mavutil, quaternion, mavwp
2 from pymavlink.dialects.v10 import ardupilotmega as mavlink

```

Se importa mavutil para ardupilotmega para obtener los interfaces en python desde pymavlink y mavwp y quaternion como soporte a la transformación a quaterniones y viceversa.

El primer paso para poder interactuar con el APM es el proceso de conexión. En el comando de conexión se le indica una de las siguiente tuplas:

- Dispositivo_serie, baudrate
- Tipo_de_puerto:IP:puerto, baudrate. En este caso baudrate es obligatorio pero se desecha.

Con estas líneas se inicia el servidor de APM Server y se realiza la conexión con el dispositivo APM o con el simulador. Se puede apreciar cómo se realiza la conexión en el método `__init__` del servidor. Y la creación de 2 objetos de servidor, una que conectaría con el dispositivo real y la que se realiza con SITL.

Una vez conectado, se debe indicar al APM qué mensajes se quieren recibir y la frecuencia con la que se quieren recibir. Debido a que se trata de un prototipo y se trata de extraer el máximo potencial del mismo, se indicará el set de instrucciones completo y una frecuencia de moderada a alta 50 hz, lo máximo que soporta el APM adquirido.

```

1 RATE = 50
2 self.master.mav.request_data_stream_send(self.master.target_system,
3                                         self.master.target_component,
4                                         mavutil.mavlink.MAV_DATA_STREAM_ALL,
5                                         RATE, 1)

```

Figura 4.4: configuración de la conexión

Desde éste momento se van a estar volcando a un buffer los mensajes que manda nuestro APM, con lo que ahora ya se puede acceder a ellos para procesarlos.

4.2.3. Lectura de datos del APM

Una vez establecida la conexión y fijados los parámetros de configuración ya se puede acceder a los datos. Para ésto en el bloque de transformación se levantará el siguiente hilo que procesará los mensajes.

```

1 MsgHandler = threading.Thread(target=self.mavMsgHandler, args=(self.master,), name='
2     msg_Handler')
3 MsgHandler.start()
4
5 def mavMsgHandler(self, m):
6     while True:
7         msg = m.recv_msg()
8         if time.time() - self.lastSentHeartbeat > 1.0:
9             self.master.mav.heartbeat_send(mavlink.MAV_TYPE_GCS, mavlink.MAV_AUTOPILOT_INVALID
10                 , 0, 0, 0)
11             self.lastSentHeartbeat = time.time()
12             # refresh the attitude
13             self.refreshAPMPose3D()
14             self.refreshAPMnavdata()
15             elif msg is None or msg.get_type() == "BAD_DATA":
16                 time.sleep(0.01)
17                 continue

```

Figura 4.5: Hilo de alimentación de mensajes MAVLink

4.2.4. Envío de misiones al APM

Con el fin de entender mejor el código en la figura 4.2 podemos observar un gráfico de cómo es el protocolo de envío de misiones al APM a través de MAVLink obtenido de la universidad University of Colorado Boulder[9]

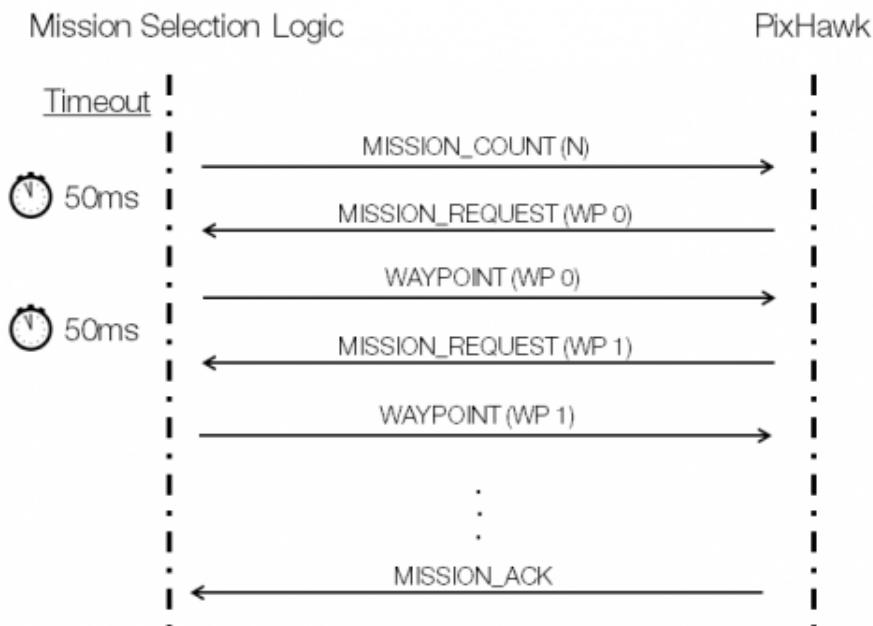


Figura 4.6: Protocolo de envío de misiones MAVLink

Para abordar el problema e implementar dicho protocolo partimos de la solución que nos propone la Universidad de Colorado Boulder[9] y la adaptamos a nuestras necesidades recibiendo de entrada un objeto de la interfaz `mission` de JdeRobot, transformándolo y enviándolo como misión. También revisamos si se han enviado órdenes de despegue y aterrizaje y en el caso de que así sea añadimos sus *mission item* al principio o al final de la misión. En éste punto se entrelazan la capa de interpretación con la capa de comunicación con el APM

```

1 def setMission(self, mission):
2
3     wp = mavwp.MAVWPLoader()
4     seq = 1
5     frame = mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT
6     radius = 10
7     pose3Dwaypoints = mission.mission
8     N = len(pose3Dwaypoints)
9     if (self.extra.takeOffDecision):
  
```

```

10     navData = pose3Dwaypoints[seq-1]
11     toff = mavutil.mavlink.MAVLink_mission_item_message(self.master.target_system,
12                                         self.master.target_component,
13                                         seq,
14                                         frame,
15                                         mavutil.mavlink.
16                                         MAV_CMD_NAV_TAKEOFF, # 22
17                                         0, 0, 0, radius, 0, 0,
18                                         navData.x, navData.y, navData.
19                                         h)
20
21     ...
22
23     for i in range(N):
24         navData = pose3Dwaypoints[i]
25         wayPoint_tmp = mavutil.mavlink.MAVLink_mission_item_message(self.master.
26                                         target_system,
27                                         self.master.target_component,
28                                         seq,
29                                         frame,
30                                         mavutil.mavlink.
31                                         MAV_CMD_NAV_WAYPOINT, # 16
32                                         0, 0, 0, radius, 0, 0,
33                                         navData.x, navData.y, navData.
34                                         h)
35
36     ...
37
38     if (self.extra.landDecision):
39         navData = pose3Dwaypoints[N-1]
40         land = mavutil.mavlink.MAVLink_mission_item_message(self.master.target_system,
41                                         self.master.target_component,
42                                         seq,
43                                         frame,
44                                         mavutil.mavlink.
45                                         MAV_CMD_NAV_LAND, # 21
46                                         0, 0, 0, radius, 0, 0,
47                                         navData.x, navData.y, navData.
48                                         h)
49
50     ...
51
52     self.master.waypoint_clear_all_send()
53     self.master.waypoint_count_send(wp.count())
54
55
56     for i in range(wp.count()):
57         msg = self.master.recv_match(type=['MISSION_REQUEST'], blocking=True)
58         self.master.mav.send(wp.wp(i))
59
60
61     self.master.arducopter_arm()

```

```

50     self.master.set_mode_auto() # start mission
51     ...

```

Como se puede observar en el código, en caso de tener `takeOffDecision` o `landDecision` activos en el atributo extra (mapeado a través de Ice) se añaden los comandos de despegue y aterrizaje, que interpreta son el primero y el último. Se puede observar también al final del código como se arman los motores con la línea `self.master.arducopter_arm()` y como se pone el dispositivo en modo automático con `self.master.set_mode_auto()`. El driver está implementado para ejecutarse en modo auto, se puede ejecutar también en modo guided pero el modo auto a diferencia del guided no inhabilita la radio y se puede recuperar en cualquier momento el control del robot aéreo. Si bien hay que decir que el modo guided o modo guiado es más preciso que el modo auto.

4.3. Capa de conexión con aplicaciones JdeRobot

En ésta capa se sirven y reciben los objetos de interfaces JdeRobot a través de los cuales se habla con la aplicación, que es el destino final de la información de sensores y la fuente original de los comandos de actuación para el avión, de modo que sean entendibles por el resto del driver. Para ésto se han implementado 4 hilos, uno por cada interfaz JdeRobot que se sirve o se recibe. En cada uno de éstos hilos se levanta un servidor ICE para recibir o servir objetos de interfaces JdeRobot mapeando atributos de la clase del servidor al estos servicios ICE.

Los 4 interfaces que van a ser servidos desde APM Server son:

- Pose3D
- NavData
- Extra
- Mission

Pose3D, NavData y Extra se describieron en el capítulo 3.1.1, adicionalmente a éstos se ha desarrollado un nuevo interfaz para dar soporte al uso de misiones, el interfaz `mission`.

```

class Pose3DData //we consumes Pose3DData
{
float x;

```

```

float y;
float z;
float h;
float q0;
float q1;
float q2;
float q3;
};

["python:seq:list"] sequence<Pose3DData> PoseSequence;

/**
 * Mission data information
 */
class MissionData
{
    PoseSequence mission;
};

/***
 * Interface to the Mission.
 */
interface Mission
{
    idempotent MissionData getMissionData();
    int setMissionData(MissionData data);
};

```

```

1      PoseTheading = threading.Thread(target=self.openPose3DChannel, args=(self.pose3D,) ,
2                                         name='Pose_Theading')
3      PoseTheading.daemon = True
4      PoseTheading.start()
5
6      ...
7
8      ExtraTheading = threading.Thread(target=self.openExtraChannel, args=(self.extra,) ,
9                                         name='Extra_Theading')
10     ExtraTheading.daemon = True
11     ExtraTheading.start()

```

Figura 4.7: Creación y arranque de los hilos encargados de servir los interfaces ICE

Cada hilo abre un servicio Ice, como todos los servicios ICE que implementamos son similares podremos como ejemplo de implementación el servicio de Pose3D. Como se puede

apreciar en esta capa tenemos 4 hilos de ejecución simultáneos sirviendo objetos ICE a través de servidores como el que sigue a éstas líneas. En paralelo se tiene otro hilo que procesa los mensajes recibidos del APM o SITL, los interpreta y cumplimenta éstos objetos ICE con lo que se ha tenido que cuidar el código para evitar problemas de concurrencia.

```

1 def openPose3DChannel( self , pose3D):
2     status = 0
3     ic = None
4     # recovering the attitude
5     Pose2Tx = pose3D
6     try :
7         ic = ICE.initialize(sys.argv)
8         adapter = ic.createObjectAdapterWithEndpoints("Pose3DAdapter", "default -p 9998")
9         object = Pose2Tx
10        # print object.getPose3DData()
11        adapter.add(object, ic.stringToIdentity("ardrone_pose3d")) #ardrone_pose3d Pose3D
12        adapter.activate()
13        ic.waitForShutdown()
14    except :
15        traceback.print_exc()
16        status = 1
17    if ic :
18        # Clean up
19        try :
20            ic.destroy()
21        except :
22            traceback.print_exc()
23            status = 1
24
25    sys.exit(status)
```

Figura 4.8: Servidor ICE encargado de servir objetos de la interfaz Pose3D

4.4. Bloque de interpretación

En ésta capa se interpretan los mensajes recibidos de una y otra capa y se transforman en el formato que sea necesario para que o interprete la opuesta.

El método `setMission()` se llama al recibir vía ICE una mission. Para identificar cuando llega, debido a que tenemos mapeados las variables con Ice, se ha construido un *listener* que valida el valor de la variable `mission` y si tiene contenido articula el método `setMission(mission)`.

```

1 MissionListener = threading.Thread(target=self.missionListener, name='MissionListener')
2 MissionListener.daemon = True
3 MissionListener.start()
4
5 def missionListener(self):
6     ...
7     Function who listen to a new mission reviewed from ICE MissionChannel thread and send
8     it to APM
9     :return: None
10    ...
11    while True:
12        if not self.mission.is_empty():
13            self.lastMission = self.mission
14            self.setMission(self.mission.getMissionData())
15            time.sleep(1)

```

Figura 4.9: Listener encargado de detectar misiones entrantes

Ya se visto cómo se realiza la transformación entre mensajes de misión en el método `setMission(mission)`, ahora nos centraremos en interpretar los mensajes recibidos del APM y crear los objetos de las interfaces que sean necesarias para su interpretación por JdeRobot. Esto lo realizamos principalmente en 2 métodos, `refreshAPMPose3D()` y `refreshAPMnavdata()`. Estos métodos se centran en buscar dentro de los paquetes recibidos aquellos que contienen información relevante, ya descritos en el capítulo 4.2, analizarlos y extraer de ellos la información necesaria y guardarla en objetos de interfaces JdeRobot para ser servidos a través de ICE.

En estas líneas de código, aparte de los métodos que recogen la actitud, podemos ver cómo se buscan los mensajes que necesitamos para extraer la información de los sensores desde `self.master.messages`. En esta variable se vuelcan los datos mensajes recibidos de la placa estabilizadora.

```
1 def refreshAPMPose3D(self):
2     # get attitude of APM
3     if 'ATTITUDE' not in self.master.messages:
4         self.attitudeStatus = 1
5         q=[0,0,0,0]
6     else:
7         attitude = self.master.messages['ATTITUDE']
8         # print(attitude)
9         yaw = getattr(attitude,"yaw")
10        pitch = getattr(attitude,"pitch") * -1
11        roll = getattr(attitude,"roll")
12        q = quaternion.Quaternion([roll, pitch, yaw])
13    # get altitude of APM
14    altitude = self.master.field('VFR_HUD', 'alt', None)
15    ...
16    # get GPS position from APM
17    ...
18    gps = self.master.messages['GPS_RAW_INT']
19    latitude = getattr(gps,"lat")/ 10e6
20    longitude = getattr(gps,"lon") / 10e6
21    self.GPS_fix_type = getattr(gps,"fix_type")
22    ...
23    # refresh the pose3D
24    ...
```

Figura 4.10: Código de ejemplo de `refreshAPMPose3D()` encargado de recorrer los mensajes y obtener los datos de actitud que se van a servir a través de Pose3D

Capítulo 5

AUV Commander

UAV Commander es una aplicación JdeRobot que hace las veces de estación de tierra o GCS (Ground Control Station) para robots aéreos. Desde UAV Commander se pueden programar misiones, enviar éstas al driver MavLink, verificar el cumplimiento de las mismas y visualizar los sensores a bordo del robot aéreo.

5.1. Diseño

La aplicación UAV Commander se estructura en 3 grandes bloques:

- GUI. La interfaz de usuario, es la parte de la aplicación que va a interactuar con el usuario.
- Core. El core de la aplicación será quien haga de orquestador entre los datos recibidos del bloque de comunicación con los driver JdeRobot y el interfaz de usuario. Se hará cargo también de obtener, de la fuente que sea preciso, IGN, Google o el bloque de conexión de drivers de JdeRobot la información necesaria y procesarla para ser mostrada por el GUI.
- Bloque de conexión con los driver JdeRobot.

La aplicación debe cumplir los requisitos descritos en el capítulo 2.2.3 que pueden resumirse en 4 grandes milestones:

1. Permitir la creación de misiones
2. Enviar misiones a través de interfaces ICE al driver conectado.

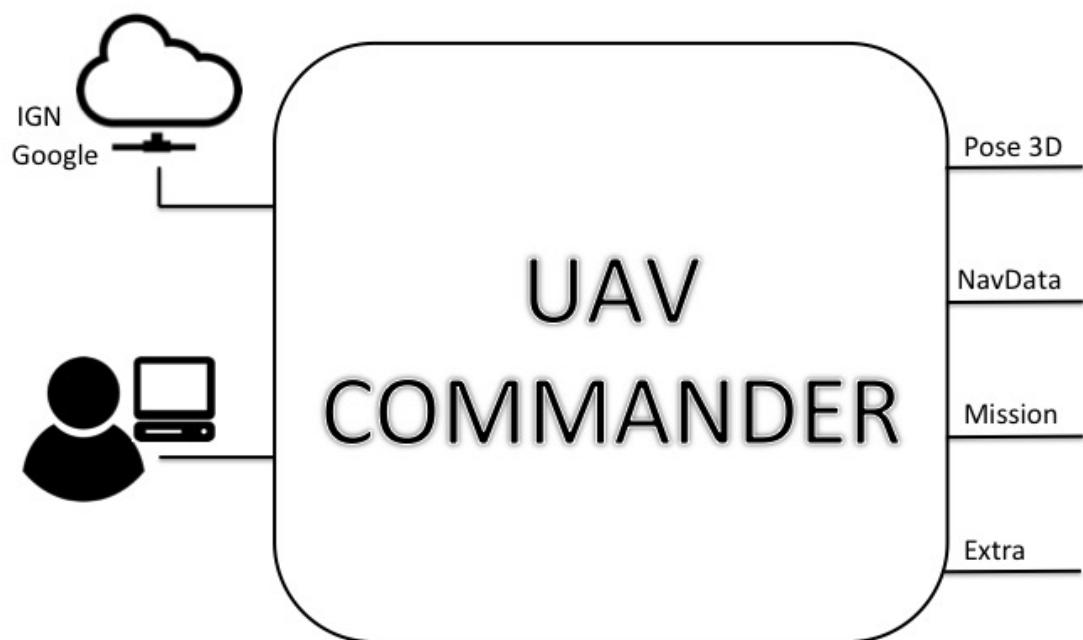


Figura 5.1: Diseño de caja negra de UAV Commander

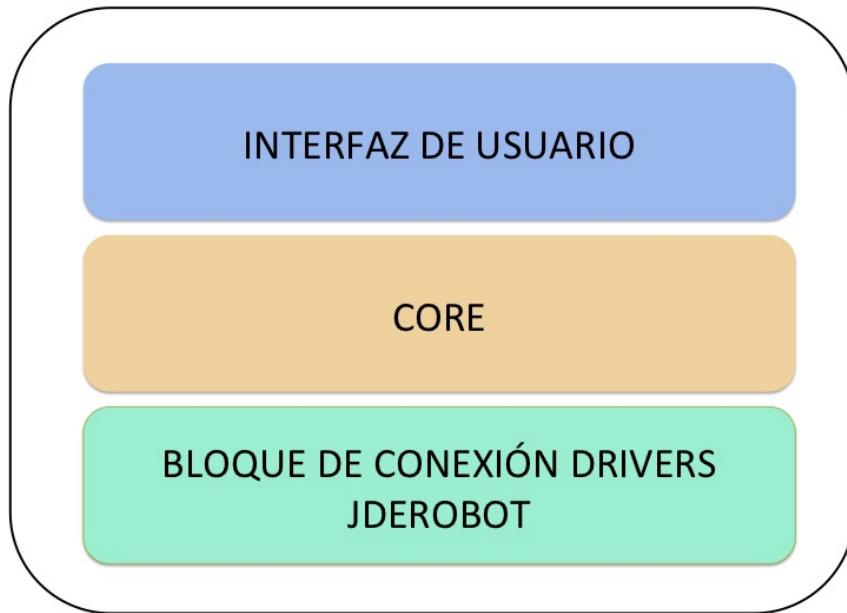


Figura 5.2: Diseño de UAV Commander

3. Presentar toda la información sensorial que nos ofrezca el driver.

- Actitud
- Cámaras de abordo

4. Validar el cumplimiento de misiones.

5.2. Bloque de conexión con aplicaciones JdeRobot

En éste bloque se implementan todos los servidores ICE necesarios para acceder a la información sensorial y para enviar los interfaces necesarios para la actuación. Éste bloque es análogo al Bloque de comunicación con aplicaciones JdeRobot del Capítulo 4.1 si bien para simplificar este proceso hemos optado por utilizar las clases implementadas por Aitor Martinez Fernandez[8] y las clases de EasyIce de Victor Arribas Raigadas[2] ambas integradas en JdeRobot. De éste modo éste bloque queda simplificado y su inicialización queda relegada a unas pocas líneas de código:

```

1  ic = EasyIce.initialize(sys.argv)
2  camera = CameraClient(ic, "UavViewer.Camera", True)
3  navdata = NavDataClient(ic, "UavViewer.Navdata", True)
4  pose = Pose3DClient(ic, "UavViewer.Pose3D", True)
5  cmdvel = CMDVel(ic, "UavViewer.CMDVel")
6  extra = Extra(ic, "UavViewer.Extra")
7  mission = MissionI(ic, "UavViewer.Mission")

```

Figura 5.3: Creación de los servicios ICE y lincado de los mismos a sus variables

En la aplicación UAV Commander se exponen 5 interfaces ICE a diferencia de los 4 que exponía el driver APM Server. Además de los que expone APM Server, a saber:

- Pose3D
- NavData
- Extra
- Mission

Se implementan 2 más:

- El interfaz camera para mostrar las imágenes y vídeo.
- El interfaz CmdVel, cuya implementación actual no se utiliza pero se ha implementado para futuras realeases, donde en un futuro puedan añadirse el control por velocidades.

Con ésto se definen los 6 objetos ICE que envían y/o reciben por el APM Server, y con el código siguiente se los asignamos al GUI quien los guarda como atributos en su clase principal.

```

1  screen.setCamera(camera)
2  screen.setPose3D(pose)
3  screen.set_initial_pose3D(pose3D)
4  screen.setNavData(navdata)
5  screen.setCMDVel(cmdvel) # Not used yet
6  screen.setExtra(extra)
7  screen.setMission(mission)

```

Figura 5.4: Pase de parámetros al interfaz gráfico

5.3. Core

El core es el bloque más complicado de la aplicación, se divide a su vez en 5 bloques:

- Obtención y actualización de mapas. Éste bloque es el encargado de obtener los mapas de cualquiera de las 2 fuentes actualmente implemetadas, el IGN o Google. También se encarga de actualizar la georeferenciación asociada a ellos una vez descargados.
- Utilidades de mapas. Aquí se encuentran los métodos de apoyo a la georeferenciación como: obtener el *bounding box* de una posición GPS, validar si nos acercamos al límite del mapa descargado o la transformación de magnitudes.
- Proyección. Es sin duda una de las partes más complejas del aplicativo. Son las clases encargadas de a partir de una posición GPS, obtener su posición en imagen que representa el mapa de la zona y viceversa.
- Procesamiento de los datos de vuelo. En este bloque se extraen, transforman (en caso que sea necesario) y distribuyen los datos de vuelo a las ventanas correspondiente y objetos correspondientes del GUI .
- Control de misiones. El control de misiones *orquesta* llamadas al resto de bloques del core para dar soporte a las misiones. Muestra éstas en la GUI y cuando desde la GUI se pulsa el botón de “Send to APM” prepara los objetos del bloque de comunicación con el diver para ser enviados.

5.3.1. Obtención y actualización de mapas

Para obtener los mapas geo-referenciados necesitamos principalmente 4 cosas como vimos en el capítulo 3:

1. La posición.
2. El *Bounding Box*.
3. El tamaño de la imagen.
4. La proyección a utilizar.

```

1 def retrieve_new_map(lat, lon, radius, width, height):
2     bbox = getBoundingBox(lat, lon, radius)
3     wms = WebMapService('http://www.ign.es/wms-inspire/pnoa-ma', version='1.3.0')
4     img = wms.getmap(layers=['OI.OrthoimageCoverage'],
5                       styles=['default'],
6                       srs='EPSG:4326',
7                       bbox=(bbox),
8                       size=(width, height),
9                       format='image/png',
10                      transparent=True)
11 ...
12 ImageUtils.prepareInitialImage(img.read(), width, height)
13 opencv_image = cv2.imread("images/imageWithDisclaimer.png", 1)
14 image = {'bytes': opencv_image, 'bbox': bbox, 'size': (width, height)}
15 return image

```

Figura 5.5: Método `retrieve_new_map(lat, lon, radius, width, height)`

Con el fin de abstraernos del montaje de la URL y envío y tratamiento de la respuesta utilizaremos la librería OWSLib¹, en su versión 0.14.0, que nos ayudará con éstos trámites. El código para obtener un nuevo mapa sería:

En el código mostrado en la Figura 5.5 podemos observar como recuperamos un nuevo mapa del IGN², que publica uno de sus numerosos WMS en '<http://www.ign.es/wms-inspire/pnoa-ma>'. Primero obtenemos el *bounding box*. El *bounding box* es la zona geográfica que vamos a recuperar, se calcula a partir de la posición GPS central y, en nuestro caso al mostrar imágenes cuadradas, un radio, (el código para calcularlo podemos observarlo en la Figura 5.6). Una vez obtenido el *bounding box* únicamente rellenamos el resto del objeto WMS y lanzamos la request que devuelve una imagen en bytes a la que debemos añadir el disclaimer que nos indica el IGN. Creamos la petición WMS con la URL del WMA PNOA y asignamos los parámetros necesarios para obtener el mapa. Al final montamos la imagen geo-referenciada recibida, que en nuestro caso será un diccionario con la imagen en OpenCV el *bounding box*, y el tamaño de la imagen.

¹<https://geopython.github.io/OWSLib/>

²Instituto geográfico nacional

```

1 def getBoundingBox(lat, lon, distance):
2     radValues = from_degrees(lat, lon)
3     rad_dist = distance / EARTH_RADIUS_WGS84
4     min_lat = radValues[0] - rad_dist
5     max_lat = radValues[0] + rad_dist
6     if min_lat > MIN_LAT and max_lat < MAX_LAT:
7         delta_lon = asin(sin(rad_dist) / cos(radValues[1]))
8         min_lon = radValues[1] - delta_lon
9         if min_lon < MIN_LON:
10             min_lon += 2 * pi
11         max_lon = radValues[1] + delta_lon
12         if max_lon > MAX_LON:
13             max_lon -= 2 * pi
14     # a pole is within the distance , The up code isn't accurate in these cases
15     else:
16         min_lat = max(min_lat, MIN_LAT)
17         max_lat = min(max_lat, MAX_LAT)
18         min_lon = MIN_LON
19         max_lon = MAX_LON
20     # back to degrees
21     southWestPoint = from_radians(min_lat, min_lon)
22     northEastpoint = from_radians(max_lat, max_lon)
23     return southWestPoint[1], southWestPoint[0], northEastpoint[1], northEastpoint[0]

```

Figura 5.6: Método getBoundingBox(lat, lon, distance) encargado de calcular esquina inferior izquierda y esquina superior derecha de un cuadrado de ancho distance

5.3.2. Utilidades de mapas

Aquí se encuentran los métodos necesarios para trabajar con geo-posicionamiento, como el método de cálculo de *bounding box* que se ha visto anteriormente. Se encuentran también transformaciones de radianes a grados, o el cálculo de la distancia entre 2 puntos.

Como ejemplo tenemos el código para calcular distancias siempre u cuando utilicemos la proyección WGS84 en la Figura 5.7.

5.3.3. Proyección

La proyección se basa en “ubicar” un punto dado en un determinado sistema de coordenadas en otro sistema completamente distinto. En nuestro caso trabajamos hasta con 3

```

1 def distance_to(valores, other):
2     radius = EARTH_RADIUS_WGS84
3     radValuesIni = from_degrees(valores[0], valores[1])
4     radValuesOther = from_degrees(other[0], other[1])
5     return radius * acos(
6         sin(radValuesIni[0]) * sin(radValuesOther[0]) +
7         cos(radValuesIni[0]) *
8         cos(radValuesOther[0]) *
9         cos(radValuesIni[1] - radValuesOther[1])
10    )

```

Figura 5.7: Método `distance_to(valores, other)`, que calcula distancia entre puntos

sistemas de coordenadas distintos el de las imágenes con X[0, MAX_WIDHT] creciente de izquierda a derecha e Y[0, MAX_HEIGHT] creciente de arriba a abajo, el sistema de coordenadas geográficas y el cartesiano.

Ambas se mueven en órdenes de magnitud diferentes y un movimiento de $1 * 10^{-5}$ puede suponer varias decenas de píxeles.

Para abordar este problema, y dado que se ha trabajado con robots aéreos con una autonomía relativamente reducida, se ha asumido que la tierra es plana en *bounding box* de menos de 2 kilómetros de ancho. Con ésta convención conseguimos simplificar notablemente la complejidad del problema, que prácticamente se convierte en una normalización de un sistema a otro. Si en un futuro se decidiese utilizar la aplicación para vuelos de mayor envergadura se debería tener en cuenta la curvatura de la tierra. La proyección en UAV Commander se realiza en ambos sentidos, de imagen a posición geográfica y de sistema de coordenadas a imagen.

Podemos ver la proyección de posición geográfica a imagen en la Figura 5.8.

5.3.4. Procesamiento de los datos de vuelo

Este bloque se encarga de extraer los datos de vuelo y proporcionárselos a los componentes del GUI que o necesiten, así como recogerlos del GUI y transformarlos a objetos de interfaz ICE. Este bloque sería muy similar al bloque de interpretación del driver APM Server. Aquí se encontrarían los métodos de refresco de datos, *getters* y *setters*.

```

1 def posCoords2Image(lonMin , latMin , lonMax , latMax , lat , lon , tamImageX , tamImageY):
2     distCoordX = round(latMax - latMin , 7)
3     distCoordY = round(lonMax - lonMin , 7)
4     x = ((latMax - lat) * (tamImageX)) / distCoordX
5     y = ((lon - lonMin) * (tamImageY)) / distCoordY
6     return round(y) , round(x)

```

Figura 5.8: Método `posCoords2Image(lonMin, latMin, lonMax, latMax, lat, lon, tamImageX, tamImageY)`, encargado de calcular la proyección de un punto en coordenadas geográficas a la matrix de píxeles de la imagen que representa el mapa

5.3.5. Control de misiones

Este bloque incluye todos los métodos necesarios para dar soporte a misiones. Se apoya fuertemente en los bloques de GUI y de conexión con drivers JdeRobot, y realiza la transformación de datos y el envío de estos de una capa a otra. Métodos como `update_waypoints()` o `sendWP(send2APM)` estarían englobados en éste bloque siendo éste último sin duda alguna el método más importante del bloque. Se apoya en el GUI para obtener los Waypoints de la vista de la tabla, interpretarlos, y llenar el objeto de interfaz missión con ella. Se puede ver en la Figura 5.9 cómo se realiza este tratamiento.

5.4. Interfaz de usuario

La interfaz de usuario es una de las partes más importantes de toda aplicación, se trata de la parte que interactúa con el usuario, con lo que la información se ha de mostrar de forma agradable, armónica y de forma precisa, para que no genere confusión en el usuario final.

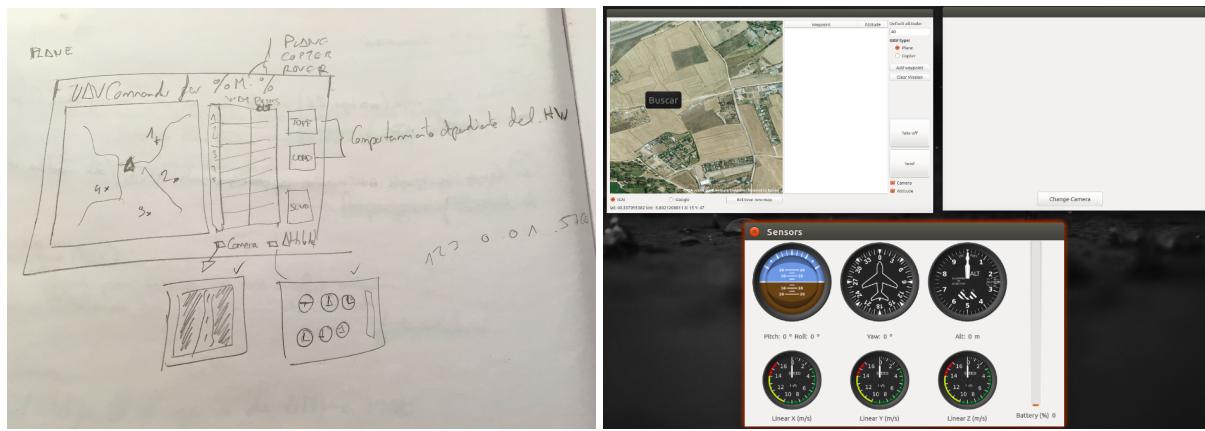
Es por tanto el bloque donde más horas se han invertido en su desarrollo junto con el bloque core de la aplicación.

La aplicación consta de 3 pantallas:

1. Pantalla principal.
2. Ventana de actitud.
3. Ventana visor de cámara.

```
1 def sendWP(self, send2APM):
2     ...
3     for row in range(colCount):
4         ...
5         pos_lat = text.find("lat:")
6         pos_lon = text.find("lon:")
7         if pos_lat != -1:
8             lat = (text[pos_lat + 4:pos_lon])
9             pose.x = float(lat)
10            lon = (text[pos_lon + 4:])
11            pose.y = float(lon)
12        else:
13            if "LAND in " in text:
14                ...
15                self.extra.land()
16            else:
17                pos = text.find("TAKE OFF to ") + 12
18                ...
19                self.extra.takeoff()
20                alt = self.table.item(row, 1)
21                pose.h = int(alt.text())
22                mission.mission.append(pose)
23                if i==0:
24                    mission.mission.append(pose)
25                i += 1
```

Figura 5.9: Método `sendWP (self, send2APM)`, encargado de enviar la misión creada por el usuario al driver



(a) Boceto en papel de uno de los diseños de UAV Commander

(b) Vista general de UAV Commander

Las pantallas de actitud y el visor de cámara se han integrado del desarrollo de UAV Viewer de Alberto Martín Florido, con lo que nos centraremos en la pantalla principal, la encargada del control y creación de misiones.

Para su correcto desarrollo se comenzó dibujando varios bocetos que fueron descartándose y puliendo hasta que se fijó el aspecto visual de la pantalla.

Para facilitar su diseño vamos a dividir ésta ventana en 3 bloques:

1. Creación y envío de misiones.
2. Control de misiones.
3. Conexión con sensores. Éste bloque tan solo conecta con las integraciones de las pantallas de actitud y visor de cámara integradas desde UAV Viewer.

5.4.1. Creación y envío de misiones

Para poder dar ésta funcionalidad de una forma intuitiva y sencilla se han implementado:

- Un mapa interactivo dónde un click crea un punto de misión, dibujando un número secuencial que muestra el orden la misión. Además según van añadiéndose nuevos puntos UAV Commander va dibujando la ruta que “debería” seguir el robot para el cumplimiento de su misión. Un simple click desencadenaría:

1. Inclusión del waypoint en la lista de waypoints

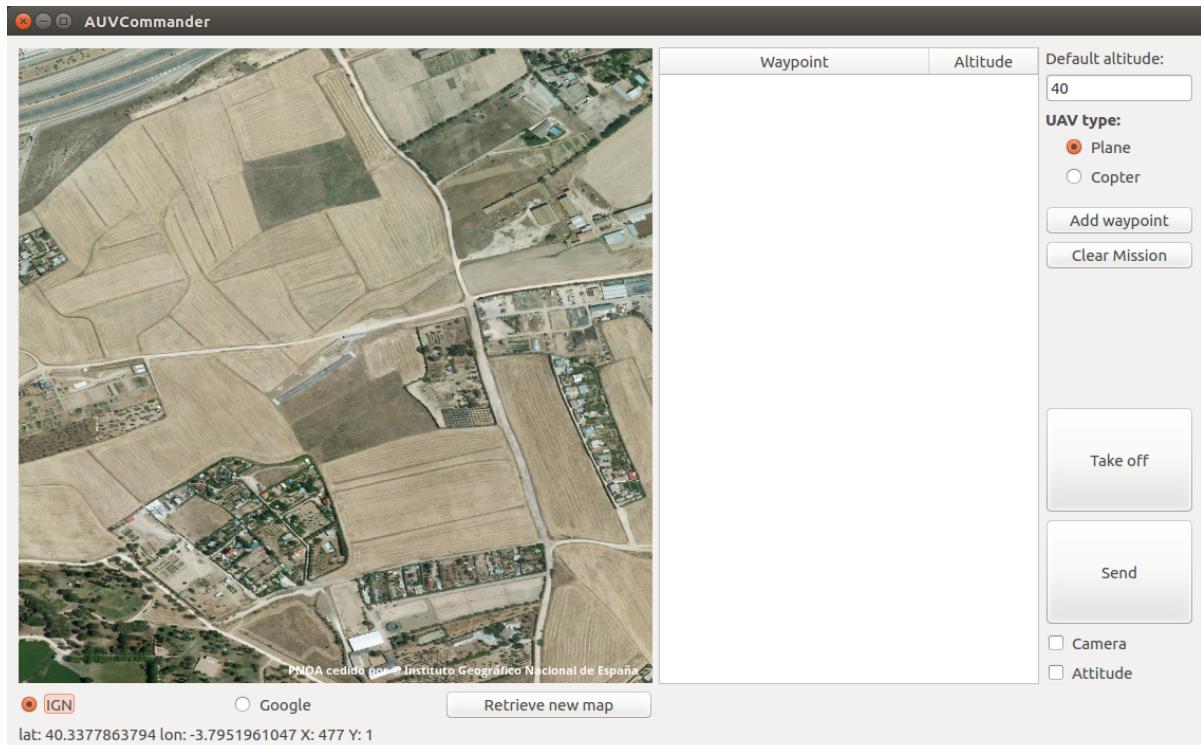


Figura 5.10: Pantalla principal de UAV Commander

2. Llamada al método de proyección `posImage2Coords(x, y, sizeX, sizeY, latMin, lonMin, latMax, lonMax)` para obtener a partir del x e y de la imagen la latitud y longitud dónde se desea ir.
3. Control de misión a realizar, verificando si se trata de un despegue, aterrizaje o un punto de paso.
4. Inserción en la tabla de seguimiento de waypoints.
5. En la próxima iteración del `threadMap` a través de la llamada a `update_position()`.
6. Se llama al método `setPosition(self, x, y, angle)` quien ejecuta el método.
7. `set_waypoints(self, wayPoints, current=True)` quien pinta el nuevo waypoint y la ruta a seguir en el mapa.

Las siguientes líneas de código muestran la implementación del método encargado de pintar los waypoints y las rutas en el mapa `set_waypoints(self, wayPoints, current=True)`

```

1 def set_waypoints( self , wayPoints , current=True):
2     n=0
3     pts = np . array (wayPoints , np . int32 )
4     pts = pts . reshape (( -1 , 1 , 2 ))
5     cv2 . polylines ( self . cvImageShadow , [ pts ] , False , ( 250 , 250 , 250 ) , thickness=1 )
6     for waypoint in wayPoints :
7         n+=1
8         s = str (n)
9         cv2 . circle ( self . cvImageShadow , ( waypoint [0] , waypoint [1] ) , 1 , [ 0 , 0 , 255 ] ,
10                      thickness=-1 , lineType=1 )
11         cv2 . putText ( self . cvImageShadow , s , ( waypoint [0] + 3 , waypoint [1] ) , cv2 .
12                         FONT_HERSHEY_COMPLEX_SMALL , 1 , [ 0 , 0 , 255 ] , thickness=1 )
13         self . refreshImage ()

```

Figura 5.11: Método `set_waypoints(self, wayPoints, current=True)`, encargado de pintar los puntos de paso en el mapa

- Botón de “Take Off”/“Land”. Éste botón apoya al mapa para crear misiones con aterrizaje y despegue, si se quiere despegar hacia un punto, bastaría con pulsarlo y pulsar después hacia dónde se quiere despegar. El comportamiento para el aterrizaje sería análogo.
- Tabla de seguimiento de waypoints. En ésta tabla se almacenan los waypoints que se van creando al hacer click sobre el mapa, cada fila es editable, de tal forma que ofrece una gran precisión en caso que sea necesario.
- Campo “default altitude” Establece la altitud de crucero, el valor de éste campo se irá propagando en cada waypoint creado en el atributo `altitude`.
- Botón “Send”. Éste botón llama al método `sendWP(self, send2APM)` del core quien recoge los objetos de misión, los procesa y los envía al bloque de comunicación con drivers JdeRobot para su envío a través de los interfaces ICE necesarios, a saber, `mission` y `extra` en el caso en el que haya aterrizaje o despegue en la misión.

5.4.2. Control de misiones

El control de misiones es la parte más tediosa de todo el desarrollo, en ella se muestra el comportamiento del robot aéreo sobre el mapa; la ruta que ha seguido, la posición actual y su orientación.

Para llevar ésto a cabo se ha tenido que dividir el mapa en capas, tenemos 3 capas:

1. Mapa original con el disclaimer. `self.cvImage`
2. Mapa original con la estela que ha dejado el robot aéreo. `self.cvImageShadow`
3. Mapa que va a mostrarse, donde se pinta la posición actual del robot aéreo `self.im_to_show`

Éste planteamiento por capas nos permite, por ejemplo, mantener la estela del robot y pintar la posición actual, añadir waypoints sin alterar lo ya dibujado, etc. Pero complica la implementación de todos los métodos que interactúan en este bloque. Algunos de los métodos las importantes del desarrollo del aplicativo se encuentran aquí y se listan a continuación:

- `update_position(self)`
- `self.setPosition(self, x, y, angle)`
- `draw_triangle(self, x, y, angle)`
- `refresh_shadow(self,x,y)`
- `refreshImage(self)`

En cada iteración del hilo `threadMap` se articula el siguiente árbol con las llamadas:

- `update_position(self)`
 - `GeoUtils.is_position_close_border(lat, lon , bbox)`
 - `ImageUtils.posCoords2Image(lonMin, latMin, lonMax, latMax, lat, lon, tamImageX, tamImageY)`
 - `setPosition(self, x, y, angle)`
 - `refresh_shadow(self, x, y)`
 - `set_waypoints(self.wayPoints)`
 - `draw_triangle(self, x, y, angle)`
 - ◊ `GeoUtils.center_of_triangle(triangle)`
 - ◊ `GeoUtils.change_coordinate_system(triangle, center,True)`

```

1 def update_position(self):
2
3     if self.pose != None:
4         pose = self.getPose3D().getPose3D()
5         lat = pose.x
6         lon = pose.y
7         bbox = self.imageMetadata["bbox"]
8         self.limit_warning = GeoUtils.is_position_close_border(lat, lon, bbox)
9         if self.limit_warning:
10             self.download_zoomed_map()
11         else:
12             bbox = self.imageMetadata["bbox"]
13             imagesize = self.imageMetadata["size"]
14             x, y = ImageUtils.posCoords2Image(bbox[0], bbox[1], bbox[2], bbox[3], lat, lon
15                 , imagesize[0], imagesize[1])
16             angle = self.sensorsWidget.quatToYaw(pose.q0, pose.q1, pose.q2, pose.q3)
17             self.setPosition(x, y, angle)
18
19     def setPosition(self, x, y, angle):
20         self.refresh_shadow(x,y)
21         self.set_waypoints(self.wayPoints)
22         self.draw_triangle(x, y, angle)
23         self.refreshImage()
24
25     def refresh_shadow(self,x,y):
26         cv2.circle(self.cvImageShadow, (x, y), 1, [255,255,102], thickness=-1, lineType=8,
27                     shift=0)

```

Figura 5.12: Métodos encargados de actualizar la posición y el recorrido efectuado en el mapa

- ◊ `rotate_polygon(triangleCartesian,angle)`
- `set_waypoints(self.wayPoints)`
- `self.refreshImage()`

A continuación se muestran las implementaciones de los métodos con más peso de éste bloque:

Capítulo 6

Experimentos

En este capítulo se presentan las pruebas realizadas al sistema y el hardware necesario para la reproducción de las mismas. Estos experimentos corresponden a los casos de prueba descritos en el capítulo 2.2. y se dividen en nueve pruebas repartidas en dos bloques: experimentos en simulación y experimentos con el avión real.

6.1. Experimentos simulados

En este apartado se presentan todos los experimentos realizados en simulación uno a uno, a saber:

1. Experimento de integración de todo el software.
2. Experimento de envío de misión con avión en vuelo.
3. Experimento de envío de misión con despegue y varios waypoints.
4. Experimento de envío de misión con despegue y aterrizaje.
5. Experimento de autozoom forzando la salida del avión del mapa.

6.1.1. Experimento de integración de todo el software

El objetivo de este experimento es probar que la integración del sistema y SITL se ha realizado correctamente. Para ello se van a realizar los siguiente pasos:

1. Arranque de la máquina virtual donde hemos instalado SITL.
2. Se arranca SITL.
3. Se mapea una nueva salida en la ip 192.168.138 en el puerto 14550 UDP.
4. Se carga una misión de prueba.
5. Se da comienzo a la misión simulada.
6. Se arranca APM Server.
7. Se arranca UAV Commander

Los pasos 1, 2 y 3 serán recurrentes en el resto de experimentos en simulación con los que de ahora en adelante se van a englobar en una supratarea que llamaremos “Arranque y configuración de SITL”

Una vez realizada la secuencia arriba descrita se comprobará el seguimiento del avión simulado a través del mapa y se comprobará que la ventana de actitud responde correctamente.

Se comprueba que funciona correctamente y se evidencia en el siguiente vídeo <https://www.youtube.com/watch?v=rHTi0buUETg>

6.1.2. Experimento envío de misión con avión en vuelo

El objetivo de este experimento es probar que el envío de misiones a APM Server funciona correctamente y validar su cumplimiento. Para ello se van a realizar los siguientes pasos:

1. Arranque y configuración de SITL.
2. Se carga una misión de prueba.
3. Se da comienzo a la misión simulada.
4. Se arranca APM Server.
5. Se arranca UAV Commander
6. Se crea una misión en UAV Commander
7. Se envía la misión a APM Server a través del botón “Send” de UAV Commander

Una vez realizada la secuencia arriba descrita se comprobará el seguimiento de la misión enviada a través del mapa validando su correcto cumplimiento. La prueba se evidencia en el vídeo https://www.youtube.com/watch?v=_X8WGSMIxug

6.1.3. Experimento de envío de misión con despegue y varios waypoints

El objetivo de este experimento es probar que el envío de misiones y maniobra de despegue a APM Server funciona correctamente y validar su cumplimiento. Para ello se van a realizar los siguiente pasos:

1. Arranque y configuración de SITL.
2. Se arranca APM Server.
3. Se arranca UAV Commander
4. Se crea una misión en UAV Commander. Ésta ha de contener como primer item la maniobra de despegue.
5. Se envía la misión a APM Server a través del botón “Send” de UAV Commander

Una vez realizada la secuencia arriba descrita se comprobará el seguimiento de la misión enviada a través del mapa validando su correcto cumplimiento. La prueba se evidencia en el vídeo <https://www.youtube.com/watch?v=0j3QttXIRbg>

6.1.4. Experimento de envío de misión con despegue y aterrizaje

El objetivo de este experimento es probar que el envío de misiones con maniobra de despegue y aterrizaje a APM Server funciona correctamente y validar su cumplimiento. Para ello se van a realizar los siguiente pasos:

1. Arranque y configuración de SITL.
2. Se arranca APM Server.
3. Se arranca UAV Commander

4. Se crea una misión en UAV Commander. Ésta ha de contener como primer item la maniobra de despegue y como último item la maniobra de aterrizaje.
5. Se envía la misión a APM Server a través del botón “Send” de UAV Commander

Una vez realizada la secuencia arriba descrita se comprobará el seguimiento de la misión enviada a través del mapa validando su correcto cumplimiento. La prueba se evidencia en el vídeo https://www.youtube.com/watch?v=4yf_PmFgR5Y

6.1.5. Experimento de autozoom forzando la salida del avión del mapa

El objetivo de este experimento es probar que el autozoom de UAV Commander funciona correctamente y validar su cumplimiento. Para ello se van a realizar los siguiente pasos:

1. Arranque y configuración de SITL.
2. Se carga una misión de prueba que dirija al avión fuera de los márgenes del mapa.
3. Se da comienzo a la misión simulada.
4. Se arranca APM Server.
5. Se arranca UAV Commander

Una vez realizada la secuencia arriba descrita se comprobará que UAV Commander descarga un nuevo mapa con una perspectiva mas amplia para seguir al avión. La prueba se evidencia en el vídeo <https://www.youtube.com/watch?v=vEcMbJoJ-Qo>

6.2. Experimentos con el avión real

En este apartado se presentan todos los experimentos realizados con el avión real, a saber:

1. Pruebas de envío recepción del sistema de radiofrecuencias.
2. Experimento de integración con APM Server y UAV Commander.
3. Experimento de seguimiento del avión a través del UAV Commander.
4. Experimento de envío de misión desde el avión en vuelo.

6.2.1. Pruebas de envío recepción del sistema de radiofrecuencias

El objetivo de este experimento es probar que se reciben correctamente las órdenes de la emisora de radiocontrol y probar el alcance del wifi del ordenador de abordo. Ésto nos delimitará el alcance al que vamos a volar. Para ello se van a realizar los siguiente pasos:

1. Arranque de la emisora de radicontrol.
2. Arranque del avión a radiocontrol y del ordenador de abordo.
3. Se separará el avión de la emisora y del ordenador hasta que se pierda la señal del wifi validando que responde correctamente a las órdenes enviadas por la emisora de radiocontrol.

Este experimento nos va a permitir conocer a qué distancia vamos a poder volar.

6.2.2. Experimento de integración con APM Server y UAV Commander

El objetivo de este experimento es probar que la integración del sistema y el avión real a radiocontrol se ha realizado correctamente. Para ello se van a realizar los siguiente pasos:

1. Arranque de la emisora de radicontrol.
2. Arranque del avión a radiocontrol y del ordenador de abordo.
3. Se arranca APM Server.
4. Se arranca UAV Commander

Una vez realizada la secuencia arriba descrita se comprobará desde UAV Commander:

1. Descarga el mapa de la zona en la que nos encontramos.
2. Se comprueba dando un paseo que se actualiza la posición en el mapa.
3. Se comprueba la actitud desde la ventana de actitud

La prueba se evidencia en el vídeo *****

6.2.3. Experimento de seguimiento del avión a través del UAV Commander

El objetivo de este experimento es probar el seguimiento de misiones a través de UAV Commander con avión real a radiocontrol. Para ello se van a realizar los siguiente pasos:

1. Arranque de la emisora de radicontrol.
2. Arranque del avión a radiocontrol y del ordenador de abordo.
3. Se arranca APM Server.
4. Se arranca UAV Commander
5. Se pone el avión a radiocontrol en el aire mediante el sistema de radiocontrol.
6. Se vuela el avión a radiocontrol siguiendo un recorrido marcado.

Una vez realizada la secuencia arriba descrita se comprobará que UAV Commander muestra la trayectoria descrita por el avión a radiocontrol en el mapa. La prueba se evidencia en el vídeo

6.2.4. Experimento de envío de misión desde el avión en vuelo

El objetivo de este experimento es probar el envío de misiones al avión real a radiocontrol. Para ello se van a realizar los siguiente pasos:

1. Arranque de la emisora de radicontrol.
2. Arranque del avión a radiocontrol y del ordenador de abordo.
3. Se arranca APM Server.
4. Se arranca UAV Commander
5. Se pone el avión a radiocontrol en el aire mediante el sistema de radiocontrol.
6. Se crea una midión desde UAV Commander y se envía a APM Server a través del botón “Send”

Una vez realizada la secuencia arriba descrita se comprobará desde UAV Commander el cumplimiento de la misión enviada. La prueba se evidencia en el vídeo

Capítulo 7

Conclusiones

7.1. Consecución de objetivos

Como se ha podido ir viendo a lo largo de este PFC: Se ha presentado el problema, los objetivos y los requisitos que deben cumplir en el capítulo dos. En el capítulo tres ha analizado y profundizado en la infraestructura que se iba a utilizar y se ha decidido que infraestructura era la mejor para abordar nuestros desarrollos. Se ha abordado cada objetivo por separado planteando un diseño que nos permita abordar de una forma sencilla y eficaz el desarrollo y se han implementado las soluciones planteadas. Con estos desarrollos, descritos en los capítulos 4 y 5, se han cubierto los requisitos presentados en el capítulo 2. Y por último se ha validado su funcionamiento mediante los experimentos del capítulo seis. Resumiendo:

1. Se ha preparado un avión real con el hardware necesario para dotarle de autonomía a través de JdeRobot
2. Se ha desarrollado un driver para JdeRobot, APM Server que nos permite conectarnos tanto al avión real como a uno simulado a través de SITL
3. Se ha construido una aplicación JdeRobot, UAV Commander para dar soporte al control de misiones en robots aéreos de ala fija, aviones.
4. Se ha validado el funcionamiento de ambos tanto en un ambiente simulado como con el avión real mediante sendos experimentos.

El seguimiento y evolución de este PFC, así como el código, vídeos y material utilizado puede encontrarse en la web <http://jderobot.org/Jafernandez>

7.2. Trabajos futuros

Este PFC abre las puertas a futuros desarrollos con este tipo de robots aéreos, que hasta ahora JdeRobot no soportaba, sentando las bases del desarrollo de aviones autónomos en JdeRobot. Se han identificado varias evoluciones de los desarrollos aquí expuestos, comenzaremos con APM Server

7.2.1. Futuros evolutivos de APM Server

- Se puede ampliar el soporte de misiones de APM Server hasta cubrir la totalidad de los comandos de misión de MAVLink. De esta forma se podría por ejemplo enviar misiones del tipo “dar X vueltas de Y radio en torno al Waypoint Z” o aumenta la velocidad durante la misión a x Km/h.
- Se pueden ampliar las interfaces para enviar información sobre la misión, el objetivo que he alcanzado, el que estoy ejecutando, a qué distancia se encuentra del objetivo, etc.

7.2.2. Futuros evolutivos de UAV Commander

- Se puede mejorar la proyección en los mapas recibidos de Google para aumentar la precisión en los mapas.
- Se pueden mejorar los cálculos de proyección para que sean mas precisos independientemente de la zona del globo donde se encuentre.
- Se puede dar soporte a través de UAV Commander a nuevos desarrollos permitiendo la ejecución de los mismos, escritos en determinado archivo, a través de un botón en la aplicación.
- Se puede añadir el control por velocidades CMDVel para dar soporte a todo tipo de drones.

- Se puede ampliar la creación de waypoints dibujando, además de la ruta a seguir, la distancia entre los puntos de control.

7.2.3. Futuros evolutivos generales

Como desarrollo adicional se podría conectar SITL a Gazebo, de ésta forma tendríamos la simulación completa y se podrían probar las misiones en entornos más realistas. Esto facilitaría la simulación de desarrollos que utilicen la visión en aviones.

Bibliografía

- [1]
- [2] V. Arribas. Mediawiki, 2017.
- [3] businessinsider.com. Drones are about to fill the skies within the next 5 years, Jun 2016.
<http://www.businessinsider.com/the-drones-report-research-use-cases-regulations-and-issues-2016-4>.
- [4] J. cano. Mediawiki, 2016.
<http://jderobot.org/J.canoma-tfg>.
- [5] Droneii.com. Drone investment trends 2016, May 2016.
<http://www.droneii.com/drone-investment-trends-2016>.
- [6] A. M. Florido. Mediawiki, 2014.
<http://jderobot.org/Amartinflorido-tfg>.
- [7] D. Jiménez. Mediawiki, 2017.
<http://jderobot.org/Jimenez-tfg>.
- [8] A. Martínez. Mediawiki, 2015.
<http://jderobot.org/Aitormf-tfg>.
- [9] U. of Colorado Boulder. Mavlink protocol: Waypoints, May 2015.
<http://www.colorado.edu/recuv/2015/05/25/mavlink-protocol-waypoints>.
- [10] rtl sdr.com. Spoofing gps locations with low cost tx sdrs, Sep 2015.
<http://www rtl-sdr com spoofing-gps-locations-with-low-cost-tx-sdrs/>.

[11] J. Vela. Mediawiki, 2017.

<http://jderobot.org/Jvela-tfg>.