Nicolas Vanegas Herrera

Sistema de comunicación y control de dron en la nube

Trabajo de fin de grado

Dirigido por Marc Sanchez Artigas

Grado de ingenieria informática

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2024

Resum.

Els drons avui dia tenen una àmplia varietat d'usos gràcies als avanços en les tecnologies de microprocessadors, que han permès la creació de dispositius més petits, potents i eficients. Aquests avanços han revolucionat la manera en què es dissenyen i operen els drons, convertint-los en eines essencials en sectors com l'agricultura, la logística, la fotografia i la vigilància. La comunicació amb drons és ara molt més senzilla, ja que aquests dispositius s'han transformat en sistemes altament complexos, equipats amb programari avançat que permet l'automatització de múltiples processos. Això no sols ha facilitat la seva operació, sinó que també ha ampliat les seves capacitats, permetent la seva integració en una varietat d'aplicacions tecnològiques.

En aquest context, el projecte desenvolupat s'enfoca en una aplicació innovadora que permet controlar un dron des de qualsevol plataforma, ja sigui una aplicació web, un dispositiu mòbil, o fins i tot altres sistemes connectats. Aquesta aplicació ha estat dissenyada amb l'objectiu d'oferir un control precís i versàtil del dron, la qual cosa permet als usuaris adaptar-se a diverses situacions i necessitats. Els resultats obtinguts amb aquest projecte inclouen una aplicació amb un codi bastant modular, la qual cosa facilita el seu manteniment, escalabilitat i implementació de noves funcionalitats. A més, es va aconseguir una ràpida recepció de peticions i una àgil publicació de dades.

Resumen.

Los drones hoy en día tienen una amplia variedad de usos gracias a los avances en las tecnologías de microprocesadores, que han permitido la creación de dispositivos más pequeños, potentes y eficientes. Estos avances han revolucionado la manera en que se diseñan y operan los drones, convirtiéndolos en herramientas esenciales en sectores como la agricultura, la logística, la fotografía y la vigilancia. La comunicación con drones es ahora mucho más sencilla, ya que estos dispositivos se han transformado en sistemas altamente complejos, equipados con software avanzado que permite la automatización de múltiples procesos. Esto no solo ha facilitado su operación, sino que también ha ampliado sus capacidades, permitiendo su integración en una variedad de aplicaciones tecnológicas.

En este contexto, el proyecto desarrollado se enfoca en una aplicación innovadora que permite controlar un dron desde cualquier plataforma, ya sea una aplicación web, un dispositivo móvil, o incluso otros sistemas conectados. Esta aplicación ha sido diseñada con el objetivo de ofrecer un control preciso y versátil del dron, lo que permite a los usuarios adaptarse a diversas situaciones y necesidades. Los resultados obtenidos con este proyecto incluyen una aplicación con un código bastante modular, lo que facilita su mantenimiento, escalabilidad e implementación de nuevas funcionalidades. Además, se logró una rápida recepción de peticiones y una ágil publicación de datos.

Abstract.

Drones today have a wide variety of uses thanks to advances in microprocessor technologies, which have enabled the creation of smaller, more powerful and efficient devices. These

advances have revolutionized the way drones are designed and operated, making them essential tools in sectors such as agriculture, logistics, photography and surveillance. Communication with drones is now much simpler, as these devices have been transformed into highly complex systems, equipped with advanced software that allows the automation of multiple processes. This has not only facilitated their operation, but has also expanded their capabilities, allowing their integration into a variety of technological applications.

In this context, the developed project focuses on an innovative application that allows controlling a drone from any platform, be it a web application, a mobile device, or even other connected systems. This application has been designed with the aim of offering precise and versatile control of the drone, allowing users to adapt to various situations and needs. The results obtained with this project include an application with a fairly modular code, which facilitates its maintenance, scalability and implementation of new functionalities. In addition, a fast reception of requests and an agile publication of data were achieved.

Indice.

1. Figuras.

2. Tablas.

3. Introducción.

3.1. Motivación.

En 1849 Venecia buscando su libertad del imperio austriaco se alza en contra declarando su independencia. Austria, dadas las circunstancias, despliega un conjunto de tácticas militares para retomar el control. La más conocida fue el despliegue de múltiples globos no tripulados para bombardear la ciudad, el cual acabaría marcando en la historia de la humanidad el primer despliegue de vehículos aéreos no tripulados (UAV). Desgraciadamente para los Austriacos, el ataque falló por un cambio en la dirección del viento, que modificó drásticamente la trayectoria de los UAV.[1]

Ilustración 1.Figura X. Sofoque insurrección veneciana por parte del imperio austriaco en 1849.

A través de los años, la industria de la aviación ha ido creciendo a pasos de gigante gracias al desarrollo de nuevas tecnologías. Tanto los vehículos aéreos tripulados como los no tripulados han tenido una rápida evolución en el mercado militar. Algunos motivos por los que estos vehículos han tenido un gran desarrollo son sus utilidades en misiones de reconocimiento de terrenos, ofensivas militares y vigilancia de zonas hostiles.

El avance de tecnologías relacionadas con el procesamiento de datos ha sido notable. Las aplicaciones potenciadas por inteligencias artificiales, junto con el tamaño y consumo reducido de los microchips actuales, han contribuido a esta evolución. Como resultado, los mercados objetivo para los que se utilizan estos vehículos se han expandido considerablemente.

Amelia-HUB es una empresa que presta múltiples servicios relacionados con el uso de drones. La empresa se enfoca en la toma de fotografías que involucra desde fines cinematográficos

hasta la inspección de terrenos mediante el uso de tecnologías LiDAR (detección y localización por luz). Uno de sus más recientes proyectos consistió en hacer una captura de datos con el dron Matrice 300 para realizar análisis topográficos de un terreno en Palencia haciendo uso de las tecnologías LiDAR.

La motivación principal de este proyecto surge por la obtención de una licitación del gobierno de la Generalitat, que buscaba desarrollar un software de detección de personas en tiempo real usando drones. Aparte del desafío de crear el software para realizar la detección también hay otro reto que surge y es la necesidad de crear una aplicación capaz de controlar a los drones.

Actualmente existen muchas aplicaciones de escritorio desde las cuales se puede hacer el control de los drones como por ejemplo Mission Planner o QGroundControl. Pero ninguna de estas nos permite integrar su software de control con las aplicaciones ya existentes en Amelia-HUB por lo que la mejor solución en este caso es desarrollar un software propio de control de drones.

3.2. Objetivos.

El objetivo de este proyecto es crear una aplicación capaz de controlar cualquier tipo de dron a distancia volviéndolos autosuficientes y quitando la necesidad de disponer de un piloto en el área. Para conseguirlo se ha propuesto hacer una aplicación web, que sea capaz de controlar el dron en tiempo real siguiendo el siguiente esquema.

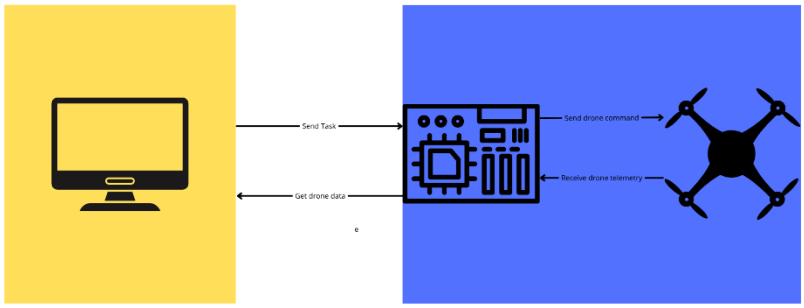

Ilustración 2.Esquema general de la aplicación.

Se necesitan tres elementos para lograr una arquitectura similar al diagrama anterior. Una aplicación para clientes que consista en una estructura de cliente-servidor, en la que el usuario envía peticiones/comandos a un servidor el cual se conectará al pc abordo y enviará las peticiones al dron. Para recibir los datos, el servidor tendrá que obtener los datos que envía al pc abordo y dejarlos en un almacenamiento de mensajes (como una cache) para que el cliente los pueda leer desde la aplicación.

Este proyecto se enfocará sobre todo lo relacionado con la creación del pc abordo. De esta manera se desarrollará una aplicación cuya finalidad es enviar comandos al dron, obtener información sobre su estado y sus diferentes componentes y publicarlos en alguna cache para que otras aplicaciones puedan usarlos a nivel usuario.

3.3. Planificación.

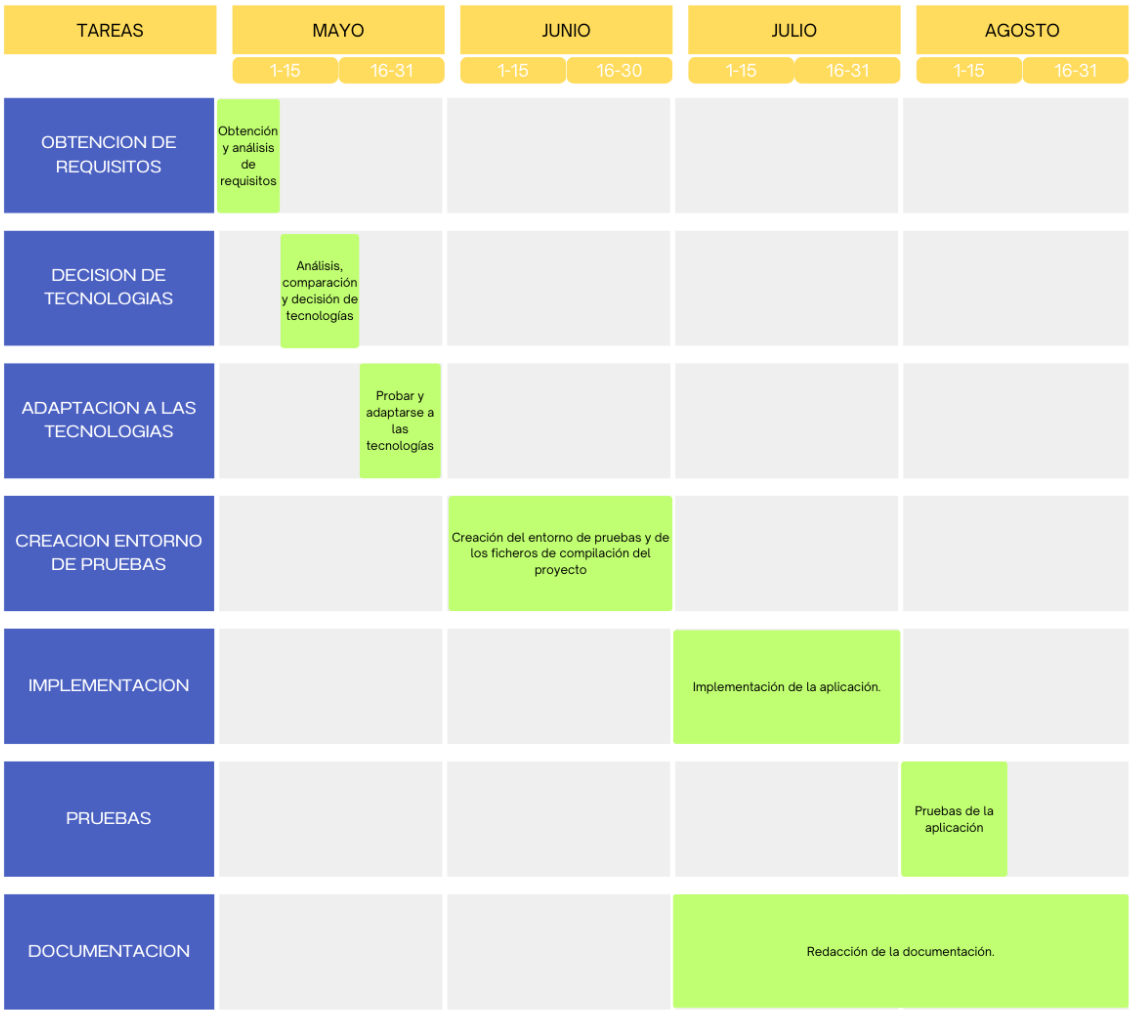

Ilustración 3Planificación del desarrollo de la aplicación.

La planificación de este proyecto consta de 7 fases.

- Obtención de requisitos:
  - Fecha: 1-7 de mayo.
  - Descripción: En esta fase de desarrollo se obtienen los requisitos de usuario y se y se realiza un análisis de estos, creando los diferentes diagramas UML que se usaran como punto de partida de la aplicación.
- Decisión de tecnologías:
  - Fecha: 8-14.
  - Descripción: En esta fase de desarrollo se buscan que tecnologías pueden resultar útiles para el proyecto. Una vez se tienen las diferentes posibilidades, se hace una comparación entre ellas y se escogen las que mejor se adapten a los requisitos de usuario.
- Adaptación a las tecnologías.
  - Fecha: 15-31.

    - Descripción: En esta fase de desarrollo se busca probar las tecnologías y adaptarse a ellas mediante la creación de pequeñas aplicaciones de pruebas independientes entre ellas para poder entender de qué forma funcionan.
- Creación entorno de pruebas:
    - Fecha: 3-28 de junio.
    - Descripción: En esta fase de desarrollo se buscan los diferentes elementos que usaremos para poder probar la aplicación a medida que vayamos progresando. También se realiza la creación e implementación de los ficheros que compilan todo el proyecto.
- Implementación:
    - Fecha: 1-31 de julio.
    - Descripción: En esta fase de desarrollo se realiza la implementación de la aplicación.
- Pruebas:
    - Fechar: 1-14 de agosto.
    - Descripción: En esta fase de desarrollo se realizan pruebas exhaustivas tanto en dron virtual como real de la aplicación y se corrigen posibles bugs que puedan ir apareciendo.
- Documentación:
    - Fecha: 1 mayo a 30 agosto.
    - Descripción: En esta fase de desarrollo se realiza la documentación de la aplicación.

4. Introducción a los UAV.

4.1. UAV.

Los UAV (unmanned aerial vehicle) mejor conocidos hoy en día como drones, son vehículos aéreos de control remoto y programables, ya que no requieren la presencia de un piloto. Los drones utilizan fuerzas aerodinámicas para elevarse mediante el movimiento de hélices remotamente controladas. Los drones también son vehículos que por su diseño pueden ser prescindibles o no recuperables y pueden disponer de una carga útil letal o no letal. Estos vehículos aéreos se diseñaron inicialmente para hacer trabajos que suponían un peligro para los humanos, relacionado con maniobras de carácter militar que suponían un alto coste de vidas humanas. Gracias al avance de las tecnologías y a la reducción de los costes y tamaño de los diferentes elementos de hardware de dichos sistemas, sus aplicaciones se han multiplicado y sus usos hoy superan lo militar. Entre algunos de ellos cabe mencionar: monitoreos de incendios forestales, inspecciones de infraestructura, entretenimiento, topografía aérea, entre muchos otros.[2]

Ilustración 4.General Atomics MQ-1 Predator. Dron militar ofensivo usado por la CIA en Afganistán.ref. [3]

4.2. Tipos.

Según el departamento de defensa de los estados unidos, se ha creado un estándar para la clasificación de los diferentes UAV, de los cuales principalmente destacaremos peso o tipo de mecanismos, altitud máxima de vuelo, grado de autonomía operacional, rol operacional, entre otros.

Clasificación por grupos.

La primera clasificación define 5 grupos diferentes los cuales están dividas según su peso máximo a la hora de despegar, la altitud máxima a la que pueden operar y las velocidades máximas a las que pueden llegar.

Grupo:	**Grupo 1**	**Grupo 2**	**Grupo 3**	**Grupo 4**	**Grupo 5**

Nombre según tamaño	Pequeño	Mediano	Grande	Mayor	Máximo
Peso máximo de despegue.	< 9.1kg	> 9.07kg & < 24.9kg	> 24.9kg & < 600kg	>600 kg	>600 kg
Altitud de operaciones	<370 m	<1,100 m	<5,500 m	<5,500 m	>5,500 m
Velocidad	<190 km/h	< 460 km/h	<460 km/h	Cualquier velocidad	Cualquier velocidad

Tabla 1. Clasificación de drones según grupos. Ref [4]

Clasificación por alcance y resistencia.

Los drones también pueden ser clasificados según el alcance y distancia que pueden recorrer tomando como punto de partida la estación de despegue.

Categoría:	**Vehículos aéreos no tripulados de muy corto alcance**	**Vehículos aéreos no tripulados de corto alcance**	**Vehículos aéreos no tripulados de corto alcance**	**Vehículos aéreos no tripulados de alcance medio**	**Vehículos aéreos no tripulados de largo alcance**
Alcance (km):	< 5	> 5 y < 50	> 50 y < 150	> 150 y < 650	> 650
Resistencia (hr):	0.5 – 0.75	1–6	8–12	12 – 36 o 48	> 36 o 48

Tabla 2. Clasificación por alcance y resistencia. Ref [4]

Clasificación por peso.

Para clasificar los drones según su peso los podemos dividir en las siguientes 5 categorías.

Categoría:	**Nano**	**Micro vehículos aéreos (MAV)**	**UAV miniatura o pequeño (SUAV)**	**Vehículos aéreos no tripulados medianos**	**Grandes vehículos aéreos no tripulados**
Peso:	< 250 g	≥ 250 g y <02 kg	≥ 02 kg y <25 kg	≥ 25 kg y <150 kg	≥ 150 kg

Tabla 3. Clasificación por peso. Ref [4]

4.3. Componentes.

En cuanto al diseño podemos ver que muchos de los componentes físicos de los drones son similares, aunque algunas excepciones. Por ejemplo, sus sistemas de control ambiental encargados de suministrar aire, controlar la temperatura y la presión dentro del dron, los cuales pueden variar en función del entorno en el que se encuentre y las características físicas como su tamaño o su peso máximo. Algunos suelen tener compartimentos en los que pueden portar diferentes objetos también denominados “payloads” como podrían ser cámaras térmicas, sensores de gas, compartimientos de carga, antenas de telecomunicaciones, entre otros.

Por lo general, al ser vehículos que en su interior no tienen vidas humanas, se construyen con materiales muy ligeros y poco resistentes. Esto conlleva al desarrollo de vehículos más rápidos y frágiles. También, como la fiabilidad de estos vehículos no es tan importante como la de un avión, el tiempo de pruebas de estos es mucho menor, lo que permite una mayor velocidad de fabricación.

Estos sistemas tienen distintos componentes de cómputo que permiten reemplazar a los pilotos por sistemas de control remotos como sistemas en un chip (SOC) o computadoras de placa única (SBC).

Sistemas modernos de hardware diseñados para el control de los UAV suelen ser llamados controladores de vuelo (FC), placa controladora de vuelo (FCB) o autopiloto. Por lo general dichos sistemas incorporan un microprocesador primario, un secundario para casos en los que el primario falle y diferentes sensores como acelerómetros, giroscopios, magnetómetros y barómetros en un solo modulo. Su integración en los drones permite el control automático de los motores para encont`rar potencias que estabilicen y faciliten la manipulación del dron.

Ilustración 5. Raspberry Pi Model2B.

4.4. Arquitectura.

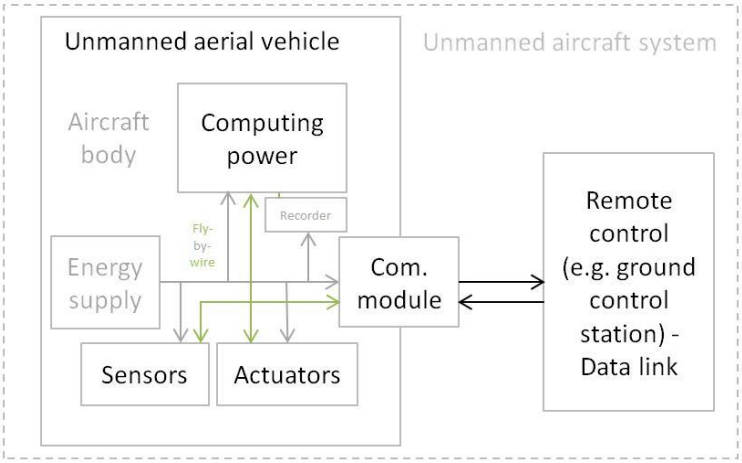

Ilustración 6. Estructura física general de un UAV.

Sensores:

Los sensores de un UAV son todos aquellos elementos hardware que nos proporcionan información sobre el estado del vehículo, estos se dividen en dos categorías. [5]

- *Exteroceptivos*: Se encargan de manejar información del exterior como el cálculo de distancias.
- *Propioceptivos*: Se encargan de manejar información del estado interno como el estado de la batería.

Actuadores:

Los actuadores son todos aquellos componentes que proporcionan fuerza, par o desplazamiento al dron a partir de alguna señal eléctrica, neumática o hidráulica al sistema algunos ejemplos podrían ser motores, hélices o servomotores entre otros. [5]

Software.

El software que se ejecuta dentro de los UAV se denomina piloto automático o pila de vuelo. El objetivo de este es ejecutar misiones de forma autónoma o con intervención remota. El autopiloto obtiene datos de los sensores, controla diferentes motores y facilita las comunicaciones con el pc abordo o estación de control (desde estas se controla remotamente al dron, se envían comandos y se obtienen los datos de telemetría recibidos por el autopiloto del UAV).

Los UAV son sistemas en tiempo real que requieren de una alta frecuencia para así poder evitar situaciones críticas en el sistema que lo puedan poner en riesgo. Algunos ejemplos pueden ser coaliciones con obstáculos, pérdidas de la estabilidad por fuertes ragas de viento, fuertes lluvias o nevadas, entre otras.

Capa	Requisito	Operaciones	Ejemplo
Firmware	El tiempo es crítico	Desde el código máquina hasta la ejecución del procesador, el acceso a la memoria	ArduCopter-v1, PX4
Middleware (Middleware)	El tiempo es crítico	Control de vuelo, navegación, gestión de radio	PX4, Cleanflight, ArduPilot
Sistema operativo	Intensivo en informática	Flujo óptico, evitación de obstáculos, SLAM, toma de decisiones	ROS, Nuttx, distribuciones de Linux, Microsoft IOT

Tabla 4. Descripción de la pila de desarrollo de un autopiloto.ref [6]

4.5. Comunicaciones.

La mayor parte de los UAV utilizan señales de radio para el control y el intercambio de datos desde el dron hasta el operador remoto. En sus principios dichos sistemas solo contaban con enlaces ascendentes de banda estrecha (es decir, solo podían enviar datos desde el piloto remoto al dron), pero con el paso del tiempo se integraron los enlaces descendentes, los cuales permitían enviar datos desde el dron hasta el operador remoto. Con esto, se consigue la creación

de enlaces bidireccionales permitían enviar datos de comando y control al dron y recibir datos de telemetría desde este. [7]

Las señales de radio que se emiten desde el operador pueden ser enviadas desde:

- Control terrestre: Alguien que opera un trasmisor o un receptor de radio que esta enlazado con el dron tales como un teléfono inteligente, una Tablet o una computadora (Estos reciben el nombre de estación de control terrestre GCS).
- Mediante sistemas de control remoto a través de redes de móviles.
- Desde otra aeronave.

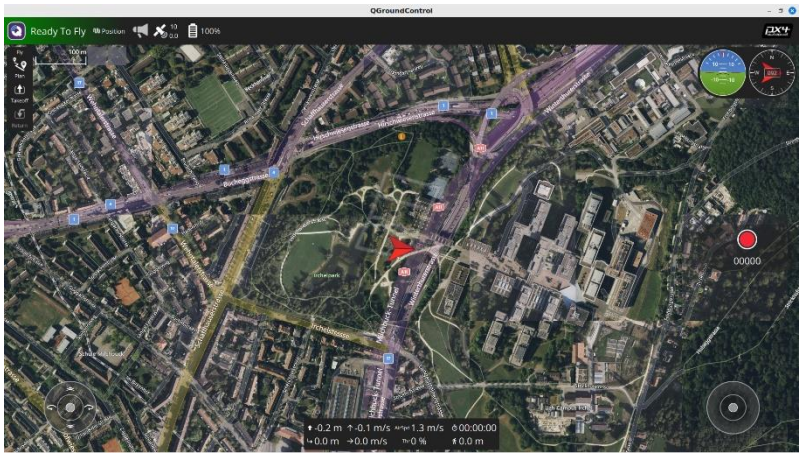

Ilustración 7. Aplicación estación de control"QGroundControl".

Antiguamente cada sistema de autopilotos seguía un protocolo de comunicación u otro dependiendo del fabricante, aunque con el tiempo se han ido estandarizando diversos protocolos de comunicaciones con el fin de homogeneizar la forma en la que las estaciones de control se comunican con los autopilotos. Algunos de los más conocidos son MAVLink (del cual hablaremos más adelante), DroneKit, ROS, entre otros.

4.6. Aplicaciones.

Hoy existen muchos campos donde los drones resultan de bastante ayuda. Esto es porque ayudan a aumentar la productividad de ciertos sectores como el de la agricultura, permitiendo realizar análisis automatizados de las zonas de cultivo, permitiendo reportes precisos sobre su estado. Gracias también a la autonomía que estos tienen, se les puede programar para que realicen sondeos automatizados de vigilancia en zonas donde se requiere disponer de seguridad las 24 horas del día. A continuación, se mencionarán otros campos en los cuales se suelen usar los drones:

Comercial.

Investigación científica para obtener datos en zonas de complejo acceso, en minería para analizar los volúmenes de material obtenido y para analizar el interior de las cuevas para evitar poner en riesgo la vida de los mineros, silvicultura, agricultura, en los puertos para analizar

cargas de los barcos, cinematografía realizando videos o fotografías para filmar diferentes ángulos.[8]

Guerra.

Drones objetivo, Micro vehículos para obtención de información, armas teledirigidas, para realizar misiones de reconocimiento.[8]

Civil.

Reparto de mercancía usando pequeños compartimientos donde se pueden poner cargas ligeras, suele ser bastante útil para enviar suministros médicos vitales en un periodo corto de tiempo. En la vigilancia de zonas conflictivas o monitorización del tráfico y el control de las velocidades de la vía pública.[8]

Entretenimiento.

Los drones también se usan para exhibiciones nocturnas con fines artísticos, los cuales se están convirtiendo en una alternativa mejor que los fuegos artificiales gracias a que son más silenciosos, más seguros y mejores para el medio ambiente.[8]

Monitoreo Ambiental.

Levantamientos topográficos para hacer modelos digitales de la superficie, Monitoreo de ecosistemas naturales para ver y controlar la biodiversidad de una zona, Agricultura de precisión con la cual se busca tener un análisis constante de los cultivos para poder mantenerlos siempre al máximo de producción, monitoreo de ríos y para controlar la integridad de cualquier tipo de infraestructura como la de un edificio.[8]

4.7. Entornos de desarrollo y pruebas.

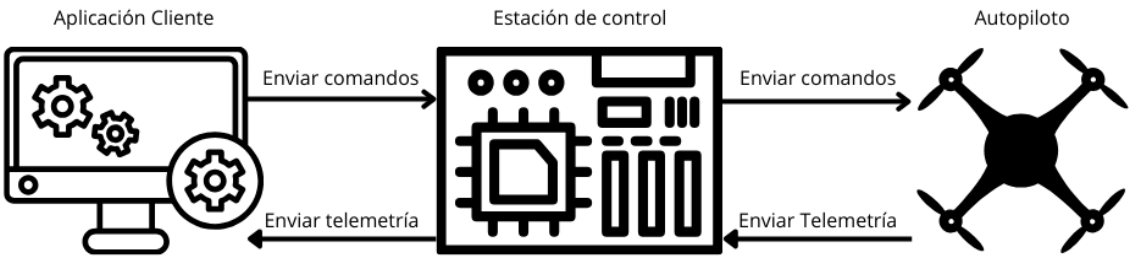

Ilustración 8. Esquema de flujo de comunicación entre los elementos que componen un UAV.

En el campo del desarrollo de software dedicado a drones podemos ver que la mayoría de las tecnologías que los componen son de código abierto.

Para el caso de los autopilotos podemos ver que la gran mayoría son gratuitas y sus desarrolladores permiten su uso a nivel profesional sin necesidad de obtener licencia. Algunos de los ejemplos pueden ser PX4, ArduPilot y Auterion.

Para el caso de las estaciones de control, podemos ver que hay múltiples aplicaciones que permiten el control y la configuración de drones como por ejemplo Mission Planner y

QGroundControl. Aunque hoy en día existen múltiples librerías como MAVSDK las cuales te permiten crear un pc abordo desde cero.

Fase de desarrollo y pruebas del pc abordo usando componentes virtuales.

El Pc abordo permite recibir peticiones de pilotos remotos y enviar comandos al dron usando un protocolo de comunicación que este sea capaz de entender. En ciertas ocasiones resulta útil poder desarrollar tu propio software para estos componentes puesto que te permite automatizar determinados procesos como la ejecución de automática de misiones.

Antes de comenzar a desarrollar, es importante crear un entorno de pruebas con el que podamos probar la aplicación. Para evitar daños materiales por un mal funcionamiento de nuestra aplicación, y reducir el tiempo de compilación de la aplicación, se recurre a cargar y ejecutar los binarios en el Pc abordo.

Para poder simular drones, hay muchas aplicaciones en internet de código abierto que cumplen este objetivo. Algunas de estas están contenidas en imágenes Docker (la cual es una Tecnología de contención de aplicaciones) lo que nos permite una fácil ejecución.

En muchas ocasiones resulta de interés poder probar que el comportamiento del dron es el esperado mediante los datos de telemetría que se obtienen de este como su posición actual, estado de la batería, velocidad, entre otros. Para lograr esto lo que podemos hacer es usar aplicaciones de terceros que están diseñadas para alcanzar nuestros requerimientos. Por ejemplo, QGroundControl o Mission Planner.

Fase de desarrollo y pruebas del pc abordo usando un dron real.

Una vez terminada la fase de desarrollo en entornos virtuales y todo funcione como se espera lo que tenemos que hacer posteriormente es probar la aplicación en un dron real.

Para esto, lo primero que tenemos que hacer es compilar nuestra aplicación y ejecutarla dentro del pc abordo. Las primeras pruebas son las de conexión con el dron y el pc abordo, una vez establecida la conexión entre estos dos y comprobada que funciona correctamente, se comprueba que los diferentes sensores funcionan correctamente mediante la obtención de telemetría.

Cuando queramos probar si el dron realiza correctamente los diferentes comandos que se le envíen como despegar o desplazarse, lo mejor para este tipo de pruebas es realizarlas en el exterior para evitar daños tanto en el dron como en las instalaciones. Como veremos en el siguiente apartado en el que se explicaran las regulaciones que hay a la hora de volar drones, lo mejor es realizar dichas pruebas en campos de vuelo autorizados para evitar posibles problemas legales.

4.8. Regulaciones.

En cuanto a normativa y regulaciones nos centraremos concretamente en la categoría abierta de drones, pues es en la que se ha tenido que seguir durante la fase de pruebas de la aplicación. Dicha categoría se refiere a aquellas operaciones que tienen un bajo riesgo operacional. No requieren autorización previa ni declaración por parte del operador antes del vuelo.

A continuación, se enumerarán algunas normativas esenciales a tener en cuenta al operar un vehículo aéreo teledirigido. Es importante señalar que existen muchas otras categorías y regulaciones que también son relevantes, pero en este apartado nos enfocaremos principalmente en los aspectos más básicos y esenciales.[9]

Registro de operadores.

- Los operadores que residan en España o tengan su actividad en el país deben registrarse en la sede electrónica de AESA si utilizan drones con un peso igual o superior a 250g o que puedan causar daños de más de 80 julios en caso de colisión.
- También deben registrarse si el dron tiene un sensor capaz de capturar datos personales, salvo que se trate de un juguete conforme a la Directiva de Juguetes.

Seguro de responsabilidad civil.

- Es obligatorio para drones con un peso igual o superior a 20 kg, conforme al Reglamento 785/2004.
- Para drones de menos de 20 kg, se aplica el Real Decreto 37/2001, con una cobertura mínima de 220.000 DEG.
- Están exentas de esta obligación las operaciones en subcategorías A1 y A3 (con drones de menos de 20 kg).

Documentación necesaria.

- Certificado de registro de operador de UAS.
- Certificado de formación del piloto (A1/A3, y A2 según sea necesario).
- Póliza de seguro de responsabilidad civil.
- Procedimientos para coordinar actividades si hay más de un piloto.

Subcategorías operacionales.

- A1: Operaciones evitando sobrevolar personas no participantes, con drones de menos de 250g o con marcado de clase C0 o C1.
- A2: Mantener una distancia de al menos 30 m de personas no participantes, utilizando drones con marcado de clase C2.
- A3: Operaciones en zonas seguras, lejos de personas y a más de 150 m de zonas residenciales, comerciales o recreativas, con drones de hasta 25 kg.

Edad mínima de pilotos.

- Generalmente es de 16 años, pero se reduce a 12 o 14 años en ciertos casos, dependiendo del tipo de dron y la subcategoría de operación.

Identificación y marcado.

- Desde enero de 2024, los drones deben tener un sistema de identificación a distancia (DRI) que transmita información del dron y del operador.

Toda la normativa mencionada proviene de AESA o agencia estatal de seguridad aérea la cual se encarga de hacer que se cumpla correctamente las normas de aviación civil dentro del territorio español.[9]

5. Especificación del controlador de drones.

5.1. Requisitos.

Documentación de los requisitos funcionales:

Requisito funcional 1: El usuario debe poder obtener los datos del dron a tiempo real. Algunos de estos datos pueden ser la posición GPS, la carga de la batería, el estado actual del dron.

Requisito funcional 2: El usuario debe poder enviar coordenadas al dron con los puntos por los cuales tiene que pasar, también ha de poder especificar diferentes parámetros como la velocidad a la que tiene que llegar a un punto determinado, la altura que tiene que alcanzar, el pitch, jaw y el roll.

Requisito funcional 3: El usuario debe poder hacer que el dron despegue del pc abordo en cualquier momento y ejecute las misiones remotamente sin tener la necesidad de haber un piloto físicamente en ese momento.

Requisito funcional 4: El usuario debe poder pausar las misiones en cualquier momento. Se espera que el dron se quede esperando hasta que el usuario le mande otra misión.

Requisito funcional 5: El usuario tiene que poder hacer que el dron vuelva a la estación de despegue en cualquier momento.

***Requisito funcional 6*:** El usuario debe ser capaz de hacer que el dron pause la ejecución de las misiones en cualquier momento.

Documentación de los requisitos no funcionales:

Requisito no funcional 1: La obtención de los datos de telemetría emitidos por el dron han de llegar con la menor latencia posible.

Requisito no funcional 2: El sistema tiene que ser capaz de atender a las peticiones que recibe lo más rápido posible.

Requisito no funcional 3: El sistema tiene que ser lo más modular posible de modo que añadir nuevas funcionalidades no represente ningún tipo de complejidad.

Diagrama de casos de uso.

Para poder cumplir con los requisitos funcionales de la aplicación se han definido los siguientes actores:

- **Usuario**: Sistema que permite enviar peticiones al dron para que ejecute ciertas acciones y obtener los datos de telemetría.
- **Pc abordo**: Sistema que recibe tareas del Usuario y actualiza el comportamiento del dron en base a las peticiones que se reciban, también obtiene los datos de telemetría del dron y los envía al Usuario.

Especificación de los casos de uso:

Ejecutar Misión:

Actores: Usuario y Pc abordo.

Descripción: Permite al Usuario cargar misiones en el dron y ejecutarlas una vez se han subido en el dron.

Flujo principal de eventos:

- El Usuario envía los puntos por los que tiene que pasar el dron y el comando específico para que pueda ejecutar la misión.

Hacer que el dron espere.

Actores: Usuario y Pc abordo.

Descripción: Permitir que el Usuario pueda pausar la ejecución del vuelo del dron en cualquier momento y se quede esperando hasta la siguiente tarea que le envié el Usuario.

Flujo principal de eventos:

- El Usuario envía el comando específico al pc abordo para que esta haga que el dron se ponga a esperar hasta la siguiente tarea que se le envíe.

Volver a la estación de despegue.

Actores: Usuario y Control Pc abordo.

Descripción: Permitir que el Usuario pueda hacer que el dron vuelva al pc abordo en cualquier momento.

Flujo principal de eventos:

- El Usuario envía el comando específico al pc abordo para que esta haga que el dron vuelva a la estación de despegue.

Pausar Misión.

Actores: Usuario y Pc abordo.

Descripción: Permitir que el Usuario pueda pausar la ejecución del vuelo del dron en cualquier momento y se quede esperando hasta la siguiente tarea que le envié el Usuario.

Flujo principal de eventos:

- El Usuario envía el comando específico al pc abordo para que esta haga que el dron se ponga a esperar hasta la siguiente tarea que se le envíe.

Obtener telemetría del dron.

Actores: Usuario y Pc abordo.

Descripción: Permitir que el Usuario pueda obtener los datos de telemetría obtenidos del dron.

Flujo principal de eventos:

- El Usuario se podrá comunicar con el dron para obtener los datos de telemetría del dron.

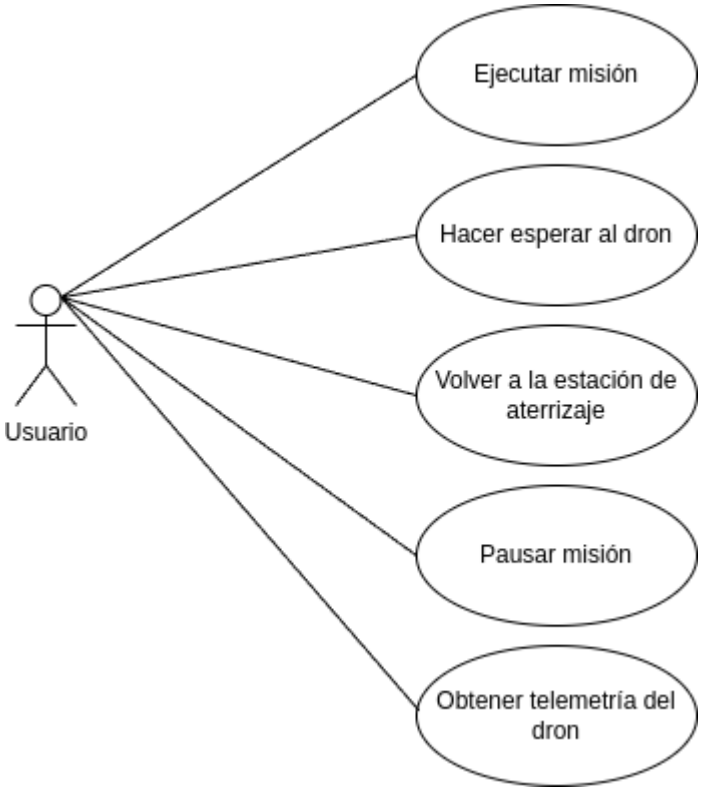

Ilustración 9. Diagrama de casos de uso.

5.2. Diagrama de secuencias.

Diagrama de secuencias para el envío de comandos:

Para cumplir los casos de uso de ejecutar misión, hacer esperar al dron, volver a la estación de aterrizaje y pausar misión. Hemos definido un mismo esquema general para cualquier interacción relacionada con el envío de comandos al dron para que realice alguna acción determinada.

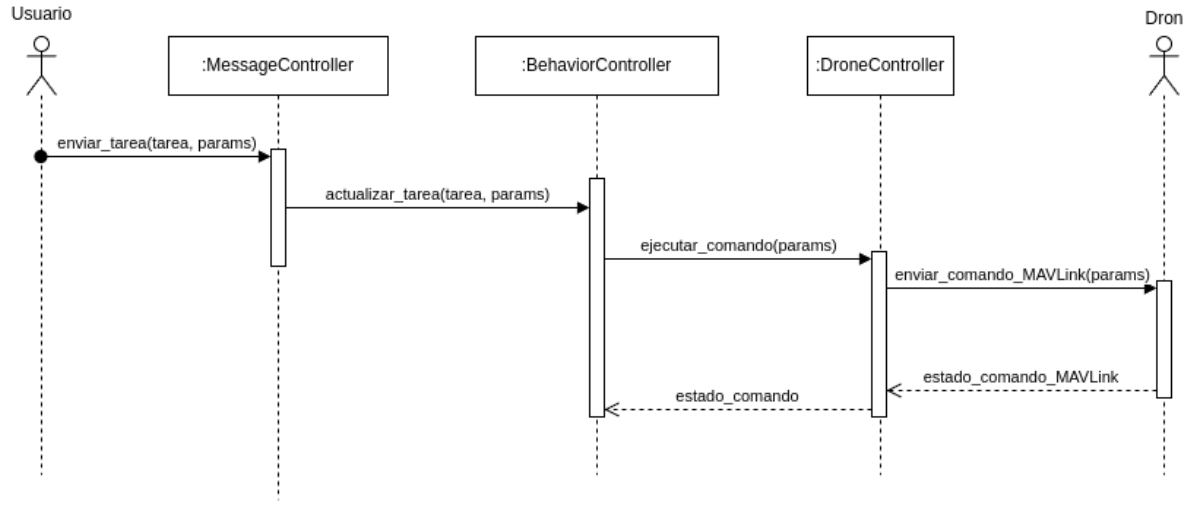

Ilustración 10.Diagrama de secuencias describiendo el envío de comandos.

Como se puede observar en la figura anterior el actor que inicia el flujo del diagrama de los comandos es el usuario, ejecutando el metodo enviar_tarea, el cual recibe como argumentos

tarea el cual será el nuevo estado del dron y los *params* que son los parámetros específicos que necesitan para ejecutar el nuevo estado.

Estos datos serán recogidos por la clase MessageController y posteriormente serán enviados por medio del método actualizar_tarea a BehaviorController para hacer la transición de un estado (los estados dentro de la aplicación son conocidos como tareas) a otro dentro de la aplicación.

Una vez se cambia el estado actual del objeto BehaviorController este se encarga de usar el método ejecutar_comando para notificar al objeto DroneController de que se quiere enviar un comando al dron y este se encargará de enviar el comando (los comandos son acciones que realiza el dron como despegar, esperar o ejecutar misión) deseado al Dron.

Diagrama de secuencias para la recepción de telemetría.

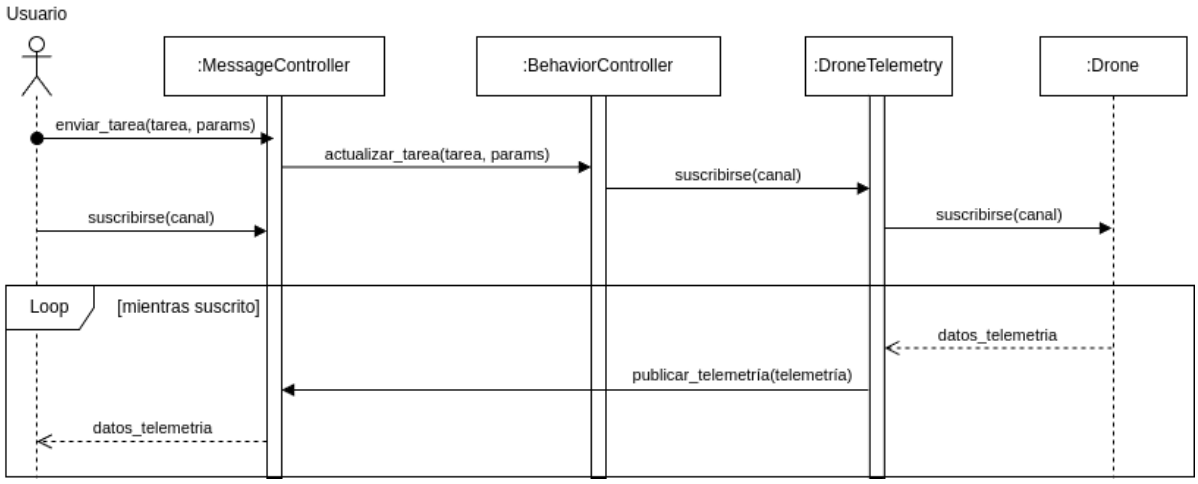

Ilustración 11. Diagrama de secuencias para la recepción de telemetría.

Para la recepción de telemetría lo primero que tendremos que hacer, es usar el método enviar_tarea al MessageController para que este actualice el objeto BehaviorController. Cada vez que se cambie de tarea al objeto BehaviorController notificara al objeto DroneTelemetry los canales de telemetría a los que este estado quiere subscribirse. Una vez notificado el objeto DronTelemetry, se crearán todas las suscripciones necesarias a los diversos canales de telemetría del dron que se le hayan pedido. Una vez creadas las suscripciones, el dron comenzara a enviar datos al objeto DroneTelemetry. Los datos que va recibiendo el objeto DroneTelemetry del dron, en cuanto los recibe, mediante la clase MessageController los publicará y posteriormente el usuario los podrá consultar cuando desee.

5.3. Tecnologías usadas.

Antes de comenzar el desarrollo de la aplicación, hubo un periodo de adaptación a las diferentes tecnologías que se usaran en este proyecto. Dicho periodo tuvo una duración de dos semanas y media. Aunque durante ese tiempo se intentó absorber la mayor cantidad de conceptos, muchos otros se adquirieron conforme se iba desarrollando la aplicación.

Cmake.

Cmake es una herramienta de código abierto diseñada para automatizar el proceso de compilación de software el cual cuenta con un lenguaje de scripting propio con el que puedes definir el proceso de construcción del proyecto.

Algunas de las grandes ventajas que nos proporciona Cmake es la creación de módulos reutilizables a lo largo de proyectos con una gran cantidad de ficheros, compilación multiplataforma lo cual nos permite compilar el código en cualquier tipo de sistema operativo compatible como bien podrían ser Linux o Windows, también nos permite la integración de librerías externas de una forma sencilla.

Se usará cmake para compilar todos los ficheros fuentes que se han creado para la aplicación, la búsqueda y el enlace de las diferentes librerías externas que usaremos como MAVSDK o ZMQ. También para crear módulos reutilizables, útiles para encapsular y reutilizar código de la manera más eficiente y sencilla posible.

src > CMakeLists.txt

```
set(SOURCES_DRONE
main.cpp
Exception.cpp
Exception.h
)

add_subdirectory(controller)
add_subdirectory(behavior)
add_subdirectory(connection)
add_subdirectory(telemetry)

add_executable(main ${SOURCES_DRONE})


target_link_libraries(main
    behavior
    connection
    controller
)


target_include_directories(main PUBLIC
${PROJECT_BINARY_DIR}
${PROJECT_SOURCE_DIR}
${PROJECT_SOURCE_DIR}/src
${PROJECT_SOURCE_DIR}/src/telemetry
${PROJECT_SOURCE_DIR}/src/connection
${PROJECT_SOURCE_DIR}/src/controller
${PROJECT_SOURCE_DIR}/src/behavior
)
```

Ilustración 12. Ejemplo de Cmake.

Cabe decir que, aunque se diga que cmake es una herramienta que nos permite compilar el código de una forma sencilla, realmente tiene una curva de aprendizaje bastante compleja debido a que su sintaxis suele ser bastante compleja de entender y su documentación no acaba de detallar de la manera más clara cómo se han de usar sus diferentes funciones.

C++.

Al decidir que lenguaje de programación usaríamos para desarrollar la aplicación buscamos cuáles tenían mejor compatibilidad con las tecnologías que tendríamos que usar en el proyecto.

Después de investigar en la documentación de las diferentes tecnologías que se usaran para este proyecto, decidimos que las que mejor se adaptaban a nuestras necesidades fueron cpp y python. Después de un análisis sobre los pros y contras que presentaba cada una de las librerías acabamos decidiendo que usaríamos cpp por los siguientes motivos.

Velocidad de ejecución.

Cpp al tratarse de un lenguaje compilado (lo que significa que el código fuente se traduce a código maquina antes de ejecutarse) suele ser más eficiente en tiempo de ejecución que no un lenguaje interpretado como python.

Orientación a objetos.

Que Cpp sea orientado a objetos hace que sea un lenguaje de programación más práctico para este tipo de proyecto, de modo que nos permite usar herencia y composición y con esto poder conseguir un código más extensible y modular.

Problemas de latencia por uso de wrappers.

Según la documentación y diversos foros de las tecnologías que usaremos, hemos visto que todas ellas son mucho más eficientes y rápidas en Cpp que no en python. Esto se debe a que la mayoría de las librerías fueron escritas en Cpp y para que pudieran ser usadas en lenguajes como python, se crean wrappers (un wrapper es una capa de enlace que se crea entre dos lenguajes de programación para que un lenguaje pueda usar código de otro). El problema principal de esto es que le añade latencia al programa y eso resulta en pérdidas de rendimiento.

Frecuencia de actualización.

Las librerías y tecnologías que se usaran están escritas en cpp y reciben actualizaciones más frecuentes en dicho lenguaje, además sus documentaciones suelen estar más completas y poseen más información que no en otras versiones.

MAVLink.

MAVLink es un protocolo de mensajería diseñado para la comunicación con drones y con todos los componentes que este tiene instalados en su sistema.[10]

MAVLink sigue un patrón de comunicación hibrido basado en publish-subscribe y point-to-point de los cuales se hablará con más detalle más adelante. Para él envió o publicación de telemetría se hace mediante streams de datos los cuales son publicados en tópicos. Para el control de misión o acciones que realiza el dron se usan retransmisiones point-to-point.

Cabe decir que MAVLink es un protocolo de comunicación bastante extenso. Aun así, tiene muchas librerías que permiten usar el protocolo sin tener que conocerlo.

Aun así, cabe destacar ciertos aspectos básicos del protocolo que han sido útiles de entender de cara a saber cómo este funciona internamente.

Canales de comunicación.

El protocolo utiliza diversos canales de comunicación para permitir el procesado paralelo de datos dentro de una misma aplicación. Para recibir y enviar datos se requiere de un canal especifico y es importante usarlos de forma correcta según el uso asignado.

Cabe decir que según el sistema operativo y las características del hardware con el que estemos trabajando podremos poseer cuantos canales sean necesarios, aunque por defecto en sistemas operativos Windows, Linux o MacOS por defecto tienen 16.[10]

Formato de paquete MAVLink 2.

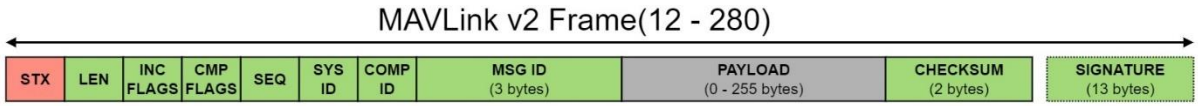

Ilustración 13. Formato de paquete MAVLink 2. ref [11]

Figura X. Frame MAVLink v2.

Índice de Byte	**Versión C**	**Contenido**	**Valor**	**Explicación Resumida**
0	uint8_t magic	Inicio de paquete	0xFD	Marca el inicio de un nuevo paquete.
1	uint8_t len	Longitud del payload	0 - 255	Longitud de la Payload
2	uint8_t incompat_flags	Flags de incompatibilidad		Flags que deben entenderse para la compatibilidad.
3	uint8_t compat_flags	Flags de compatibilidad		Flags que pueden ignorarse si no se entienden.
4	uint8_t seq	Número de secuencia	0 - 255	Detecta pérdida de paquetes.
5	uint8_t sysid	ID del sistema (remitente)	1 - 255	ID del sistema que envía el mensaje.
6	uint8_t compid	ID del componente	1 - 255	ID del componente que envía el mensaje.
7 a 9	uint32_t msgid:24	ID del mensaje	0 - 16777215	Identifica el tipo de mensaje.
n=0: N/A, n=1: 10, n≥2: 10 a (9+n)	uint8_t payload[max 255]	Payload		Contenido del mensaje.

(n+10) a (n+11)	uint16_t checksum	Suma de verificación		Verificación del mensaje.
(n+12) a (n+25)	uint8_t signature[13]	Firma		Firma opcional para seguridad.

Tabla 5. Especificación de los campos del frame MAVLink v2. ref [12].

Recepción de datos.

La recepción/decodificación de datos cuenta de dos fases:

- Parsear los streams de datos en objetos que representen cada paquete.
- Descodificar el mensaje contenido en el payload del mensaje en un struct definido en C con los diferentes campos definidos en el esquema XML.

Payload.

Los payloads son estructuras XML, las cuales poseen un formato especifico. La razón de esto es poder parsear los mensajes recibidos y poder guardar los datos obtenidos en estructuras C previamente establecidas. Un incorrecto uso de las estructuras XML puede conllevar a que el mensaje que se ha enviado se acabe descartando por no poseer el formato especificado. [13]

Para el envío de comandos se usa una estructura similar a la que se verá a continuación.

```
<enum name="MAV_CMD">
....
    <entry value="94" name="MAV_CMD_NAV_PAYLOAD_PLACE">
        <description>/description>
        <param index="1">Maximum distance to descend</param>
        <param index="2">Empty</param>
        <param index="3">Empty</param>
        <param index="4">Empty</param>
        <param index="5">Latitude (deg * 1E7)</param>
        <param index="6">Longitude (deg * 1E7)</param>
        <param index="7">Altitude (meters)</param>
</entry>
...
</enum>
```

Ilustración 14. Ejemplo XML de un comando. Ref [13]

Algunos de los campos más relevantes que podemos observar son:

- name: El nombre del struct C al que pertenece.
- description: Breve descripción del comando.
- param: En este campo es donde se ponen los parámetros que necesita el comando.
- index: índice del parámetro.

Para el envío de mensajes como telemetría se usa una estructura como la siguiente:

```
<message id="147" name="BATTERY_STATUS">
      <description>Battery information.</description>
      <field type="uint8_t" name="id" instance="true">Battery
ID</field>
      <field type="uint8_t" name="battery_function"
enum="MAV_BATTERY_FUNCTION">Function of the battery</field>
      <field type="uint8_t" name="type" enum="MAV_BATTERY_TYPE">Type
(chemistry) of the battery</field>

<field type="int16_t" name="current_battery" units="cA"
invalid="-1">Battery current</field>

<field type="int8_t" name="battery_remaining" units="%"
invalid="-1">Remaining battery energy</field>

</message>
```

Ilustración 15. Ejemplo XML de un mensaje. Ref [13]

Algunos de los campos relevantes que podemos destacar de este ejemplo son:

- message:
- name: Es un nombre simbólico y representativo del mensaje, no afecta al parseo de datos. Se usa para proveer una forma fácil de identificar de que tipo de mensaje se trata.
- id: identificador del mensaje.
- field:
- name: Se usa en código para saber a qué variable se le tiene que asignar ese valor.
- type: Tipo del dato que se está enviando.
- enum: en ciertas ocasiones en vez de enviar un solo dato, puedes enviar un dato compuesto por otros valores dentro de un struct.

Tener conocimientos básicos en la estructura de los mensajes y cómo se envían puede resultar interesante para los casos en que se busque personalizar alguna acción como un despegue, aterrizaje o también en casos en que usemos sensores sin soporte de las librerías que usamos. En estos casos se tendría que crear un mensaje customizado, implementar los diferentes parseadores y diseñar las estructuras XML dentro del autopiloto y la aplicación de pc abordo que estemos diseñando. [13]

MAVSDK.

Como ya vimos anteriormente MAVLink es un protocolo de mensajería de drones bastante ligero. Algo que se debe destacar de dicho protocolo es la complejidad y la larga cantidad de funcionalidades de las que este dispone. Esto hace que su documentación sea cuanto menos extensa.

Gracias a esto, surgieron diferentes librerías que buscaban diseñar una API que permitiera usar el protocolo sin la necesidad de tener grandes conocimientos sobre él protocolo MAVLink. Una de las librerías que más destaco fue MAVSDK.

MAVSDK es una colección de librerías diseñada para varios lenguajes de programación que permite interactuar con sistemas MAVLink enfocándose principalmente en facilitar el desarrollo de software de comunicación con drones.[14]

ZeroMQ.

ZeroMQ (también conocida como ZMQ, 0MQ o zmq) es una librería de mensajería de alto nivel de alto rendimiento diseñada para sistemas distribuidos y comunicación entre procesos. La librería brinda comunicación entre sockets haciendo uso de diferentes formas de transporte de datos en un mismo proceso o entre diferentes procesos, TPC y multicast.[15]

A parte de su alto rendimiento enviando y recibiendo datos, algunas características bastante interesantes de la librería son:

- Es universal en cuanto al lenguaje de programación, esto quiere decir que múltiples aplicaciones que usan zeromq pueden comunicarse entre ellas sin necesidad de adaptadores o algún elemento software que les permita entender los mensajes que se pasan entre ellas.
- Permite implementar múltiples patrones de mensajes como pub/sub o request/reply lo que brinda flexibilidad a la hora de desarrollar.
- Tiene una comunidad de código abierto bastante grande y activa.
- También permite transportar mensajes mediante diferentes protocolos como IPC, TPC, UDP o WebSocket.

BehaviorTreeCpp.

BehaviorTreeCpp.

BehaviorTreeCpp es una biblioteca de CPP diseñada para implementar arboles de comportamiento. Esta librería está diseñada para ser flexible, fácil de usar y rápida.

Arboles de comportamiento.

Los árboles de comportamiento son una forma de estructurar cambios de tareas en agentes autónomos como robots, entidades virtuales como NPC (personajes no jugables o de adorno) de un video juego.

Algunas de las consideraciones que se deberían tener respecto a los diagramas de estados es que los árboles de comportamiento se prefieren sobre los diagramas de estado en algunos contextos por su capacidad para manejar comportamientos complejos de forma más flexible y clara. Mientras los diagramas de estado pueden complicarse con numerosos estados y transiciones, los árboles de comportamiento permiten descomponer tareas en una estructura jerárquica, facilitando la comprensión, reutilización y expansión de los comportamientos. Esto es especialmente útil en inteligencia artificial y desarrollo de videojuegos, donde los sistemas deben ser tanto complejos como manejables.

Los árboles de comportamiento permiten la creación de sistemas complejos de comportamiento cuya ventaja es que son bastante modulares.

Como ya explicaremos más adelantes un árbol de comportamiento está compuesto por conjuntos de bloques, en donde cada bloque representa una acción única e independiente. Esto significa que estos bloques deben cumplir con la condición de ser tareas componibles, lo que quiere decir que deben poder ser usadas para construir otros comportamientos.[16]

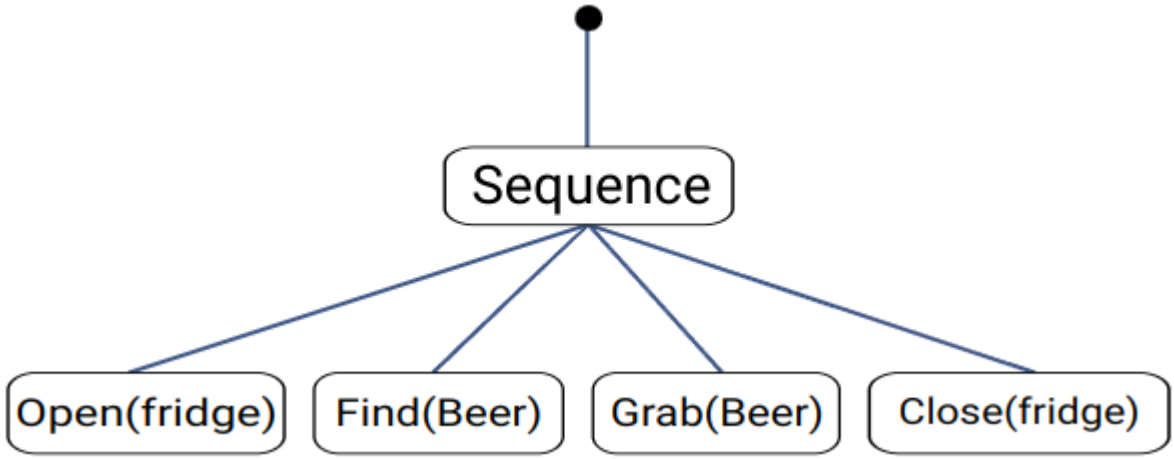

Ilustración 16. Ejemplo de árbol de comportamiento. Ref[17]

Funcionamiento de los árboles de comportamiento.

El flujo de ejecución del árbol de comportamiento comienza en el nodo más alto de toda la jerarquía denominado Root.

Root envía una señal denominada tick que se propaga a lo largo de todos los nodos hijos. Comienza desde el que está más a la derecha y se va propagando hacia los que están más a la izquierda de la estructura que se ha definido.[16]

Cada vez que un nodo del árbol recibe un tick hará que este ejecute su tarea y una vez finalice debe retornar uno de los siguientes estados:

- SUCCESS: Indica que la ejecución del nodo fue exitosa.
- FAILURE: Indica que algo ha fallado en la ejecución del nodo.
- RUNNING: Indica que el nodo necesita más tiempo para poder retornar un resultado.

Una vez retornado el estado, este se propaga a través de los nodos superiores y mediante los nodos de control (hablaremos de ellos más adelante) se toma una decisión sobre si pasar al siguiente hijo, volver a ejecutar los nodos anteriores o incluso parar la ejecución del árbol.

Una consideración importante para tener en cuenta es que la ejecución de todos los nodos del árbol de comportamientos se realiza de forma secuencial.[16]

Elementos de un árbol de comportamientos.

Behavior tree distingue principalmente entre 4 tipos diferentes de nodos:

- *ControlNode o nodos de control*: Pueden tener de 1 a N hijos. Normalmente ayuda a propagar los ticks basado en los resultados obtenidos de sus nodos hijos.
- *DecoratorNode o nodos decoradores*: Puede tener un solo hijo. El uso de estos es alterar los resultados de los estados devueltos por sus hijos y hacer que se ejecuten un cierto número de veces.
- *ConditionNode o nodos de condiciones*: No pueden tener hijos. Sirven para controlar condiciones del sistema (hacer alguna comprobación antes de continuar con la ejecución de los demás nodos).
- *ActionNode o nodos de acción*: No pueden tener hijos. Estos nodos son los que ejecutan las acciones del sistema.

BehaviorTreeCpp es una librería que cuenta con una gran cantidad de nodos de control, aunque para este proyecto se usaran Sequence y ReactiveSecuence.

Type of ControlNode	**Child returns FAILURE**	**Child returns RUNNING**
Sequence	Restart	Tick again
ReactiveSequence	Restart	Restart

- Restart significa que la secuencia se reiniciara desde el primer nodo hijo hasta el último.
- Tick again significa que la secuencia hará que se ejecute el mismo nodo de nuevo. Los nodos anteriores no son ejecutados de nuevo.[18]

Ventajas de los árboles de comportamiento.

Las ventajas principales de los árboles de comportamiento son[14]:

- Estructura jerárquica: Este tipo de estructura permite organizar comportamientos complejos de una manera estructurada. Esto permite descomponer las tareas complejas en subtareas más pequeñas. Por ejemplo, podríamos tener una tarea denominada “Buscar una Cerveza” y esta dividirla en tareas más pequeñas y concretas como “Agarrar objeto”, esto no solo nos permite dividir tareas complejas en nodos más cortos, sino que también nos permite reutilizar dichos componentes en otros árboles.
- Facilidad de comprensión: Los árboles de comportamiento se representa de una manera bastante intuitiva y fácil de entender. Cada nodo del árbol representa una acción o decisión especifica lo que facilita la lectura y la comprensión del flujo de comportamiento. En contraste con las máquinas de estados finitos en donde los estados y sus transiciones suelen ser difíciles de seguir y comprender tanto en su forma gráfica como textual debido a que sus estructuras suelen ser menos estructuradas y más dispersas.
- Son más expresivos: Los árboles de control son más expresivos gracias a la gran cantidad de nodos de los que dispone la librería. También, los usuarios tienen la posibilidad de crear sus propios nodos, lo que permite la creación de sistemas más modulares, personalizables y complejos.

Estructura de los nodos de acción.

En BehaviorTree Cpp tenemos dos tipos diferentes de acciones. Por un lado, tenemos las síncronas, que siempre devolverán de estado SUCCESS o FAILURE. Este tipo de nodos se suelen usar cuando la tarea que realiza el nodo no tarda mucho tiempo en ser ejecutada.

También tenemos las asíncronas, las cuales pueden retornar SUCCESS, FAILURE y también RUNNING. Este tipo de nodos se suele usar cuando se trata de tareas que requieren periodos largos de tiempo de ejecución.[19]

```
// Example of custom SyncActionNode (synchronous action)
// without ports.
class ApproachObject : public BT::SyncActionNode
{
public:
  ApproachObject(const std::string& name) :
      BT::SyncActionNode(name, {})
  {}

  // You must override the virtual function tick()
  BT::NodeStatus tick() override
  {
    std::cout << "ApproachObject: " << this->name() << std::endl;
    return BT::NodeStatus::SUCCESS;
  }
};
```

Ilustración 17. Ejemplo de nodo síncrono.ref[19]

Algo importante que podemos resaltar de los nodos síncronos es que poseen un único método el cual es tick(). Este método es el que se ejecuta cada vez que se recibe un tick y es el que contiene la lógica de la tarea.

```
class MoveBaseAction : public BT::StatefulActionNode
{
  public:
    MoveBaseAction(const std::string& name, const BT::NodeConfig& config)
      : StatefulActionNode(name, config)
    {}

    static BT::PortsList providedPorts()
    {
        return{};
    }

    // this function is invoked once at the beginning.
    BT::NodeStatus onStart() override;

    // If onStart() returned RUNNING, we will keep calling
    // this method until it return something different from RUNNING
    BT::NodeStatus onRunning() override;

    // callback to execute if the action was aborted by another node
    void onHalted() override;
};
```

Ilustración 18. Ejemplo de nodo Asíncrono.

En los nodos asíncronos, a diferencia de los nodos síncronos podemos observar tres nuevos métodos: [20]

- OnStart(): Este método se ejecuta únicamente cuando recibe el primer tick. Se usa para casos en los que pueda retornar directamente FAILURE o SUCCESS.
- OnRunning(): Si OnStart() retorna RUNNING, entonces se ejecutará este método hasta que pueda retornar FAILURE o SUCCESS.
- OnHalted(): Si un nodo previo provoca que falle toda la secuencia de control y aún quedan nodos bajo ese nodo de control que no se han ejecutado, hará que se ejecute este método.

Representación a bajo nivel de los árboles de comportamiento.

Para explicar los árboles de comportamiento se usan estructuras en forma de árbol para representar el flujo de tareas que seguirá un sistema. A bajo nivel es diferente puesto que un compilador no entiende de diagramas de árbol.

Para estructurar los flujos de control de un árbol de comportamiento se usan ficheros XML los cuales representan dichas tareas. Por lo que es normal primero realizar un diagrama en forma de árbol y después traducirlo a un fichero XML para que cuando se compile la aplicación el programa pueda saber que estructura de árbol se está usando.

Para ilustrar un ejemplo, se mostrará un diagrama en forma de árbol y su traducción a lenguaje XML.

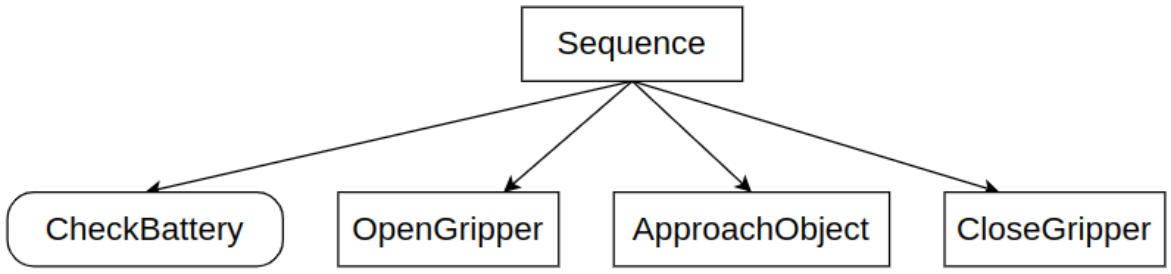

Ilustración 19. Diagrama de árbol de revisión de batería. Ref [19]

```
<root BTCPP_format="4" >
    <BehaviorTree ID="MainTree">
       <Sequence name="root_sequence">
           <CheckBattery   name="check_battery"/>
           <OpenGripper    name="open_gripper"/>
           <ApproachObject name="approach_object"/>
           <CloseGripper   name="close_gripper"/>
       </Sequence>
    </BehaviorTree>
</root>
```

Ilustración 20. Diagrama XML del diagrama de árbol de revisión de batería. Ref [19]

Como podemos ver en el diagrama anterior, la primera etiqueta de todo el diagrama es root, ya que este es el nodo principal de la aplicación y el padre de todos los nodos del árbol. Después tenemos un nodo de control Sequence. Este será el nodo que controle los estados de sus hijos y dentro están los nodos de condición CheckBattery usado para comprobar el estado de la batería y luego tres nodos de acción los cuales son usados para hacer que el sistema realice ciertas acciones definidas por cada nodo en cuestión.

Sistemas de memoria compartida entre nodos BLACKBOARD.

Una particularidad bastante útil de la librería BehaviorTreeCpp es que nos provee un espacio de memoria compartida en la cual se pueden definir cualquier tipo de datos que se quieran guardar dentro de ella y estos pueden ser accedidos por cualquiera de los nodos que se encuentren dentro del árbol de comportamiento. Cada árbol de comportamiento tiene su zona de memoria y no puede ser accedida por un nodo que no pertenezca a este, a no ser que lo especifiquemos dentro del esquema XML del árbol de comportamientos.

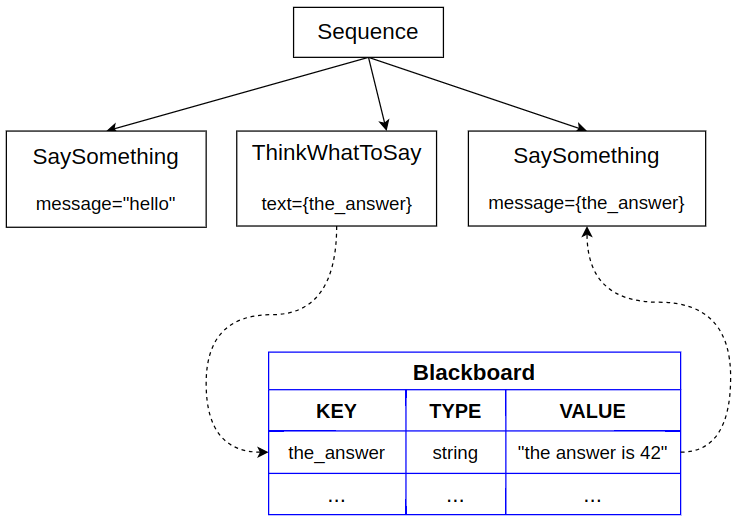

Ilustración 21. Ejemplo de BlackBoard.

Las entradas del blackboard son accedidas por lo nodos haciendo uso de puertos. Los puertos de un nodo son punteros a los datos del BlackBoard. BehaviorTreeCpp diferencia entre dos tipos de puertos. Están los InputPorts, con los cuales podemos acceder a la información para solo lectura. También tenemos OutputPorts los cuales son puertos de solo escritura.[21]

```
// It is mandatory to define this STATIC method.
static PortsList providedPorts()
{
  // This action has a single input port called "message"
  return { InputPort<std::string>("message") };
}
```

Ilustración 22. Ejemplo de declaración de puertos. Ref[21]

6. Diseño.

Para el diseño de la aplicación vamos a dividirla en tres sistemas. La aplicación de cliente, la aplicación del pc abordo y el dron.

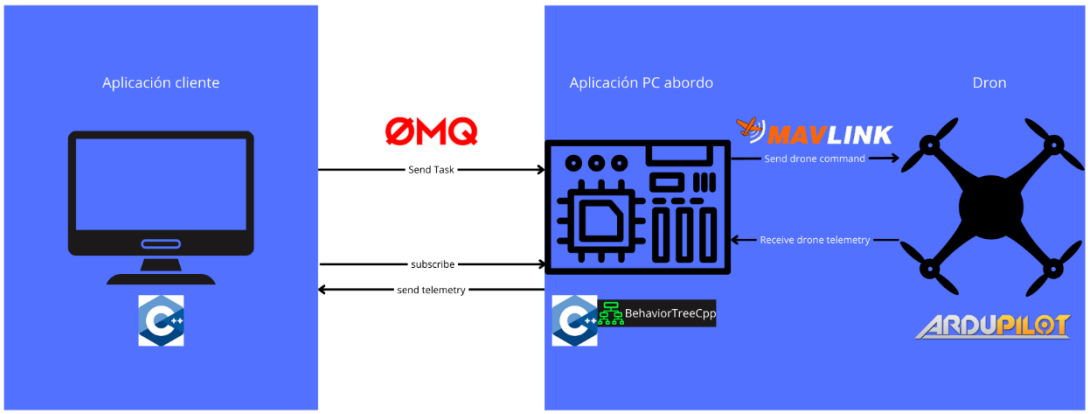

Ilustración 23. Arquitectura de la aplicación.

Aplicación Pc abordo.

El pc abordo es una aplicación desarrollada en cpp la cual se encargará de realizar la conexión con el dron mediante el protocolo de comunicación con drones MAVLink. Es la que se encarga de enviar comandos al dron tales como despegar, aterrizar, volver a la estación de despegue, subir y ejecutar misión, entre otros. También recibir la telemetría del dron y publicarla en tópicos haciendo uso de la librería ZMQ.

Dicha aplicación será ejecutada en una controladora mono placa Raspberry PI 5 modelo B gracias a su ligero tamaño y bajo consumo energético. Esta se conectará al dron haciendo uso de radio frecuencia, de modo que tanto el dron como la Raspberry han de tener un receptor y emisor de radiofrecuencia.

Para cada tipo de petición que pueda realizar el usuario sobre el dron, se establecerá una serie de estados que irán cambiando a medida que se vayan recibiendo peticiones. Para controlar los estados del dron y sus transiciones se usará la librería behaviorTreeCpp. Algunos de los estados que tendremos será el de ejecutar misión, esperar, volver a la estación de despegue, entre otros comandos útiles para el control del dron.

El PC a bordo también se encargará de recibir solicitudes del cliente y ajustará el estado del dron según lo que reciba. Para que el cliente pueda enviar solicitudes al dron, lo hará enviando

mensajes al PC a bordo utilizando la librería ZMQ. Las solicitudes que el cliente puede enviar se muestran en la siguiente tabla:

Tarea	Parámetros	Descripción
mision	item[]	Cuando el pc abordo recibe este mensaje comienza la ejecución del estado de ExecuteMission. En este estado se suben los items de la misión al dron, se despega en caso de aún no haberlo hecho y se comienza la ejecución de la misión. Una vez finaliza la misión retorna a la estación de despegue.
hold		Cuando el pc abordo recibe este mensaje comienza la ejecución del estado de Loiter. En este estado se le envía un comando al dron para que este se detenga en el punto en el que se encuentra en ese momento.
return		Cuando el pc abordo recibe este mensaje comienza la ejecución del estado de ReturnToLaunch. En este estado el dron vuelve a la estación de despegue.
pause		Cuando el pc abordo recibe este mensaje comienza la ejecución del estado de Pause. En este estado el dron se queda a la espera de una nueva misión.

Tabla 6. Tipos de mensaje que recibe el pc abordo.

Los ítems son los puntos por los que tiene que pasar el dron. Cada ítem se compone de latitud, longitud, altura relativa, altura absoluta, velocidad, pitch y jaw.

Nombre	Tipo	Descripción
Latitud.	Float.	Latitud a la que se quiere enviar al dron.
Longitud.	Float.	Longitud a la que se quiere enviar al dron
Altura relativa.	Double.	Altura relativa al punto de despegue.
Altura absoluta.	Double.	Altura relativa al nivel del mar
Velocidad.	Float.	Velocidad a la que ha de ir hasta alcanzar ese punto
Pitch.	Float.	Movimiento que puede hacer el dron hacia arriba o hacia abajo para poder avanzar
Jaw.	Float.	Movimiento permite que el dron gire sobre su eje en forma circular

Tabla 7. Elementos que componen al tipo ítem.

Aplicación de prueba de cliente.

La aplicación cliente consiste en dos pequeñas aplicaciones separada. La primera es un pequeño programa escrito en cpp que permite enviar peticiones por medio de la librería ZMQ al pc abordo para poder cambiar el estado del dron.

La segunda aplicación está escrita en cpp y permite conectarse a los canales de telemetría en los que el pc abordo publica los datos mediante ZMQ. Los canales de telemetría de los cuales dispondremos son los siguientes:

Canal.	Parámetros.	Tipo.	Descripción.
[Mission]	actual-total	int-int	Recibimos la cantidad de ítems por los cuales ha pasado el dron y también el total de items.
[State]	estado	string	Recibimos un string que nos informa sobre la acción actual que este ejecutando el dron.
[Health]	salud-estado_arm	boolean-boolean	Nos permite saber si el dron se encuentra en buenas condiciones y si está preparado para despegar.
[GPS]	latitud-longitud-altura_relativa-altura_absoluta	float-float-double-double	Nos permite saber la posición actual del dron.

Tabla 8. Canales de telemetría en los que publica datos el pc abordo.

Los estados de telemetría nos informan de que acciones está ejecutando actualmente el dron. Los principales estados de los que disponemos son:

Estado	Descripción
Hold.	El dron se encuentra quieto esperando una nueva tarea.
ReturnToLaunch.	El dron se encuentra volviendo a la estación de despegue.
TakeOff.	Nos informa de que el dron está despegando.
Mission	Nos informa de que el dron está ejecutando una misión

Tabla 9. Estados que pueden ser recibidos por el canal [State].

Dron.

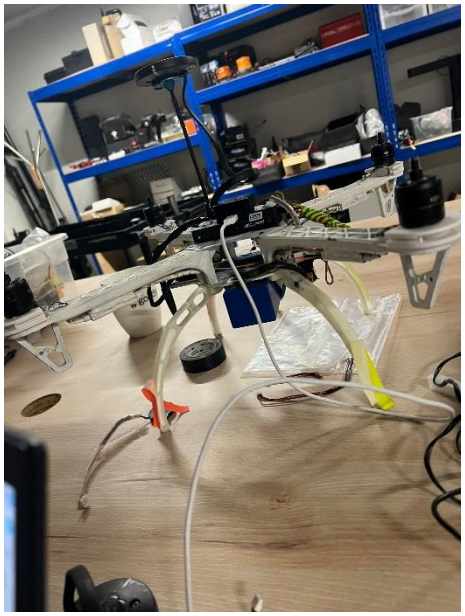

Ilustración 24. Dron de pruebas.

Para el dron usamos una placa CubePilot Cube Black la cual será configurada con el software firmware de ardupilot. Esta se encargará de autogestionar los diferentes sensores y actuadores que hemos instalado en el sistema. En nuestro caso tiene instalados 4 motores para las hélices, un sensor gps, una batería, entre otros.

Cabe decir que ardupilot es compatible con el protocolo de comunicación MAVLink, de modo que puede recibir los mensajes, parsearlos y ejecutar comandos que se le envíen u obtener información que se le esté enviando. También es capaz de envolver datos en mensajes XML de MAVLink y enviarlos al pc abordo.

7. Implementación.

Para llevar a cabo el desarrollo de la aplicación del pc abordo, la cual se encargará de enviar comandos MAVLink al dron y enviar y recibir datos al cliente, se ha seguido el siguiente flujo de trabajo.

Diseño de diagrama de clases.

Para realizar la implementación de la aplicación lo primero que se realizó fue el siguiente diagrama de clase, el cual se usara de referencia para poder llevar a cabo el desarrollo de la aplicación.

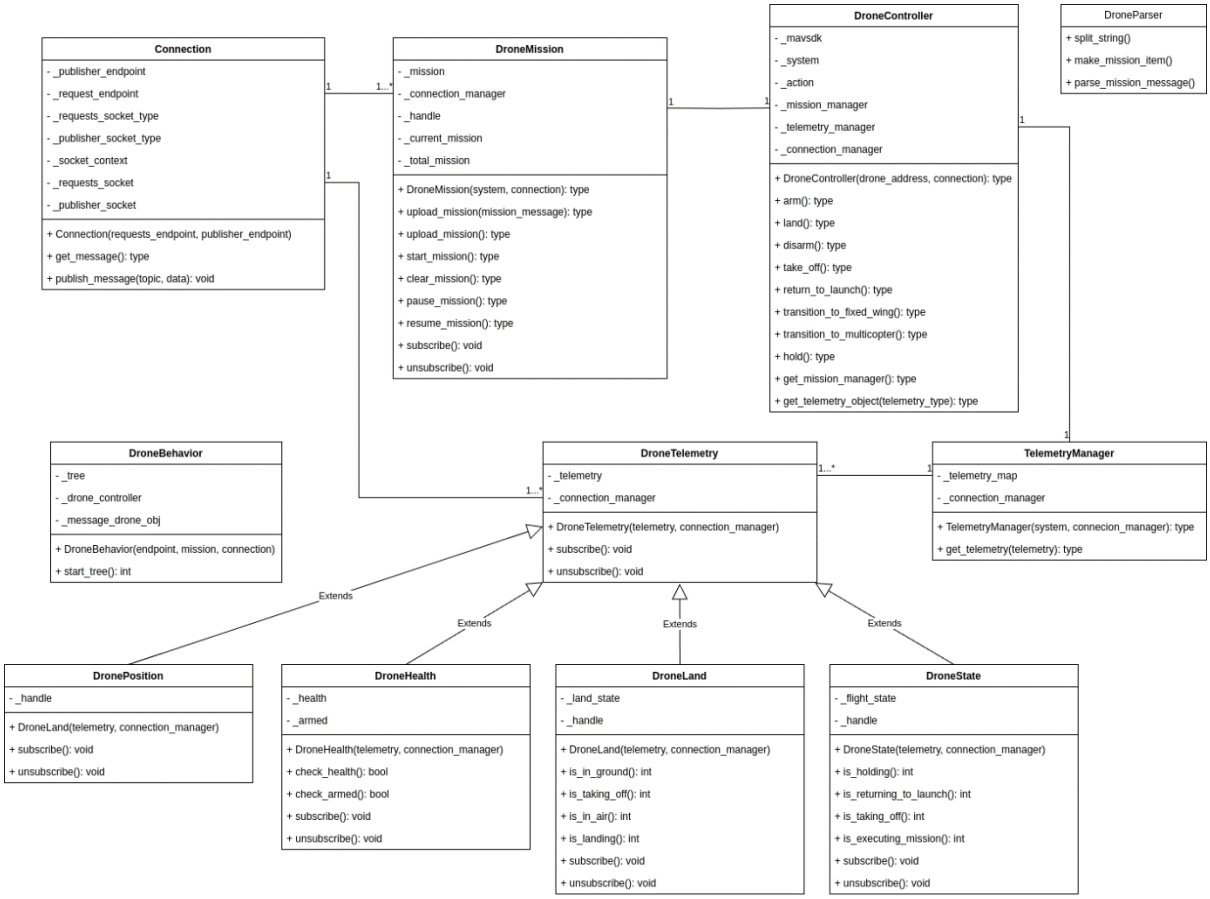

Ilustración 25. Diagrama de clases de la aplicación pc abordo.

A continuación, se hace una breve descripción del propósito de cada clase.

- Connection: La clase Connection se encargará de la recepción de mensajes y también de la publicación de datos en canales a los que se pueda subscribir el cliente.
- DroneController: Esta clase se encarga de establecer conexión con el dron y también de enviarle comandos.
- DroneMission: La clase DroneMission se encarga de enviar misiones al dron, hacer que este las comience a ejecutar, pausarlas, cancelarlas y también de seguir el progreso de estas.

- DroneParser: La clase drone parser se encarga de parsear los datos obtenidos en los mensajes recibidos desde el cliente.
- DroneTelemetry: La clase DroneTelemetry contiene todos los métodos generales para las clases de telemetría.
- DronePosition: La clase DronePosition hereda de DroneTelemetry y se encarga de obtener y publicar los datos de posición recibidos desde el dron al canal de telemetría [Positon].
- DroneHealth: La clase DroneHealth hereda de DroneTelemetry y se encarga de obtener y publicar los datos de salud recibidos desde el dron al canal de telemetría [Health].
- DroneLand: La clase DronePosition hereda de DroneTelemetry y se encarga de obtener y publicar los datos de los estados del dron en el momento de aterrizar recibidos desde el dron al canal de telemetría [Land].
- DroneState: La clase DronePosition hereda de DroneTelemetry y se encarga de obtener y publicar los datos del estado actual del dron recibidos desde el dron al canal de telemetría [State].
- TelemetryManager: La clase TelemetryManager se encarga de instanciar las clases de telemetría y guardarlas es una estructura de datos.
- DroneBehavior: La clase DroneBehavior es la clase que se encarga de manejar todos los estados de la aplicación.

Cabe decir que esta breve introducción se ira expandiendo según se vayan explicando la aplicación.

Creación entorno de trabajo.

Primero se ha creado el fichero de compilación principal de la aplicación. En este definimos el punto de entrada del proyecto.

```
cmake_minimum_required(VERSION 3.20)
project(MAVLINKDRONE VERSION 1.0)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
add_subdirectory(src)
```

Pseudocódigo. fichero principal de compilación del proyecto.

Después se crean los dos directorios principales de proyecto. src/ es el directorio que contiene todos los ficheros y directorios fuentes de la aplicación. build/ es el directorio que contiene todos los ficheros compilados de la aplicación.

Creación del directorio src/.

En este directorio se crearon las carpetas que contendrán los diferentes módulos de la aplicación. En concreto se han creado 4 directorios.

- Controller/ contiene todo el código relacionado con el control del dron, por ejemplo, cómo establecer conexión, subir, ejecutar y monitorizar misiones.

- Telemetry/ contiene todos los ficheros que desde los que se obtienen los datos de telemetría de los diferentes elementos que tiene el dron como el gps, el estado de los sensores y actuadores, el estado de la batería.
- Connection/ contiene los ficheros relacionados con las comunicaciones con el exterior, como recibir mensajes del cliente u obtener datos de telemetría.
- Behavior/ contiene todos los ficheros relacionados con el control del comportamiento del dron y la transición entre las diferentes tareas o estados.

También se crea el fichero main.cpp. Este es el fichero de partida de la aplicación. Contiene un hilo principal de ejecución el cual se encarga de manejar los estados del dron. Luego un thread que se encarga de recibir mensajes del cliente y notificar al hilo principal que se ha recibido una petición.

Se crea también el fichero CMakelists.txt de este directorio. En este se incluyen los directorios de los módulos creados y se configura el fichero desde el que se creará el ejecutable de la aplicación.

```
set(SOURCES_DRONE
main.cpp
)
add_subdirectory(controller)
add_subdirectory(behavior)
add_subdirectory(connection)
add_subdirectory(telemetry)
add_executable(main ${SOURCES_DRONE})
target_link_libraries(main
behavior
connection
controller
)
target_include_directories(main PUBLIC
${PROJECT_BINARY_DIR}
${PROJECT_SOURCE_DIR}
${PROJECT_SOURCE_DIR}/src
${PROJECT_SOURCE_DIR}/src/telemetry
${PROJECT_SOURCE_DIR}/src/connection
${PROJECT_SOURCE_DIR}/src/controller
${PROJECT_SOURCE_DIR}/src/behavior
)
```

Pseudocódigo. Fichero Cmakelists.txt del directorio src/.

Creación del módulo connection.

Para la creación del módulo de connection se realizó la siguiente configuración en el fichero CMakelists.txt:

```
set(CONNECTION_SOURCES
${CMAKE_CURRENT_SOURCE_DIR}/Connection.cpp
```

```
${CMAKE_CURRENT_SOURCE_DIR}/Connection.h
)
set(ZeroMQ_DIR
/home/nico/Public/libzmq/build
)
add_library(connection
${CONNECTION_SOURCES}
)
find_package(ZeroMQ REQUIRED)
find_library(CMAKE_LIBRARY_ZMQPP
NAMES zmqpp
HINTS /home/nico/Public/zmqpp/build
NO_DEFAULT_PATH
)
target_include_directories(connection
PUBLIC /home/nico/Public/zmqpp/src/zmqpp
)
target_link_libraries(connection
PUBLIC zmq
${CMAKE_LIBRARY_ZMQPP}
)
```

Pseudocódigo. Fichero de configuración CMakeLists.txt del módulo connection.

Con este fichero primero se definen los ficheros fuentes que dispone el módulo. Después se especifica la función del módulo como una librería y se le asigna un nombre. Una vez categorizado el módulo, se buscan las librerías que necesite y se vinculan a este. Para este módulo usaremos la librería externa zmq.

Implementación del módulo connection.

El módulo de connection se encarga de recibir mensajes del cliente y publicar datos en canales de telemetría a los cuales el cliente se podrá suscribir y acceder. Este consiste temporalmente de una sola clase llamada Connection.

Implementación del constructor de la clase Connection.

```
Connection::Connection(std::string request_endpoint, std::string publisher_endpoint) :
_requests_socket(_context, _requests_socket_type),
_publisher_socket(_context, _publisher_socket_type)
{

// Create socket connection for requests.
_request_endpoint = request_endpoint;
zmqpp::socket_type type = zmqpp::socket_type::pull;
_requests_socket.bind(_request_endpoint);

// Create socket connection for publishing messages.
_publisher_endpoint = publisher_endpoint;
```

```
zmqpp::socket_type publisher_type = zmqpp::socket_type::publish;
_publisher_socket.bind(_publisher_endpoint);
}
```
Pseudocódigo. Constructor de la clase Connection.

En el constructor de la clase Connection se crean dos sockets: uno es de tipo “pull” para poder recibir mensajes. El otro de tipo publish el cual se usará publicar datos en los diferentes canales de telemetría.

Los dos sockets hacen un “bind” al puerto pasado por parámetro y se quedan a la espera de recibir datos y de publicarlos. Para poder hacerlo se han definido los siguientes métodos:

```
DroneMessageType Connection::get_message()
{
DroneMessageType incomming_message;

// Receieve and return message.
zmqpp::message message;
_requests_socket.receive(message);
message >> incomming_message.task >> incomming_message.items;
return incomming_message;
}
```
Pseudocódigo. Función para recibir mensajes.

La función get_message se usa para recibir mensajes de forma bloqueante. Cada vez que se ejecuta el método se llama a la función de la librería zmqpp receive, la cual espera hasta que le llegue un mensaje del cliente. Una vez lo recibe se descompone en dos elementos, un task y otro items. Task es un parámetro el cual se usará para cambiar el comportamiento del dron y otro items el cual contiene los datos de la misión.

Para enviar publicar datos se ha implementado la función publish_message. Esta recibe dos parámetros, topic y data. El parámetro topic define el canal en el que se publicaran los datos y el parámetro data son los datos que se quieren publicar. Para publicar los datos se usa el método send de zmq y como parámetro se concadenan tanto el topic como el data.

```
void Connection::publish_message(std::string topic, std::string data)
{
// Publish data to a especific topic.
_publisher_socket.send(topic + data);
}
```
Pseudocódigo. Función publish_message de la clase Connection.

Cabe decir que en zmq los canales van identificados por la primera secuencia de caracteres del mensaje que se está publicando. Por tanto, si nos suscribimos al canal “AB” y se publican los datos “ABC” y “ABD” desde el publicador, entonces recibiremos datos de los dos canales anteriores, ya que la primera secuencia “AB” coincide con la primera secuencia de los nuevos mensajes.

Implementación del thread de recepción de datos.

```
void message_manager(std::shared_ptr<DroneMessageType> drone_message,
std::shared_ptr<Connection> connection)
{
while (true)
{
auto message = connection.get()->get_message();
*drone_message = message;
}
}
```
Pseudocódigo. Thread de recepción de mensajes.

El Thread de recepción de mensajes message_manager se encarga de hacer la recepción de mensajes del cliente. Este recibe por parámetro dos punteros. drone_message es un puntero que comparten el hilo principal de ejecución y el thread. Su finalidad es que este thread sea capaz de enviar los datos de las nuevas request al hilo principal. Por otro lado, connection es un objeto de tipo Connection el cual nos permite mediante su metodo get_message() obtener mensajes del cliente.

Creación del módulo controller.

Para crear el módulo controller se ha tenido que realizar la siguiente configuración en el fichero CMakelists.txt:

```
set(DRONE_CONTROLLER_SOURCES
${CMAKE_CURRENT_SOURCE_DIR}/DroneController.cpp
${CMAKE_CURRENT_SOURCE_DIR}/DroneController.h
${CMAKE_CURRENT_SOURCE_DIR}/DroneMission.cpp
${CMAKE_CURRENT_SOURCE_DIR}/DroneMission.h
${CMAKE_CURRENT_SOURCE_DIR}/DroneParser.cpp
${CMAKE_CURRENT_SOURCE_DIR}/DroneParser.h
)
add_library(controller ${DRONE_CONTROLLER_SOURCES})
find_package(MAVSDK REQUIRED)
target_link_libraries(controller
MAVSDK::mavsdk
telemetry
connection
)
```
Pseudocódigo. Fichero de configuración CMakeLists.txt del módulo controller.

Con este fichero definimos cuales son los ficheros fuente del módulo, Se agrega como librería dentro del proyecto y se especifican sus ficheros fuente. Y se vinculan las librerías de telemetry, mavsdk y connection.

Implementación del módulo controller.

El módulo de controller se encarga de establecer conexión con el dron, enviar comandos y hacer seguimiento de las misiones. Para realizar la implementación de este módulo se creó la clase Drone Controller.

Implementación del constructor de la clase Drone Controller.

Drone Controller es la clase se encarga de establecer conexión con el dron y enviarle comandos. Esta clase importa la librería mavsdk la cual nos brinda diferentes clases como por ejemplo System o Action, las cuales son librerías necesarias para poder interactuar con el dron

```
DroneController::DroneController(std::string source_address, std::shared_ptr<Connection>
connection_manager)
{
auto config = mavsdk::Mavsdk::Configuration(mavsdk::Mavsdk::ComponentType::GroundStation);
_mavsdk = std::make_shared<mavsdk::Mavsdk>(config);

Auto connetion_result = _mavsdk.get()->add_any_connection(source_address);

if (connetion_result != mavsdk::ConnectionResult::Success)
{
throw DroneException("Couldn't succesfully stablish connection with drone, try again");
}

while (_mavsdk.get()->systems().size() == 0)
{
std::this_thread::sleep_for(std::chrono::seconds(1));
}
auto system = _mavsdk.get()->systems()[0];

if (!system)
{
throw DroneException("Time out waiting for system, try connecting again");
}
_system = system;
...
}
```

Pseudocódigo. Constructor de DroneController.

El constructor de la clase DroneController recibe dos parámetros source_address y connection_manager. El parámetro source_address es la dirección de red o el path del dispositivo de radio frecuencia que esté conectado al pc abordo.

Connection	URL Format
UDP	udp://[Bind_host][:Bind_port]
TCP	tcp://[Server_host][:Server_port]
Serial	serial://[Dev_Node][:Baudrate]

Tabla 10. Formato de la source_address permitidas por el constructor DroneController.

Cuando se ejecuta el constructor, se lleva a cabo una configuración inicial que permite a mavsdk identificar el tipo de dispositivo que estamos desarrollando. Por ejemplo, en este caso, se está desarrollando una estación de control. Si la conexión se realiza a través de TCP o UDP, el sistema busca al dron en la red local, mientras que, si se trata de una conexión serial, se

identifica el dispositivo que se usará para comunicarse con el dron, que normalmente es un equipo de radiofrecuencia. Una vez que se establece la conexión, mavsdk se encarga de gestionar la comunicación entre el dron y la computadora a bordo utilizando el protocolo MAVLink.

Implementación de las funciones del Drone Controller.

Las funciones implementadas en esta clase son las relacionadas con el envío de comandos al dron para que ejecute ciertas acciones. Algunas de estas son despegar, volver a la estación de despegue, aterrizar, entre otras. Para poder llevar a cabo esto se ha de mandar el mensaje XML correspondiente al dron para que este lo pueda parsear y posteriormente ejecutar como ya se había explicado en el apartado 5.4.3 en donde hablamos de cómo están definidos de forma general todos los mensajes de comandos que se pueden usar para enviar acciones al dron. Gracias a la API que nos proporciona MAVSDK se podrán enviar mensajes sin tener la necesidad de definir dichos XML para enviar los mensajes puesto que la librería se encarga de esto.

Antes de poder realizar las acciones usando la librería mavsdk se han de instanciar ciertos objetos. Estos son creados en el momento en el que se ejecuta el constructor de la clase Dron Controller, por lo que este no solo crea la conexión con el dron, sino que también instancia y guarda los objetos necesarios para poder comunicarse con este.

```
auto system = _mavsdk.get()->systems()[0];
_action = std::make_shared<mavsdk::Action>(system);
```

Por un lado, tenemos el objeto System el cual mantiene persistente la conexión con el dron y por el otro, tenemos el objeto Action, el cual nos permite enviar los comandos al dron. Ambos objetos son guardados como miembros de la clase Drone Controller. El objeto System y Acion se guardan puesto que si se destruye el primero se perderá la conexión con el dron, y el segundo para poder enviar los comandos al dron.

```
DroneController::ActionStatus DroneController::arm()
{auto request_status = _action.arm();
deal_with_request_state(request_status);
}
DroneController::ActionStatus DroneController::take_off()
{
auto request_status = _action.takeoff();
deal_with_request_state(request_status);
}
DroneController::ActionStatus DroneController::return_to_launch()
{
auto request_status = _action.return_to_launch();
deal_with_request_state(request_status);
}
DroneController::ActionStatus DroneController::land()
{
auto request_status = _action.land();
```

```
deal_with_request_state(request_status);
}
DroneController::ActionStatus DroneController::disarm()
{
auto request_status = _action.disarm();
deal_with_request_state(request_status);}
void DroneController::hold()
{
_action.hold();
deal_with_request_state(request_status);}
```

Pseudocódigo. funciones de envió de comandos al dron.

Como se puede observar en este código, para enviar comandos al dron, solo hace falta llamar al método que lo represente y la librería mavsdk se encarga de empaquetar los datos necesarios para enviar el comando al dron y enviarlo por medio de los canales de telemetría de MAVLink.

El código anterior define todos los comandos que se enviaran al dron. Más concretamente tenemos 6:

- Arm: la función arm se encarga de preguntar al dron si el estado de todos su sensores y actuadores están listo para despegar y en caso de ser así, pone en movimiento las hélices del dron.
- Disarm: Esta función se encarga de decirle al dron que apague sus sensores y actuadores.
- Hold: Esta función le dice al dron que se quede en un punto esperando hasta nuevo aviso.
- Return to launch: Esta función le dice al dron que vuelva a la estación de despegue.
- TakeOff: Esta función le dice al dron que comience el despegue.
- Land: Esta función se usa para hacer que el dron aterrice.

Implementación de la clase Drone Mission.

Para reducir las responsabilidades de la clase Drone Controller se crea la clase Drone Mission. Esta se encarga de enviar misiones al dron, las ejecuta y sigue su estado.

Para poder enviar las misiones al dron y ejecutarlas se necesita un objeto de la librería mavsdk de tipo Mission. Este objeto es creado en el constructor de la clase y es guardado como miembro de la clase Drone Mission en la variable _mission.

```
DroneMission::DroneMission(std::shared_ptr<mavsdk::System>                    system,
std::shared_ptr<Connection> connection_manager)
{
_mission = std::make_shared<mavsdk::Mission>(system);
...
}
```

Pseudocódigo. Constructor de la clase Drone Mission.

Para poder subir misiones al dron se usa la función upload_mission. Esta recibe por parámetro un String el cual contiene toda la información de la misión. Las misiones están compuestas por una lista de datos denominados ítems, cada ítem contienen la latitud, longitud, altitud relativa, velocidad, el pitch y el jaw de cada uno de los sitios por los que tiene que pasar el dron. De modo que una misión se puede ver como una lista de ítems.

```
DroneMission::MissionState DroneMission::upload_mission(std::string mission_message)
{
auto mission_plan = parse_mission_message(mission_message);
auto status_request = _mission.get()->upload_mission(mission_plan);
...}
```

Pseudocódigo. Código de la función upload_mission de la clase Drone Mission.

La función upload_mission de la librería de mavsdk recibe como parámetro un objeto MissionPlan de la misma librería. Para poder tener la misión en dicho formato se ha implementado el siguiente parseador:

```
mavsdk::Mission::MissionPlan parse_mission_message(std::string mission)
{
std::vector<std::string> mission_string_list = split_string(mission, "\n");

std::vector<mavsdk::Mission::MissionItem> mission_items;

mavsdk::Mission::MissionPlan mission_plan{};

for (std::string mission_string : mission_string_list)
{
std::vector<std::string> mission_data = split_string(mission_string, " ");
auto mission_item = make_mission_item(mission_data);
mission_items.push_back(mission_item);
}

mission_plan.mission_items = mission_items;
return mission_plan;
}
```

Pseudocódigo. Función para parsear misiones.

La función parse_mission_message recibe un String por parámetro con todos los datos sobre la misión que se quiere cargar en el dron y devuelve un objeto MissionPlan con todos los datos de la misión que se quiere enviar parseados.

Para la implementación de los métodos que se encargan de ejecutar misiones y pausarlas se han desarrollado las siguientes funciones.

```
DroneMission::MissionState DroneMission::start_mission()
{
auto request_status = _mission.get()->start_mission();
}
```

```
DroneMission::MissionState DroneMission::pause_mission()
{
auto request_status = _mission.get()->pause_mission();
}
```
Pseudocódigo. Funciones para empezar y pausar misiones.

La función start_mission usa la función de la librería mavsdk start_mission para que empiece a ejecutar la misión que haya cargada actualmente en el dron. Para pause_mission se hace algo similar, se usa el método pause_mission de la librería mavsdk para poder pausar la ejecución de la misión que este realizando el dron en ese momento.

Para poder hacer el seguimiento del progreso de las misiones se ha implementado el siguiente método:

```
void DroneMission::subscribe()
{
_handle = _mission.get()->subscribe_mission_progress([this](mavsdk::Mission::MissionProgress
progress)
{
_current_mission = progress.current;
_total_mission = progress.total;
std::string data = std::to_string(_current_mission) + "-" + std::to_string(_total_mission);
_connection_manager.get()->publish_message("[Mission]:", data);
});
}
```
Pseudocódigo. Función de suscripción de la clase Dron Mission.

Mavsdk implementa también métodos con los cuales puedes suscribirte a los canales de telemetría. Esto se hace mediante los métodos subscribe de esta librería. En este caso usamos el método subscribe_mission_progress, este recibe como argumento un callback el cual será ejecutado cada vez que obtengamos datos por el canal al que nos hayamos suscrito, en este caso, en el del progreso de la misión.

Es importante mencionar que cada vez que se recibe un mensaje del dron, la propia librería se encarga de analizar el XML que contiene los datos de la telemetría. Los datos se almacenan en estructuras específicas dentro del módulo de telemetría. Por ejemplo, en el caso del canal que monitorea el progreso de las misiones, los datos se guardan en la estructura llamada MissionProgress.

Para acceder a los datos que recibimos tenemos que usar el callback que le pasamos como argumento a la función de subscribe_mission_progress. Dentro de esta función podemos tratar los datos que recibimos y usarlos como sea necesario.

Para enviar los datos sobre el progreso de la misión al cliente se usa el objeto _connection_manager de tipo Connection. Usamos su método publish_message para poder publicar los datos del progreso de la misión en el canal de [Mission] y pasamos como datos el progreso actual de la misión.

Creación del módulo telemetry.

El módulo de telemetry se encarga de obtener los datos del dron haciendo suscripciones a sus canales de telemetría mediante el uso de la librería mavsdk. Cada módulo tiene diversas clases y cada una representa alguna acción como aterrizaje o sensores concretos como el GPS. Para empezar el desarrollo de este módulo primero se implementó su fichero Cmakelists.txt.

```
set(TELEMETRY_SOURCES
${CMAKE_CURRENT_SOURCE_DIR}/TelemetryManager.cpp
${CMAKE_CURRENT_SOURCE_DIR}/TelemetryManager.h
${CMAKE_CURRENT_SOURCE_DIR}/DronePosition.cpp
${CMAKE_CURRENT_SOURCE_DIR}/DronePosition.h
${CMAKE_CURRENT_SOURCE_DIR}/DroneHealth.cpp
${CMAKE_CURRENT_SOURCE_DIR}/DroneHealth.h
${CMAKE_CURRENT_SOURCE_DIR}/DroneLand.cpp
${CMAKE_CURRENT_SOURCE_DIR}/DroneLand.h
${CMAKE_CURRENT_SOURCE_DIR}/DroneState.cpp
${CMAKE_CURRENT_SOURCE_DIR}/DroneState.h
${CMAKE_CURRENT_SOURCE_DIR}/DroneTelemetry.h
)
add_library(telemetry ${TELEMETRY_SOURCES})
find_package(MAVSDK REQUIRED)
target_link_libraries(telemetry
connection
MAVSDK::mavsdk)
```

Pseudocódigo. fichero CMakelists.txt del módulo de telemetría.

Lo primero que se hace en este fichero es decir cuáles son los ficheros fuentes de este módulo. Se agrega como librería dentro del proyecto y se pasan los ficheros fuentes que estarán dentro de esta. Luego se vinculan las librerías que se usaran en este módulo que son la de connection y la de mavsdk.

Implementación del módulo telemetry.

```
class DroneTelemetry
{
protected:
std::shared_ptr<mavsdk::Telemetry> _telemetry;
std::shared_ptr<Connection> _connection_manager;
public:
DroneTelemetry(const std::shared_ptr<mavsdk::Telemetry> telemetry,
std::shared_ptr<Connection> connection_manager)
{
_telemetry = telemetry;
_connection_manager = connection_manager;
}
~DroneTelemetry() {}
virtual void subscribe () = 0;
virtual void unsubscribe() = 0;
```

```
};
```
Pseudocódigo. fichero de cabecera de la clase abstracta DroneTelemetry.

Para implementar el módulo de telemetría, se ha creado primero una clase abstracta que servirá como base para las demás clases de telemetría. Esta clase se llama DroneTelemetry y en ella se definen dos métodos: subscribe y unsubscribe con los cuales se suscribirán y cancelarán suscripción con los canales de telemetría que les corresponda.

Además, la clase cuenta con dos miembros: uno es un objeto de tipo Telemetry proveniente de la librería mavsdk necesario para poder suscribirse a los canales de telemetría del dron, y el otro es un objeto de tipo Connection, que se utilizará principalmente para publicar los datos de telemetría que se reciban del dron.

Hasta ahora, se han creado cuatro clases que heredan de DroneTelemetry. Estas son:

- Drone Health, que permite conocer el estado del dron antes de despegar.
- Drone Land, que informa sobre el estado del dron al momento de aterrizar.
- Drone Position, que proporciona la posición actual del dron.
- Drone State, que permite saber el estado actual del dron.

Implementación de la clase DroneHealth.

La clase DroneHealth nos permite saber si lo sensores y actuadores del dron se encuentran en buen estado y listos para volar. Esto es útil antes de realizar los despegues, ya que el autopiloto del dron necesita comprobar todos los elementos que lo componen antes de empezar el vuelo.

Para obtener información sobre la capacidad de despegue del dron, se utiliza el método subscribe para recibir datos sobre su estado. Este método se suscribe a dos canales diferentes: uno que monitorea el estado general del dron a través de la función subscribe_health_all_ok, y otro que verifica si el dron está correctamente calibrado y listo para operar mediante el método subscribe_armed. Ambos métodos reciben un callback como parámetro, que se ejecuta cada vez que se reciben nuevos datos del dron. Cuando estos callbacks se activan, los valores obtenidos se guardan en variables dentro de la clase y se publican al cliente mediante el método del objeto Connection publish_message.

```
void DroneHealth::subscribe()
{
_health_handle = _telemetry.get()->subscribe_health_all_ok([this](bool health)
{
_health = health;
auto value = health ? "Ready" : "Not Ready";
_connection_manager.get()->publish_message("[Health]:", value);
});

_armed_handle = _telemetry.get()->subscribe_armed([this](bool armed)
{
_armed = armed;
```

```
auto value = armed ? "Ready" : "Not Ready";
_connection_manager.get()->publish_message("[Arm]:", value);
});
}
```

Pseudocódigo. Función subscribe () de la clase DroneHealth

Para poder consultar el valor de las variables que contienen los datos obtenidos desde el callback se han creado dos métodos. Check_health es la función que nos permite saber si el dron se encuentra es condiciones para despegar mediante los datos obtenidos desde el dron. Check_armed nos permite saber si todos los sensores y actuadores están listos para despegar.

Implementación de la clase DroneState.

La clase Drone State nos permite saber el estado actual en el cual se encuentra el dron. Esta clase es muy útil, ya que nos permite saber que se encuentra haciendo el dron en cualquier momento. Los estados que podemos obtener son:

- Ready: Este estado nos permite saber si el dron esta armado y listo para despegar.
- Takeoff: Este estado nos permite saber si el dron está despegando.
- Hold: Este estado nos permite saber si el dron está esperando.
- Mission: Este estado nos permite saber si el dron está ejecutando una misión.
- ReturnToLaunch: Este estado nos permite saber si el dron está volviendo a la estación de retorno.
- Land: Este estado nos permite saber si el dron está aterrizando.

Para obtener la información sobre los estados se usa la función de la librería mavsdk subscribe_flight_mode con la cual nos suscribimos al canal de telemetría que se encarga de enviar los datos sobre el estado del dron. Por parámetro recibe un callback, mediante el cual obtendremos los datos del estado del dron, los guardamos en variables de la clase y también los publicamos usando el método publish_message de la clase Connection.

```
void DroneState::subscribe()
{
_handle = _telemetry.get()->subscribe_flight_mode([this](mavsdk::Telemetry::FlightMode state)
{
_flight_state = state;
std::ostringstream oss;
oss << state;
_connection_manager.get()->publish_message("[State]:", oss.str());
});
}
```

Pseudocódigo. Implementación método subscribe () de la clase DroneState.

Si la aplicación necesita consultar el estado actual del dron, se han desarrollado funciones que dan acceso a dicha información.

- La función is_holding nos permite saber si el dron está en estado de espera.

- La función is_returning nos permite saber si el dron está en estado retorno a la estación de despegue.
- La función is_taking_off nos permite saber si el dron está despegando.
- La función is_executing_mission nos permite saber si el dron está ejecutando una misión.

Todas estas funciones consultan la variable _flight_state en la que se guardan los datos sobre el estado obtenidos en el callback que se le pasa al método subscribe (). Y retornan un cierto o falso en función de si el dron está o no en el estado que se esté consultado.

Implementación de la clase Drone GPS.

La clase Drone GPS se usa principalmente para saber cuál es la posición actual del dron y notificar al cliente sobre la localización de este. Los datos que recibimos y enviamos es la latitud, longitud, altura relativa al punto de despegue y la altura con respecto al nivel del mar. Para obtener dichos datos se ha implementado el siguiente método subscribe. En este obtenemos los datos de la localización y la enviamos al cliente mediante el método publish_message en el canal [Position].

```
void DronePosition::subscribe ()
{
_handle = _telemetry.get()->subscribe_position([this](mavsdk::Telemetry::Position position)
{
auto response = std::to_string(position.latitude_deg) + "-" +
std::to_string(position.longitude_deg) + "-" +
std::to_string(position.absolute_altitude_m) + "-" +
std::to_string(position.relative_altitude_m);
_connection_manager.get()->publish_message("[Position]:", response);
});
}
```

Pseudocódigo. Método subscribe de la clase Drone GPS.

Implementación de la clase Drone Land.

La clase Drone Land nos permite saber el estado en el que se encuentra el dron a la hora de hacer un aterrizaje. A diferencia de la clase Drone State, esta provee estados más concretos que resultan útiles en casos en los que el dron se encuentre aterrizando.

```
void DroneLand::subscribe()
{
_handle = _telemetry.get()->subscribe_landed_state([this](mavsdk::Telemetry::LandedState state) -
> void
{
_land_state = state;
std::ostringstream oss;
oss << state;
_connection_manager.get()->publish_message("[Land]:", oss.str());
});
```

```
}
```
Pseudocódigo. Método subscribe de la clase Drone Land.

El método subscribe de la clase DroneLand utiliza el método subscribe_landed_state de la clase Telemetry de la libreria masvsdk. Este método recibe un callback que se ejecutará cada vez que lleguen datos sobre el estado del aterrizaje del dron. Cada vez que se reciba esta información, se guardará en variables de la clase y se publicará en el canal de telemetría [LAND].

Para poder consultar en qué estado se encuentra el dron se han definido las siguientes funciones dentro de la clase Drone land:

- Is_in_ground: Esta función nos permite saber si el dron está en el suelo.
- Is_taking_off: Esta función nos permite saber si el dron está despegando.
- Is_in_air: Esta función nos permite saber si el dron está en el aire.
- Is_landing: Esta función nos permite saber si el dron está aterrizando.

Implementación de la clase Telemetry Manager.

La clase Telemetry Manager tiene la función principal de crear las distintas instancias de los objetos que reciben telemetría, almacenándolas en un hash_map. Esto permite que las instancias permanezcan en memoria de forma continua, evitando que se cancelen las funciones de suscripción, ya que la clase se mantiene en la estructura de datos dentro de TelemetryManager.

```
TelemetryManager::TelemetryManager(const std::shared_ptr<mavsdk::System> system,
std::shared_ptr<Connection> _connection_manager)
{
auto telemetry = std::make_shared<mavsdk::Telemetry>(system);
auto drone_health = std::make_shared<DroneHealth>(telemetry, _connection_manager);
_telemetry_map.insert({TelemetryType::HEALTH, drone_health});

auto drone_Land = std::make_shared<DroneLand>(telemetry, _connection_manager);
_telemetry_map.insert({TelemetryType::LAND, drone_Land});

auto drone_state = std::make_shared<DroneState>(telemetry, _connection_manager);
_telemetry_map.insert({TelemetryType::STATE, drone_state});

auto drone_position = std::make_shared<DronePosition>(telemetry, _connection_manager);
_telemetry_map.insert({TelemetryType::POSITION, drone_position});
}
```
Pseudocódigo. Constructor de la clase TelemetryManager.

En caso de que se quiera acceder a una clase de telemetría concreta se ha creado el método get_telemetry, el cual recibe por parámetro el tipo de telemetría que se desee.

```
std::shared_ptr<DroneTelemetry> TelemetryManager::get_telemetry(TelemetryType telemetry)
{
```

```
return _telemetry_map[telemetry];
}
```
Pseudocódigo. Función get_telemetry de la clase TelemetryManager.

Creación del módulo de behavior.

El módulo de behavior se encarga de manejar el comportamiento del dron mediante la librería BehaviorTree.cpp. En el podemos ver los diferentes nodos que hemos creado para el manejo de los estados del dron, y las diferentes funciones que se encargan de ejecutar el árbol de comportamientos que usaremos para interactuar con este. Para comenzar el desarrollo de este módulo se comenzó por definir su CMakeLists.txt.

```
set(SOURCES_BT
${CMAKE_CURRENT_SOURCE_DIR}/DroneBehavior.cpp
${CMAKE_CURRENT_SOURCE_DIR}/DroneBehavior.h
${CMAKE_CURRENT_SOURCE_DIR}/PrepareToFlight.h
${CMAKE_CURRENT_SOURCE_DIR}/ReturnToLaunch.h
${CMAKE_CURRENT_SOURCE_DIR}/StartMission.h
${CMAKE_CURRENT_SOURCE_DIR}/CheckTask.h
${CMAKE_CURRENT_SOURCE_DIR}/TakeOff.h
${CMAKE_CURRENT_SOURCE_DIR}/Land.h
${CMAKE_CURRENT_SOURCE_DIR}/Hold.h
)
set(behaviortree_cpp_DIR
/home/nico/Public/BehaviorTree.CPP/build/
)
add_library(behavior ${SOURCES_BT})
find_library(CMAKE_LIBRARY_BH
NAMES behaviortree_cpp
HINTS /home/nico/Public/BehaviorTree.CPP/build
NO_DEFAULT_PATH
)
target_include_directories(behavior
PUBLIC /home/nico/Public/BehaviorTree.CPP/
)
target_link_libraries(behavior
behaviortree_cpp
controller
telemetry)
configure_file(./bh.xml ${PROJECT_BINARY_DIR}/src/bh.xml)
```
Pseudocódigo. Fichero CMakeLists.txt

Con este fichero especificamos cuales son los ficheros fuente del módulo. Se agrega como librería dentro del proyecto y se especifican cuáles son sus ficheros fuentes. También se vinculan los módulos de behabiortree_cpp, controller y telemetry.

Diseño del árbol de comportamientos.

En este apartado nos centraremos en explicar cuál es el árbol de comportamiento de la aplicación.

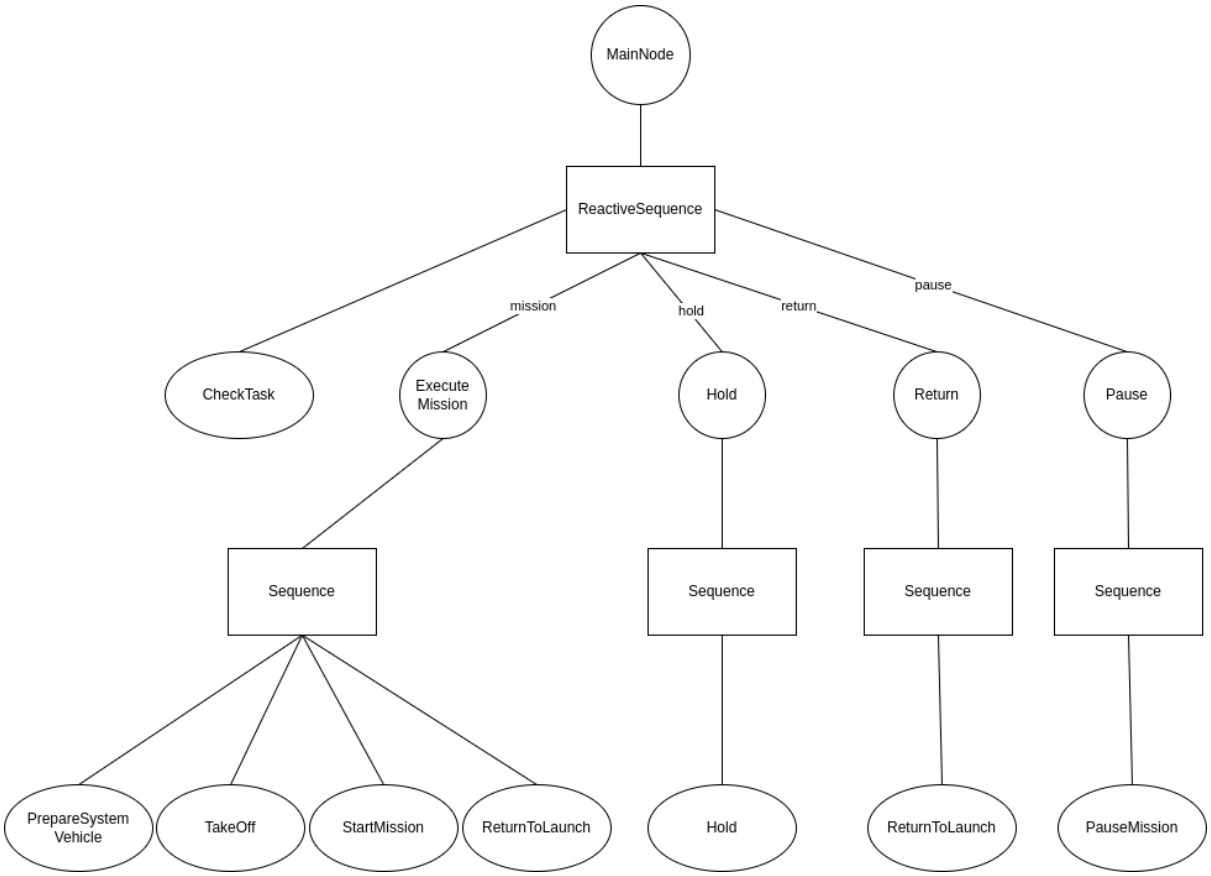

Ilustración 26. Diagrama árbol de comportamientos.

Para el control de los estados del dron se han definido diferentes subárboles que se ejecutan dentro del árbol principal. Para que se ejecute un árbol u otro se tendrá que cumplir las condiciones que se establezcan en su desencadenante.

Árbol	Desencadenante	Descripción
ExecuteMission.	Tarea: mission Parámetros: ítems []	Este árbol se encarga de hacer que el dron se prepare para despegar y cargue la misión al dron. Una vez listo despega y cuando se alcance una cierta altura comenzará a ejecutar la misión que se haya subido. En cuanto acaba la misión vuelve a la estación de despegue.
Hold	Tarea: hold	Este árbol se encarga de hacer que el dron se quede esperando a una nueva instrucción en el punto en el que la haya recibido.

Return	Tarea: return	Este árbol se encarga de hacer que el dron vuelva inmediatamente a la estación de despegue.
Pause	Tarea: pause	Este árbol se encarga de hacer que el dron pause la misión que este ejecutando y espere por una nueva.

Tabla 11. Árboles que controlan los estados del dron.

Como se explicará con mayor detalle más adelante, se creará un nodo específico para facilitar la transición entre un árbol y otro. Este nodo recibirá los desencadenantes, que son enviados como mensajes por el cliente. Cada desencadenante cuenta de dos cosas. Tarea es un string que indica cual es el estado al que el usuario quiere hacer que el dron cambie. Parámetros son los parámetros necesarios para poder ejecutar el árbol.

Antes de iniciar la implementación se comentará brevemente cuáles son los nodos creados para interactuar con el dron y cuál es su función.

Nodo	Descripción
CheckTask	Este nodo se encarga de recibir las notificaciones de las peticiones entrantes desde el thread de recepción de mensajes y actualiza el árbol que se está ejecutando actualmente.
PrepareSystem Vehicle	Este nodo se encarga de hacer que el dron se prepare para volar.
TakeOff	Este nodo se encarga de hacer que el dron despegue.
StartMission	Este nodo se encarga de hacer que el dron empiece a ejecutar la misión
ReturnTo Launch	Este nodo se encarga de hacer que el dron retorne a la estación de despegue.
Hold	Este nodo se encarga de hacer que el dron se quede esperando en una misma posición.
PauseMission	Este nodo se encarga de hacer que el dron pause la misión que este ejecutando actualmente.

Tabla 12. Nodos de la aplicación.

El funcionamiento de los estados de la aplicación es aparentemente sencillo. El nodo principal estará constantemente ejecutando el nodo CheckTask y algún árbol. El árbol que se ejecuta al iniciar la aplicación es el del árbol Hold. Gracias al nodo CheckTask podemos ver si hay alguna petición nueva disponible. En el caso de que haya alguna, el nodo CheckTask actualizara el estado según el mensaje que se reciba desde el thread de mensajes y se ejecutara uno u otro árbol según la Tarea que reciba conforme con lo definido en la tabla 11.

Implementación de la clase Drone Behavior.

La clase Drone Behavior se encarga de construir el árbol de comportamientos de la aplicación y también de enviar las señales tick de las cuales se ha hablado en el 5.3 en donde se habló de la librería BehaviorTreeCpp.

```
DroneBehavior::DroneBehavior(std::string endpoint,
```

```
std::shared_ptr<DroneMessageType> message_drone_obj,
std::shared_ptr<Connection> connection_manager)
{
BehaviorTreeFactory factory;
_drone_controller = std::make_shared<DroneController>(endpoint, connection_manager);
_message_drone_obj = message_drone_obj;

factory.registerNodeType<PrepareToFlight>("PrepareSystemVehicle", _drone_controller);
factory.registerNodeType<StartMission>("StartMission", _drone_controller);
factory.registerNodeType<TakeOff>("TakeOff", _drone_controller);
factory.registerNodeType<ReturnToLaunch>("ReturnToLaunch", _drone_controller);
factory.registerNodeType<Hold>("Hold", _drone_controller);
factory.registerNodeType<PauseMission>("PauseMission", _drone_controller);
factory.registerNodeType<CheckTask>("CheckTask", _drone_controller, message_drone_obj);

factory.registerBehaviorTreeFromFile("./bh.xml");
_tree = std::make_shared<Tree>(factory.createTree("MainTree"));

}
```

Pseudocódigo. Constructor de la clase DroneBehavior.

En el constructor de la clase DroneBehavior se crea un objeto BehaviorTreeFactory en el cual se tienen que registrar todos los nodos del sistema. Mediante el método registerNodeType podemos registrar los nodos que hemos creado para la aplicación. También cargamos el árbol de comportamiento definido en forma XML y finalmente creamos un objeto de tipo Tree el cual representa el árbol de comportamientos que acabamos de crear.

```
int DroneBehavior::start_tree()
{
auto status = _tree.get()->tickOnce();
while (status == NodeStatus::RUNNING || status == NodeStatus::FAILURE)
{
auto status = _tree.get()->tickOnce();
}
return 0;
}
```

Pseudocódigo. método start_tree de la clase DroneBehavior.

Dentro de la clase DroneBehavior se ha definido la función start_tree. Este metodo se encarga de ejecutar la función tickOnce del objeto Tree con la cual podemos mandar ticks al árbol de comportamientos. Este bucle se ejecutará prácticamente durante todo el ciclo de vida de la aplicación

Diseño del árbol de comportamiento en formato XML.

La estructura xml que se ha diseñado a partir del esquema en forma de árbol que se muestra en la ilustración 26 la siguiente:

Árbol principal.

```
<BehaviorTree ID="MainTree">
<ReactiveSequence>
<CheckTask current_task="{task}" current_mission="{mission}" />
<SubTree ID="ExecuteMission" _skipIf="task != 'mission'" exec_task="{task}"
exec_mission="{mission}" />
<SubTree ID="ReturnLaunch" _skipIf="task != 'return'" rtl_task="{task}" />
<SubTree ID="Loiter" _skipIf="task != 'hold'" loiter_task="{task}" />
<SubTree ID="Pause" _skipIf="task != 'pause'" pause_task="{task}" />
</ReactiveSequence>
</BehaviorTree>
```

Pseudocódigo. Diagrama XML del árbol principal de ejecución.

Árbol de ExecuteMission.

```
<BehaviorTree ID="ExecuteMission">
<Sequence>
<PrepareSystemVehicle mission="{exec_mission}" />
<TakeOff />
<StartMission />
<ReturnToLaunch />
<Script code=" exec_task:='' " />
</Sequence>
</BehaviorTree>
```

Pseudocódigo. Diagrama XML del árbol ExecuteMision.

Árbol de Hold.

```
<BehaviorTree ID="Loiter">
<Sequence>
<Hold />
<Script code=" loiter_task:='' " />
</Sequence>
</BehaviorTree>
```

Pseudocódigo. Diagrama XML del árbol Hold.

Árbol de Return.

```
<BehaviorTree ID="ReturnLaunch">
<Sequence>
<ReturnToLaunch />
<Script code=" rtl_task:='' " />
</Sequence>
</BehaviorTree>
```

Pseudocódigo. Diagrama XML del árbol Return.

Árbol de PauseMission.

```
<BehaviorTree ID="Pause">
<Sequence>
```

```
<PauseMission />
<Script code=" pause_task:='' " />
</Sequence>
</BehaviorTree>
```
Pseudocódigo. Diagrama XML del árbol PauseMission.

Todo el XML anterior forma parte del fichero que se vio previamente en el constructor de la clase DroneBehavior. A partir de este se crea todo el árbol de comportamiento de la aplicación.

Implementación del nodo CheckTask.

El nodo CheckTask es un nodo síncrono el cual se encarga principalmente de revisar si hay nuevas peticiones del cliente y en base al mensaje obtenido hacer un cambio de árbol de ejecución teniendo en cuenta la Tarea recibida.

Para hacer la transición dentro del árbol de comportamiento se usa el atributo xml definido por la librería BehaviorTree.CPP "skipIf="task != 'hold'". Donde task es un atributo del blackboard(Se habla de el en el capítulo 5.3 en donde se explica la librería BehaviorTree.CPP) en el cual ponemos la Tarea que haya enviado el cliente. La etiqueta _skipIf lo que hace es evitar la ejecución de un nodo o subárbol de ejecución en caso de que una condición se cumpla. De este modo solo se ejecutará el árbol cuyo task sea igual al que tiene asignado, es decir si el valor de "task = hold" se ejecutara el árbol que sea "skipIf="task != 'hold'".

```
NodeStatus tick() override
{
if (_task.get()->task != "")
{
setOutput("current_task", _task.get()->task);
setOutput("current_mission", _task.get()->items);
_task.get()->task = "";
_task.get()->items = "";
std::cout << "Updating current task" << "\n";
_drone_controller.get()->hold();
return BT::NodeStatus::FAILURE;
}
return BT::NodeStatus::SUCCESS;
}
```
Pseudocódigo. Método X del nodo CheckTask.

En esta función tick() lo que hacemos comprobar si hay una nueva petición y en caso de que sea así, se actualiza el valor de la task.

Implementación del nodo PrepareToFlight.

El nodo PrepareToFlight se encarga de preparar el dron para que pueda empezar a volar.

```
BT::NodeStatus tick() override
{
std::cout << "Executing Prepare to Flight..." << "\n";
auto mission_items = getInput<std::string>("mission");
```

```
if (!mission_items)
{
throw RuntimeError("error reading port [mission]:", mission_items.error());
}

std::cout << "Uploading mission..." << "\n";
auto mission_manager = _drone_controller.get()->get_mission_manager();
mission_manager.get()->upload_mission(mission_items.value());

// Subscribing to health channel
auto drone_telemetry = _drone_controller.get()->get_telemetry_object(TelemetryType::HEALTH);
auto drone_health = static_cast<DroneHealth *>(drone_telemetry.get());
drone_health->subscribe();

while (!drone_health->check_health())
{
std::this_thread::sleep_for(std::chrono::milliseconds(5));
}

_drone_controller.get()->arm();
std::this_thread::sleep_for(std::chrono::seconds(1));

while (!drone_health->check_armed())
{
std::this_thread::sleep_for(std::chrono::milliseconds(5));
}
drone_health->unsubscribe();
return BT::NodeStatus::SUCCESS;
}
```

Pseudocódigo. Método tick() del nodo PrepareToFlight.

Con este nodo lo que se hace es cargar los datos de misión del dron haciendo uso de la clase DroneMission y luego enviar un comando arm mediante la clase DroneController. Antes no se había mencionado pero la clase DroneController crea y guarda una instancia de DroneMission y TelemetryManager. De este modo los nodos del árbol solo necesitan un objeto externo a la ejecución del árbol y tampoco se tiene que estar instanciando objetos dentro del mismo.

Después de enviar el comando arm, se crea una suscripción al canal de health mediante la clase DroneHealth y se queda esperando hasta que el dron esté listo para despegar.

Implementación del nodo TakeOff.

El nodo TakeOff es un nodo asíncrono que sirve para hacer que el dron despegue. En este nodo hicimos la implementación de dos métodos:

```
BT::NodeStatus onStart()
{
std::cout << "Executing take off" << "\n";
```

```
_drone_telemetry = _drone_controller.get()->get_telemetry_object(TelemetryType::STATE);
_drone_telemetry->subscribe();
// Send command to drone to make a take off.
auto action_state = _drone_controller.get()->take_off();

// Check if the command was successfully uploaded.
if (action_state == DroneController::ActionStatus::Busy ||
action_state == DroneController::ActionStatus::Failed)
return NodeStatus::FAILURE;

std::this_thread::sleep_for(std::chrono::seconds(1));
return NodeStatus::RUNNING;
}
```

Pseudocódigo. función onStart() del nodo TakeOff.

El método onStart de este nodo lo que hace es suscribirse mediante la clase DroneState a canal de telemetría del estado del dron y mediante la clase DroneController ejecutar la función take_off para enviar el comando de despegue al dron.

```
BT::NodeStatus onRunning()
{
auto telemetry_state = static_cast<DroneState *>(_drone_telemetry.get());

if (telemetry_state->is_taking_off() || !telemetry_state->is_holding())
{
return NodeStatus::RUNNING;
}
telemetry_state->unsubscribe();
return NodeStatus::SUCCESS;
}
```

Pseudocódigo. Función onRunning del nodo take_off().

El método onRunning de esta clase mediante el método de la clase DroneState is_taking_off comprueba si el dron está despegando. Si aún este despega, retorna un RUNNING. Si la función retorna falso, indicara que el dron ya ha terminado de despegar.

Implementación del nodo StartMission.

Este nodo es asíncrono y se encarga de empezar la ejecución de misiones. Para este nodo se han implementado dos métodos:

```
BT::NodeStatus onStart()
{
_drone_telemetry = _drone_controller.get()->get_telemetry_object(TelemetryType::STATE);
_drone_telemetry->subscribe();
auto mission_manager = _drone_controller.get()->get_mission_manager();
mission_manager.get()->subscribe();

// Send command to drone to start mission.
```

```
std::cout << "Executing Start Mission" << "\n";
auto request_status = mission_manager.get()->start_mission();
if (request_status == DroneMission::MissionState::Busy ||
request_status == DroneMission::MissionState::Denied ||
request_status == DroneMission::MissionState::Error)
return NodeStatus::FAILURE;

std::this_thread::sleep_for(std::chrono::seconds(1));
return NodeStatus::RUNNING;
}
```

Pseudocódigo. Método onStart del nodo StartMission

Esta función onStart hace la suscripción al canal de telemetría del dron que publica los datos del estado mediante la clase DroneState. Después, mediante la clase DroneMission ejecuta la función start_mission y con esta envía un comando MAVLink al dron para que empiece la misión.

```
BT::NodeStatus onRunning()
{
auto telemetry_mission = static_cast<DroneState *>(_drone_telemetry.get());
if (telemetry_mission->is_executing_mission())
{
return NodeStatus::RUNNING;
}

_drone_controller.get()->get_mission_manager().get()->unsubscribe();
telemetry_mission->unsubscribe();
return NodeStatus::SUCCESS;
}
```

Pseudocódigo. función onRunning del nodo StartMission.

Este método lo que hace es mediante la clase DroneState ver si el dron aún está ejecutando la misión mediante la función is_executing_mission. Si aún la está ejecutando quiere decir que necesita más tiempo para terminarla, por lo tanto, retornara un Running. Y si no retornara un SUCCESS.

Implementación del nodo ReturnToLaunch.

El nodo ReturnToLaunch es un nodo asíncrono el cual se encarga de hacer que el dron vuelva a la estación de despegue.

```
BT::NodeStatus onStart()
{
std::cout << "Executing return to launch" << "\n";

// Getting telemetry state object from drone controller
_drone_telemetry = _drone_controller.get()->get_telemetry_object(TelemetryType::STATE);
_drone_telemetry->subscribe();
```

```
// Sending return to launch command to drone
auto request_status = _drone_controller.get()->return_to_launch();

// Check if request failed.
if (request_status == DroneController::ActionStatus::Busy ||
request_status == DroneController::ActionStatus::Failed)
return NodeStatus::FAILURE;
std::this_thread::sleep_for(std::chrono::seconds(1));
return NodeStatus::RUNNING;
}
```

Pseudocódigo. Metodo onStart() del nodo ReturnToLaunch.

La función onStart del nodo ReturnToLaunch se suscribe al canal de telemetría que publicados datos sobre el estado del dron. También, mediante la clase DroneController haciendo uso del método return_to_launch se envía un comando al dron para que regrese a la estación de despegue.

```
BT::NodeStatus onRunning()
{
_drone_telemetry = _drone_controller.get()->get_telemetry_object(TelemetryType::STATE);
auto drone_health = static_cast<DroneState *>(_drone_telemetry.get());
if (drone_health->is_returning_to_lauch())
{
return NodeStatus::RUNNING;
}
drone_health->unsubscribe();
return NodeStatus::SUCCESS;
}
```

Pseudocódigo. Función onRunning() del nodo ReturnToLaunch.

La función onRunning de este nodo se encarga de ver si el dron ya ha regresado a la estación de despegue mediante el método is_returning_to_lauch de la clase DroneState. Si aún no ha vuelto quiere decir que necesita un poco más de tiempo para regresar por lo que se retorna un RUNNING. Y si ya ha regresado se retorna un SUCCESS.

Implementación del nodo Hold.

El nodo hold se encarga de hacer que el dron se quede esperando en la posición la que reciba la orden.

```
BT::NodeStatus onStart()
{
std::cout << "Executing Hold." << "\n";
_drone_controller.get()->hold();
// Subscribing to position channel
auto drone_telemetry = _drone_controller.get()->get_telemetry_object(TelemetryType::POSITION);
auto drone_position = static_cast<DronePosition *>(drone_telemetry.get());
drone_position->subscribe();
return NodeStatus::RUNNING;
```

```
}
```
Pseudocódigo. Función onStart del nodo Hold.

Esta función se encarga de mantener al dron en una posición fija utilizando la función hold de la clase DroneController. Además, dado que este nodo se ejecuta al inicio de la aplicación, emplea la clase DronePosition para suscribirse al canal de telemetría del dron, a través del cual se transmite la información de su ubicación. Así, desde el comienzo recibiremos datos sobre la posición del dron.

```
BT::NodeStatus onRunning()
{
return NodeStatus::RUNNING;
}
```
Pseudocódigo. Función onRunning del nodo Hold.

La función onRunning del nodo Hold siempre retorna RUNNING. Esto se debe a que se considera un nodo que nunca acabará su ejecución hasta que no se ejecute otro nodo u otro árbol.

Implementación del nodo PauseMission.

El nodo PauseMission se encarga de hacer que se pause la ejecución de la misión que este ejecutando el dron. Este nodo implementa dos métodos.

```
BT::NodeStatus onStart()
{
_drone_telemetry = _drone_controller.get()->get_telemetry_object(TelemetryType::STATE);
_drone_telemetry->subscribe();
auto mission_manager = _drone_controller.get()->get_mission_manager();

// Send command to drone to start mission.
std::cout << "Executing Start Mission" << "\n";
auto request_status = mission_manager.get()->pause_mission();
if (request_status == DroneMission::MissionState::Busy ||
request_status == DroneMission::MissionState::Denied ||
request_status == DroneMission::MissionState::Error)
return NodeStatus::FAILURE;

std::this_thread::sleep_for(std::chrono::seconds(1));
return NodeStatus::RUNNING;
}
```
Pseudocódigo. función onStart del nodo PauseMission.

La función onStart de este nodo se encarga de suscribirse al canal de telemetría por el que recibimos los datos del estado del dron mediante la clase DroneState. También, mediante el método pause_mission de la clase DroneMission se envía un comando al dron para que pause la ejecución de la misión que esté realizando actualmente.

```
BT::NodeStatus onRunning()
```

```
{
return NodeStatus::SUCCESS;
}
```
Pseudocódigo. función onRunning del nodo PauseMission.

El método onRunning de la clase PauseMission siempre retorna SUCCESS. Al igual que el nodo Hold, esto se debe a que se considera un nodo que nunca acabará su ejecución hasta que no se ejecute otro nodo u otro árbol.

Implementación de la función main.

La función main es el punto de partida de la aplicación, en esta se instancian los objetos más relevantes del programa y se crea el thread de recepción de mensajes.

```
int main()
{
// Create the connection class to receive and publish data.
std::string request_endpoint = "tcp://localhost:4242";
std::string publisher_endpoint = "tcp://localhost:4243";
std::shared_ptr<Connection> connection = std::make_shared<Connection>(request_endpoint,
publisher_endpoint);

// Create the pointer that gets the messages.
std::shared_ptr<DroneMessageType> drone_message = std::make_shared<DroneMessageType>();

// Create a thread that manages the reception of messages.
std::thread message_manager_thread(message_manager, drone_message, connection);

// Creates the threes behavior three.
std::string drone_endpoint = "serial:///dev/serial/by-id/usb-XXXXX:XXXX";

DroneBehavior tree = DroneBehavior(drone_endpoint, drone_message, connection);

// Start ticking the three
tree.start_tree();

message_manager_thread.join();

return 0;
}
```
Pseudocódigo. Función main de la aplicación.

La función main crea el objeto Connection y le pasa como argumentos las IP del servidor de recepción y de publicación de datos. Después crea el thread de recepción y le pasa el puntero drone_message por el cual enviará los datos de los mensajes al árbol de comportamiento. Luego se crea una instancia de la clase DroneBehavior. Dentro de DroneBehavior se crea la instancia de DroneController y esta creara la conexión con el dron que reciba por medio de la

variable drone_endpoint. También se le pasa una instancia de drone_message para que cuando se registre el nodo CheckTask se le pase dicho puntero como parametro para que pueda recibir los mensajes desde el thread de envio de datos. Por último, se usa el método start_tree de la clase DroneBehavior para que se comience a ejecutar el arbol de comportamiento.

8. Pruebas.

Para validar el programa primero se probó cada componente por separado para comprobar su correcto funcionamiento.

Validación de la compilación del proyecto.

Nombre Prueba	Estado prueba
Comprobar correcta compilación	Al ejecutar el comando cmake con los argumentos necesarios podemos ver que todos los ficheros fuente se compilan correctamente.
Comprobar correcta creación de módulos	Dentro de la carpeta de construcción del proyecto podemos ver el ejecutable de la aplicación y los diferentes binarios en sus respectivos módulos.

Tabla 13. Pruebas de compilación.

En este apartado para probar la compilación se ha usado el comando cmake. Una vez ejecutado, podemos observar que la aplicación se compila satisfactoriamente y se generan los ficheros ejecutables con los cuales poder ejecutar la aplicación.

Comprobar conexión de la aplicación con el dron.

Nombre Prueba	Estado prueba
Comprobar conexión con el dron vía MAVLink	Una vez creados los objetos de conexión de MAVSDK, obtenemos desde el autopiloto del dron que la conexión se ha establecido correctamente.

Tabla 14. Pruebas de conexión con el dron.

Para comprobar que se establece la conexión con el dron, gracias a la librería MAVSDK y al autopiloto del dron, podemos obtener información de depuración enviada por este. Este primer intercambio de mensajes nos muestra como efectivamente se ha establecido la conexión con el dron.

Comprobar funcionalidad de la API de control del dron.

Nombre Prueba	Estado prueba
Comprobar envió de comandos de despegue	El dron recibe el comando de despegue enviado desde el pc abordo, y comienza el despegue.
Comprobar envió de comandos de preparación para vuelo	El dron recibe el comando de preparación para vuelo enviado desde el pc abordo, y comienza a preparar el vuelo.
Comprobar envió de comandos de ejecución de misión	El dron recibe el comando de ejecución de misión enviado desde el pc abordo, y comienza la ejecución de misión.

Comprobar envió de comandos de transición a ala fija	El dron recibe el comando de transición a ala fija enviado desde el pc abordo, y comienza a hacer la transición.
Comprobar envió de comandos de transición a quad-rotor	El dron recibe el comando de transición a quad-rotor enviado desde el pc abordo, y comienza a hacer la transición.
Comprobar envió de comandos de espera	El dron recibe el comando de espera enviado desde el pc abordo, y se queda en vuelo dando giros en el mismo sitio hasta que recibe otro comando.
Comprobar envió de comandos de retornar	El dron recibe el comando de retornar a la estación de aterrizaje enviado desde el pc abordo, y comienza a retornar.
Comprobar envió de comandos de pausar misión	El dron recibe el comando de pausar misión enviado desde el pc abordo, y se queda esperando a que se le envíe otra tarea.
Comprobar envió de comandos de subida de misión	El dron recibe el comando de subida de misión enviado desde el pc abordo, y enviar los datos de la misión que ha de subir.
Comprobar envió de comandos de pausa de misión	El dron recibe el comando de pausa de misión enviado desde el pc abordo, y espera hasta la siguiente tarea o a que se reanude la misión.

Tabla 15. Pruebas de la API del dron.

Gracias al dron virtual y a QGroundControl podemos ver en tiempo real las acciones que realiza el dron. Usando esto podemos validar el funcionamiento de la aplicación.

Comprobar objetos receptores de telemetría.

Nombre Prueba	Estado prueba
Comprobar obtención telemetría STATE	Usando las clases de telemetría y suscribiéndonos al tópico de estado, obtenemos en tiempo real los datos del estado de vuelo del dron.
Comprobar obtención telemetría GPS	Usando las clases de telemetría y suscribiéndonos al tópico de GPS, obtenemos en tiempo real los datos del GPS del dron.
Comprobar obtención telemetría MISSION	Usando las clases de telemetría y suscribiéndonos al tópico de misión, obtenemos en tiempo real los datos del estado de la misión del dron.
Comprobar obtención telemetría HEALTH	Usando las clases de telemetría y suscribiéndonos al tópico del estado del dron, obtenemos en tiempo real los datos del estado del dron.

Tabla 16. Pruebas de la recepción de telemetría.

Usando MAVSDK para generar la subscripción a los diferentes canales de telemetría del dron, pudimos obtener los datos que nos interesaban de forma correcta.

Comprobar árboles de comportamiento del dron.

Nombre Prueba	Estado prueba
Comprobar estado de espera	Haciendo la ejecución de este árbol de comportamiento se puede observar que la secuencia de los nodos funciona correctamente. Hace que el dron permanezca a la espera en el aire o en tierra.
Comprobar estado de ejecutar misión	Haciendo la ejecución de este árbol de comportamiento se puede observar que la secuencia de los nodos funciona correctamente. Hace que el dron se prepare para ejecutar la misión, despegue, haga transición a ala fija y comience a ejecutar la misión. Una vez finalizar vuelve a la estación de despegue y desarma los motores.
Comprobar estado de preparación de vuelo	Haciendo la ejecución de este árbol de comportamiento se puede observar que la secuencia de los nodos funciona correctamente. Hace que el dron prepare todos sus componentes para poder despegar.
Comprobar estado de retorno	Haciendo la ejecución de este árbol de comportamiento se puede observar que la secuencia de los nodos funciona correctamente. Hace que el dron vuelva inmediatamente a la estación de despegue.
Comprobar transición de estados	Podemos ver que cada vez que recibe el pc abordo una petición con una tarea diferente, el dron cambia su estado según la petición que esta haya recibido inmediatamente.

Tabla 17. Prueba de los árboles de comportamientos del dron.

Para esta prueba se provo cada árbol de comportamiento creado y cada uno de sus nodos, gestionando sus estados con la telemetría obtenida por el dron. También se probó la transición entre los diferentes árboles.

Comprobar las clases de peticiones y publicación de datos.

Nombre Prueba	Estado prueba
Comprobar recepción de peticiones	Para comprobar la recepción de mensajes se usan las funciones de la librería de ZMQ y se guardan los streams de datos recibidos.
Comprobar parceo de peticiones	Para el parceo de la información obtenida de los mensajes, se pasan los datos recibidos a la función que se encarga del parseo y obtenemos que los diferentes objetos se instancian correctamente, respetando los campos definidos.
Comprobar publicación datos	Para la comprobación de la publicación de datos se ha creado un programa que

	consumiera de los diferentes tópicos definidos en este proyecto el cual se conecta al pc abordo y obtiene los datos de esta. Efectivamente se han obtenido los datos de telemetría de los diferentes canales (Health, Mission, Status, GPS) con su respectivo formato.

Tabla 18. Pruebas de la recepción y publicación de datos.

Para probar las conexiones con el pc abordo, se han creado aplicaciones de prueba que usan ZMQ y se ha establecido comunicación con el dron. Mediante esa conexión enviamos tareas para poder cambiar de un estado a otro al dron y ver si se comporta como se espera según las peticiones realizadas.

9. Conclusiones.

Conclusiones del proyecto.

Según los resultados que se han obtenido a lo largo del desarrollo de la tesis se hará un análisis de lo que se ha obtenido y se repasaran los requisitos funcionales y no funcionales para verificar que hayan sido cumplidos. Con esto podremos comprobar si el software diseñado se ajusta a los requerimientos del usuario.

Requisitos de envío de tareas.

Listaremos los comandos que el dron es capaz de recibir y ejecutar.

- Ejecutar misión: El pc abordo es capaz de recibir peticiones con la misión que el usuario quiere ejecutar, con sus respectivos parámetros de posición y velocidad y ejecutarla.
- Volver a la estación de despegue: El pc abordo recibe la petición de volver a la estación de despegue y el dron retornara.
- Pausar misión: El pc abordo es capaz recibir la petición de pausar misión y en caso de ser así se quedará esperando a la siguiente tarea.
- Esperar: El pc a bordo es capaz de recibir una petición de esperar, lo que hará que el dron permanezca en su posición actual hasta que reciba una nueva tarea.

Requisito de recepción de telemetría.

- El cliente puede recibir datos de telemetría del dron.
- El cliente es capaz de recibir los datos del canal de telemetría que le interese.

Cumplimiento de requisitos no funcionales.

Aunque no pudimos medir con precisión la velocidad de recepción y envío de datos debido a la falta de tiempo, las pruebas realizadas indicaron que ambos procesos ocurrían casi de manera instantánea. También cabe mencionar que la transición entre los diferentes estados era bastante rápida.

También podemos observar que el código es bastante modular. La implementación de nuevas funcionalidades no es complicada, ya que el diseño de la aplicación permite una gran flexibilidad para incorporar cambios.

Conclusiones personales.

Gracias a la tesis de fin de grado me he enfrentado aún reto de bastante dificultad para mí, ya no solo por la complejidad que me supuso la aplicación en sí, sino también el tiempo del que disponía para poder llevarla a cabo.

Muchas fueron las horas de depuración de errores, consultas de documentación y tecleo incesante corrigiendo errores he implementado funciones nuevas. Horas de frustración y agobio seguidas de una sensación de satisfacción y alegría cada vez que superaba un obstáculo.

Después de 4 meses de gran intensidad he de decir que a nivel personal estoy bastante orgulloso con el trabajo realizado. Desde el minuto 0 hasta el final ha sido un reto que a nivel profesional considero que me ha enriquecido bastante.

Gracias a la tesis me he dado cuenta de algo muy valioso para mí a nivel personal. Ese algo es la transición de un alumno con poca confianza en sus decisiones a alguien que entiende que, aunque aún le queda mucho por aprender, su voz y opinión también tienen poder y es relevante de una u otro manera.

Aunque la tesis fue un gran reto que asumí prácticamente solo, he de agradecer a Amelia-HUB por haberme brindado la oportunidad de trabajar con ellos. Y también he de agradecer más concretamente a mi jefe Ferran Roure por haber tenido paciencia conmigo sobre todo al principio de mi estancia en la empresa y por haberme apoyado en los momentos de incertidumbre en los que no sabía cómo abordar algún error o problema de diseño de la aplicación.

10. Referencias.

[1] McLaren, E. (s. f.). *El primer bombardeo aéreo de la historia: Austria bombardea Venecia en 1849*. https://fdra-aereo.blogspot.com/2019/06/el-primer-bombardeo-aereo-de-la.html

[2]*UNMANNED AERIAL VEHICLES*. (s. f.).

https://web.archive.org/web/20090724015052/http://www.airpower.maxwell.af.mil/airchronicles/apj/apj91/spr91/4spr91.htm

[3] Wikipedia contributors. (2024, 29 agosto). *Unmanned aerial vehicle - Wikipedia*.

https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle#/media/File:MQ-1_Predator_UAV_(cropped).jpg

[4] Classification *of the Unmanned Aerial Systems | GEOG 892: Unmanned Aerial Systems*. (s. f.). https://www.e-education.psu.edu/geog892/node/5

[5] Floreano, Dario, Science, technology and the future of small autonomous drones, 2015, vol 521, 461-462, https://infoscience.epfl.ch/entities/publication/66c076df-ae34-466d-b493-35abce66aa2e

[6] Wikipedia contributors. (2024b, agosto 29). *Unmanned aerial vehicle - Wikipedia*. https://en.wikipedia.org/wiki/Unmanned_aerial_vehicle#cite_ref-89:~:text=Flight%20stack%20overview

[7] Van Der Meer, R. (2021, 3 junio). *Enlace descendente o de usuario — SIM IoT Olivia*. SIM IoT Olivia. https://www.oliviawireless.es/glosario-de-iot/enlace-descendente-o-de-usuario

[8] *15 Usos de drones que quizás no conocías - Aviation Group*. (2024, 14 marzo). Aviation Group. https://www.aviationgroup.es/actualidad/usos-drones-no-conocias/

[9] AESA agencia estatal de seguridad aerea.

https://www.seguridadaerea.gob.es/es/ambitos/drones/operaciones-con-uas-drones/operaciones-con-uas-drones---categoria-abierta-subcategorias-a1-a2-y-a3

[10] *Introduction · MAVSDK Guide*. (s. f.). https://mavsdk.mavlink.io/main/en/

[11] Frame MAVLink https://mavlink.io/assets/packets/packet_mavlink_v2.jpg

[12] *Serialization · MAVLink Developer Guide*. (s. f.). https://mavlink.io/en/guide/serialization.html

[13] *MAVLink XML Schema · MAVLink Developer Guide*. (s. f.). https://mavlink.io/en/guide/xml_schema.html

[14] *Introduction · MAVSDK Guide*. (s. f.-b). https://mavsdk.mavlink.io/main/en/

[15] *ZeroMQ*. (s. f.). https://zeromq.org/

[16] *Introduction to BTs | BehaviorTree.CPP*. (s. f.). https://www.behaviortree.dev/docs/learn-the-basics/BT_basics

[17] *BehaviorTree.CPP*. https://www.behaviortree.dev/assets/images/intro_build_trees-d8cabed430fbf41b95a63c09f44b528a.svg

[18] *Sequences | BehaviorTree.CPP*. (s. f.). https://www.behaviortree.dev/docs/nodes-library/SequenceNode

[19] *Your first Behavior Tree | BehaviorTree.CPP*. (s. f.). https://www.behaviortree.dev/docs/tutorial-basics/tutorial_01_first_tree

[20] *Reactive and Asynchronous behaviors | BehaviorTree.CPP*. (s. f.). https://www.behaviortree.dev/docs/tutorial-basics/tutorial_04_sequence

[21] *Blackboard and ports | BehaviorTree.CPP*. (s. f.). https://www.behaviortree.dev/docs/tutorial-basics/tutorial_02_basic_ports

11. Futuras implementaciones.

El desarrollo de la aplicación de PC abordo es un proceso complejo que ha demandado una considerable cantidad de tiempo, lo que implica que todavía hay muchas áreas por mejorar y nuevas funcionalidades por agregar. Este es un proyecto en constante evolución, y se seguirán realizando actualizaciones para optimizar la aplicación.

Debido al poco tiempo disponible, se tuvo que priorizar el cumplimiento de los requisitos esenciales del usuario para el primer entregable del proyecto. Esto significó dejar de lado algunas funcionalidades que se tenían previstas. Sin embargo, esto no quiere decir que esas funciones no se implementarán en futuras versiones de la aplicación.

Actualmente, las funciones que ofrece el dron son limitadas. Aunque son bastante útiles para su manejo general, hay ciertos aspectos que podrían mejorarse. Algunos de estos aspectos incluyen:

- Envío de códigos de respuesta a las peticiones del cliente: Por ahora, el cliente sabe que las peticiones que envía al dron funcionan puesto que recibe telemetría y con esta puede ver los cambios en los estados del dron. No obstante, se considera oportuno tener un sistema de envío de respuestas al cliente confirmando que se ha recibido correctamente su solicitud. También informar sobre si esta pudo ser tratada correctamente o si presenta algún fallo la petición.
- Tener un sistema que compruebe que la aplicación funciona correctamente: En caso de que el programa llegue a fallar por algún bug inesperado en el sistema, estaría bien crear un sistema auxiliar que se encargue de saber si el programa del pc abordo está funcionando correctamente. En caso de que no lo este que tome alguna medida como forzar que el dron vuelva a la estación de despegue, o reiniciar la aplicación o la placa en la que se esté ejecutando la aplicación.
- Crear estados en el árbol de ejecución que se encarguen de manejar errores críticos dentro del dron: En momentos en los que el dron presente algún fallo crítico, tener un árbol de comportamiento adaptados al fallo en cuestión y tomar medidas tales como hacer aterrizajes de emergencia o volver inmediatamente a la estación de despegue.
- Crear un estado que permita el control del dron de forma manual: Además de simplemente enviar misiones al dron, podría ser muy útil contar con un modo en el que puedas indicar un punto específico y hacer que el dron se desplace directamente hacia ese lugar.