UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Configuración e implementación de UAV y bases de carga para levantamiento topográfico

INFORME DE ACTIVIDADES PROFESIONALES

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Rodrigo Miguel Reyes Valderrama

ASESORA DE INFORME

M.I. Tanya Itzel Arteaga Ricci

Ciudad Universitaria, Cd. Mx., 2026

PROTESTA UNIVERSITARIA DE INTEGRIDAD Y HONESTIDAD ACADÉMICA Y PROFESIONAL
(Titulación con trabajo escrito)

De conformidad con lo dispuesto en los artículos 87, fracción V, del Estatuto General, 68, primer párrafo, del Reglamento General de Estudios Universitarios y 26, fracción I, y 35 del Reglamento General de Exámenes, me comprometo en todo tiempo a honrar a la institución y a cumplir con los principios establecidos en el Código de Ética de la Universidad Nacional Autónoma de México, especialmente con los de integridad y honestidad académica.

De acuerdo con lo anterior, manifiesto que el trabajo escrito titulado CONFIGURACION E IMPLEMENTACION DE UAV Y BASES DE CARGA PARA LEVANTAMIENTO TOPOGRAFICO que presenté para obtener el titulo de INGENIERO EN COMPUTACIÓN es original, de mi autoría y lo realicé con el rigor metodológico exigido por mi Entidad Académica, citando las fuentes de ideas, textos, imágenes, gráficos u otro tipo de obras empleadas para su desarrollo.

En consecuencia, acepto que la falta de cumplimiento de las disposiciones reglamentarias y normativas de la Universidad, en particular las ya referidas en el Código de Ética, llevará a la nulidad de los actos de carácter académico administrativo del proceso de titulación.

RODRIGO MIGUEL REYES VALDERRAMA
Número de cuenta: 318086721

Dedico este trabajo a las personas que fueron parte de este logro.

A mis padres, Miguel y Gabriela, por estar siempre para mí y apoyarme en cada paso de mi vida; por mostrarme que la familia es la fuerza más grande que uno puede tener. Gracias por darme alas para volar y raíces para recordar siempre quién soy.

A mi hermano Diego, por las pequeñas lecciones de vida que solo un hermano puede dar, y por ser ese recordatorio constante de que la familia también es amistad.

A mi pareja Daniela, por su amor y comprensión infinita en los tiempos difíciles, por acompañarme celebrando las victorias y ayudándome en los tropiezos. Contigo descubrí que la vida no solo se vive, se disfruta.

Sin ustedes, este logro no tendría el mismo sentido. Estoy profundamente agradecido por todo; gracias por siempre confiar en mí.

A la Ing. Tanya Ricci y al Ing. Jesus Flores, por compartir su sabiduría con generosidad y por guiarme con paciencia y profesionalismo. Su mentoría ha sido fundamental en mi formación.

Índice general

Objetivo 4
Objetivo 4
Objetivos Específicos 4
Introducción 4

1. Marco Teórico **5**
1.1. Levantamiento topográfico 5
1.2. Microprocesador 6
1.3. Arquitecturas 6
1.3.1. Von Neumman 6
1.3.2. Harvard 6
1.3.3. RISC 6
1.3.4. ARM 7
1.4. Sistema Embebido 7
1.5. Software Libre 7
1.6. GNU/Linux y sus distribuciones 8
1.7. Lenguajes de programación 9
1.7.1. Bajo nivel y lenguajes de sistema 9
1.7.2. Alto nivel y abstracción 10
1.7.3. Ejemplos 10
1.7.3.1. C++ 10
1.7.3.2. Python 10
1.8. Modelo V 10
1.9. Git 11
1.10. ROS 2 12
1.11. Docker 13
1.12. Pruebas de software 14
1.12.1. Tipos de pruebas 14
1.12.1.1. Unitarias 14
1.12.1.2. Integración 14
1.12.1.3. Sistema 14
1.12.1.4. De rendimiento 14
1.13. MQTT 15
1.13.1. Cliente 15
1.13.2. Broker 15
1.14. AWS S3 15
1.15. UNIX Sockets 15

2. Antecedentes **16**
2.1. Sobre la empresa 16

3. Proyecto a desarrollar **18**
3.1. Configuraciones implementadas 19
3.2. Entorno de desarrollo 20
3.3. C++ 20
3.4. ROS 2 20
3.5. GNU/Linux - Ubuntu 20
3.6. Python 20
3.7. Docker 20
3.8. MQTT 21

4. Implementación y Pruebas **22**
4.1. Simulación y Ejecución del Software 22
4.2. Buenas prácticas del desarrollo de software 24
4.2.1. Mantras 24
4.2.2. Modelo V 25
4.2.3. Pruebas Unitarias 25
4.2.3.1. Gtest 26
4.2.4. Manejador de versiones 27
4.2.4.1. Git 27
4.2.4.2. Issue/Feature 28
4.2.4.3. Ramas 28
4.2.4.4. Merge Request 29
4.3. Desarrollo de *Issues/Features* 30
4.3.1. Software del UAV 30
4.3.1.1. Cargar archivos a S3 AWS 30
4.3.1.2. Promesas inválidas 31
4.3.1.3. Cancelación de vuelo 32
4.3.1.4. Forking Server 32
4.3.1.5. Control de streaming 34
4.3.2. Software del centro de carga 35
4.3.2.1. Algoritmo de manejo de motor a pasos 35

5. Resultados obtenidos y Conclusiones **38**

Glosario **40**

Objetivo

Desarrollar un software para dar autonomía a dispositivos de vuelo no tripulados (*Unmanned Aerial Vehicle* "UAV" por sus siglas en inglés).

Objetivos Específicos

- Realizar levantamientos topográficos que incluyen toma de video, fotografía aérea, mapeo y supervisión.
- Intercomunicar el UAV con su centro de carga para resguardo del exterior cuando se realizan vuelos autónomos.

Introducción

Cuando comencé la carrera no tenía claro cuál sería mi campo de profundización o a qué me iba a dedicar cuando egresara. La oportunidad de trabajar en esta empresa de desarrollo se dio cuando yo aún no terminaba la carrera, lo cual, me dio el panorama profesional de lo que yo podría hacer al termino de la carrera. Trabajar y estudiar no es sencillo, pero sí te da la posibilidad de aplicar los conocimientos teóricos en la practica de manera simultanea. Gracias a lo anterior, pude ejercer en el campo laborar desarrollando este proyecto que ha sido un pilar en la decisión de dedicarme al desarrollo de sistemas embebidos.

Este informe se estructura de la siguiente manera: inicialmente, se presenta el marco teórico con los conceptos y herramientas base. Posteriormente, se describe el entorno laboral y mi rol dentro de la empresa, dando paso a la definición del proyecto y sus configuraciones técnicas. El núcleo del trabajo detalla el desarrollo de *issues* y *features*, para concluir con una reflexión sobre los resultados obtenidos y el aprendizaje que fortaleció mi perfil profesional.

Capítulo 1

Marco Teórico

1.1. Levantamiento topográfico

La topografía es una ciencia que estudia las dimensiones de grandes zonas de tierra (González, 2006). Un levantamiento topográfico (Véase figura 1.1) es un conjunto de técnicas nacidas de la topografía con objetivo el calculo de las dimensiones de volúmenes para elaboración de planos y perfiles.

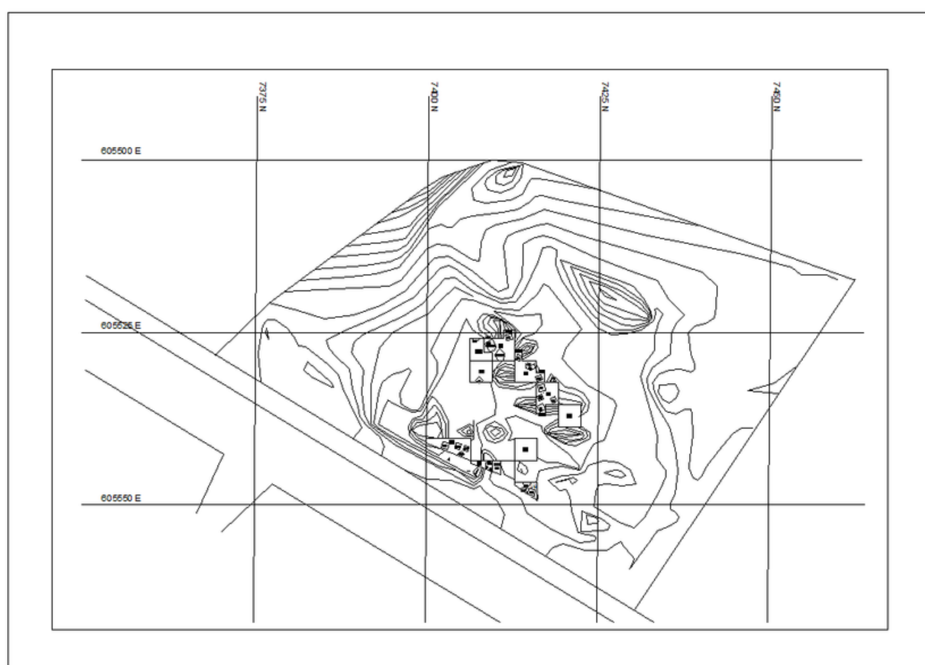

Figura 1.1: Ejemplo de un plano con levantamiento topográfico (Sandoval, 2017)

La topografía usa como base la geometría plana, trigonometría rectilínea y esférica, entre otros conocimientos matemáticos.

Existen diferentes tipos de levantamientos topográficos para diferentes tipos de necesidades, entre los que mas nos interesan está el levantamiento planialtimétrico, que con el uso de UAVs podemos conseguir varios productos como sería ortomosaico, que significa: fotografía de alta resolución de todo el terreno con escala. También

podemos conseguir una nube de puntos, cuya definición la podemos sintetizar como conjunto de miles de puntos siendo la base para modelos 3D. Dos más serían el Modelo Digital de Superficie (MDS) y Modelo Digital del Terreno (MDT), que son modelos que representan elevaciones.

1.2. Microprocesador

Los microprocesadores son predominantes en la industria para uso de procesamiento en computadoras modernas (Josh Schneider, 2024). Este tipo de procesadores combinan los componentes y la función de una unidad central de procesamiento (CPU) en un solo circuito integrado o unos pocos conectados.

1.3. Arquitecturas

En el mundo de la computación las arquitecturas son esenciales para un mejor aprovechamiento y rendimiento del hardware. La arquitectura de hardware y arquitectura de instrucciones son conceptos complementarios en el diseño de computadoras, la correcta selección de ambos es crucial para el uso que se le va a dar. Es importante mencionar los tipos de arquitectura, ya que, el proyecto que estoy presentando requiere de una arquitectura especializada, y para entender el porque explicaré algunas de estas.

1.3.1. Von Neumman

Esta arquitectura fue propuesta por John Von Neumman en 1945, esta es la base de la mayoría de las computadoras modernas, se caracteriza por tener un mismo bus tanto como una misma memoria para instrucciones y datos (Alonso, 2025). El procesador no puede leer una instrucción y un dato al mismo tiempo, ya que ambos comparten el mismo bus.

1.3.2. Harvard

La arquitectura Harvard fue propuesta en 1944 en la Universidad de Harvard, con el objetivo de mejorar el rendimiento de la arquitectura Von Neumman (Corvo, 2024). Lo que caracteriza a esta arquitectura es que separan las instrucciones y los datos, teniendo cada uno sus propios buses y memorias. En esencia esto permite que se pueda acceder a una instrucción mientras se accede a un dato.

1.3.3. RISC

La arquitectura RISC (Reduced Instruction Set Computer) se caracteriza por el uso de un repertorio de instrucciones simplificado y optimizado, lo que permite que las operaciones se ejecuten con mayor velocidad y eficiencia (Luna et al., 2022). Las instrucciones RISC son de longitud fija y generalmente se ejecutan en un ciclo de reloj, teniendo una alta eficiencia.

1.3.4. ARM

La arquitectura ARM (Advanced RISC Machine) es una arquitectura de procesador basada en el conjunto de instrucciones RISC (Hat, 2022). En la actualidad es altamente empleada en *smartphones*, sistemas embebidos y en general dispositivos de bajo consumo. Se usa en este tipo de dispositivos porque combina una eficiencia energética con un alto rendimiento y un conjunto de instrucciones reducido.

1.4. Sistema Embebido

Un sistema embebido (Véase figura 1.2) es un sistema de computación diseñado con componentes esenciales dentro de una placa base de tamaño reducido para propósitos específicos (TRBL Services, 2021). El procesamiento de este sistema es llevado a cabo generalmente por microprocesadores incluyendo también una interfaz de entrada/salida.

Figura 1.2: Sketch sistema embebido

1.5. Software Libre

Software libre es un movimiento que se centra en la libertad, sus principios filosóficos se basan en “cuatro libertades” para el usuario: la libertad de usar el software, la libertad de estudiarlo y modificarlo, la libertad de redistribuir copias y la libertad de distribuir las modificaciones.

En la historia del software, al principio, este nació siendo un complemento del hardware sin ser visto como un producto valioso, siendo así que grupos como SHARE (IBM) y DECUS incentivaban el intercambio de software con cierta facilidad (Barahona, 2021). No fue hasta en 1969 cuando IBM cambio el rumbo del software y empezó a venderlo por separado del hardware, impulsando al software propietario que tomo mucha fuerza rápidamente. El proyecto que acuñó el término de software

libre tuvo lugar en la década de 1980, cuando Richard Stallman anunció el Proyecto GNU con el objetivo de crear un sistema operativo UNIX basándose en la filosofía de software libre, la cual definió con las "cuatro libertades".

Al pasar de los años, este movimiento ha logrado importantes hitos a nivel mundial, impulsados por el poder de su gran comunidad. Su gran fuerza es debido a que su enfoque es totalmente en beneficio de los usuarios.

La comunidad es grande y activa porque cada persona que participa desarrollando lo hace para mejorar una herramienta que le sirve a ella y a miles más. Esta dinámica crea un ecosistema de apoyo que garantiza software más seguro, centrado en las personas, pero sobre todo transparente y de acceso libre.

De este movimiento de software libre nació un concepto más joven conocido como "código abierto" (*open source*), que data de finales de la década de 1990. Aunque parece ser una alternativa al término "software libre", en filosofía y práctica hoy van de la mano, siendo asociados bajo el concepto de "Software Libre y de Código Abierto" (Free and Open Source Software "FOSS" por sus siglas en inglés).

1.6. GNU/Linux y sus distribuciones

GNU fue un proyecto iniciado por Richard Stallman inspirado en crear un sistema operativo similar a UNIX pero que fuera FOSS (Stallman, 2021). La pieza clave, el núcleo del sistema, fue el kernel de Linux creado por Linus Torvalds en 1991. De esta unión, el núcleo del proyecto liderado por Linus Torvalds y Richard Stallman proporcionando las herramientas y componentes del Proyecto GNU surgió lo que hoy conocemos como GNU/Linux.

Tenemos diferentes distribuciones de GNU/Linux (Véase figura 1.3), de las cuales resaltan tres grandes familias de distribución y de ahí han surgido innumerables versiones de este sistema operativo.

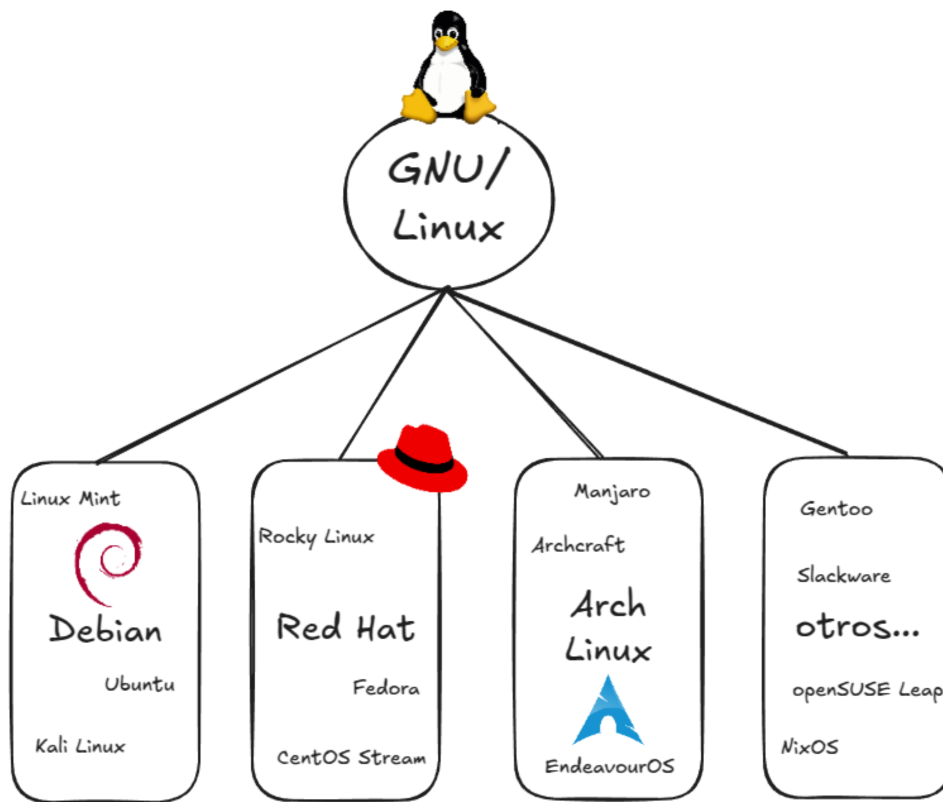

Figura 1.3: Principales distribuciones de GNU/Linux

GNU/Linux es parte fundamental en el mundo del desarrollo de software, siendo la base para la creación de grandes sistemas y con tendencia a ser de los favoritos para los desarrolladores.

1.7. Lenguajes de programación

Un lenguaje de programación es el conjunto de reglas, símbolos y palabras reservadas para plasmar instrucciones que posteriormente se traducen a un nivel que la computadora pueda interpretar. En esencia, es la manera en la que los humanos se comunican con las computadoras sin necesidad de escribir en código binario. La principal diferencia entre ellos radica en su nivel de abstracción respecto al hardware.

1.7.1. Bajo nivel y lenguajes de sistema

Los lenguajes de bajo nivel permiten al desarrollador tener un control directo sobre el hardware, lo que los hace sumamente eficientes en tiempos de ejecución; un ejemplo clásico es el lenguaje ensamblador (Aguinaga, 2020). Estos lenguajes son los más cercanos al lenguaje máquina, pero requieren una gran cantidad de código para realizar tareas sencillas y un conocimiento profundo de la arquitectura del procesador.

1.7.2. Alto nivel y abstracción

Los lenguajes de alto nivel surgen para facilitar el desarrollo de software, priorizando la legibilidad y la productividad humana. Estos lenguajes suelen abstraer detalles complejos, como la gestión de la memoria o la arquitectura del procesador, permitiendo que el mismo código sea ejecutado en diferentes plataformas con cambios mínimos.

1.7.3. Ejemplos

Mencionaré los lenguajes de programación utilizados en este proyecto:

1.7.3.1. C++

Desarrollado por Bjarne Stroustrup en 1979 (van den Beuken, 2023), C++ se caracteriza por ser un lenguaje híbrido que integra capacidades de distintos niveles de abstracción. Por un lado, mantiene el control de bajo nivel heredado de C, permitiendo la gestión manual de memoria, el uso de apuntadores y la representación directa de estructuras de datos. Por otro lado, ofrece potentes herramientas de alto nivel que facilitan la abstracción y la escalabilidad, tales como la programación orientada a objetos y una robusta biblioteca estándar. Esta naturaleza dual lo hace ideal para sistemas donde el rendimiento es crítico sin sacrificar la organización de sistemas complejos.

1.7.3.2. Python

Desarrollado por Guido van Rossum en 1989 (Carvalho, 2023), Python se sitúa en un nivel de abstracción muy alto. Es un lenguaje multiparadigma, lo que significa que soporta diversos enfoques de programación (orientada a objetos, imperativa y funcional) adaptándose a diferentes necesidades. Su principal fortaleza es la claridad sintáctica y su vasto ecosistema de bibliotecas, lo que ha impulsado su dominio en áreas como la ciencia de datos, la inteligencia artificial y el desarrollo web, permitiendo al desarrollador enfocarse en la lógica del problema más que en los detalles técnicos del hardware.

1.8. Modelo V

El modelo V (Véase figura 1.4) es un modelo de proceso en el desarrollo de software para validación y verificación, es una de las metodologías de desarrollo de software más populares (Sami, 2018).

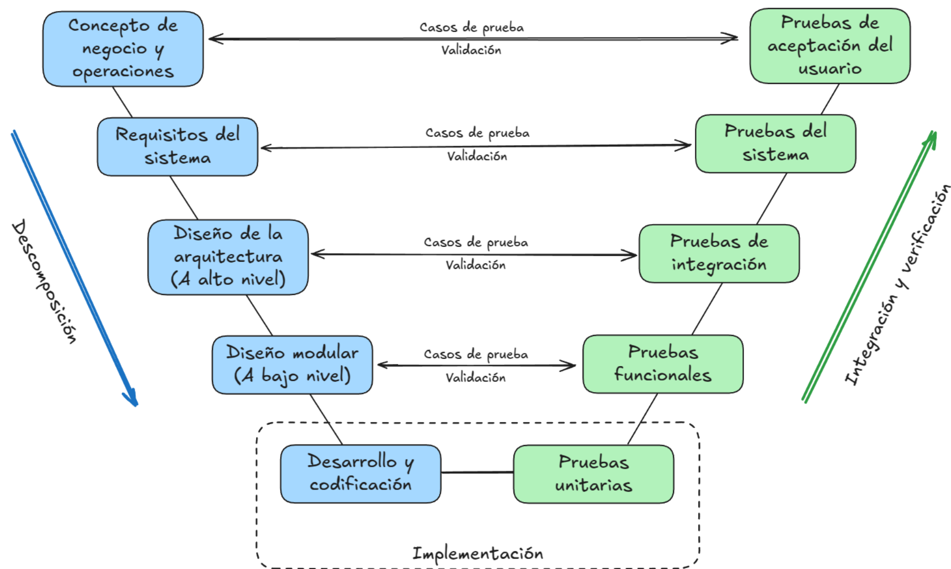

Figura 1.4: Modelo V Sami, 2018

Este modelo de proceso de desarrollo de software se basa en la asociación de una fase de pruebas para cada etapa de desarrollo. Del lado izquierdo de la V (Véase figura 1.4) se encuentra la fase de definición de requisitos y del diseño del sistema, del lado derecho, se encuentra la fase de pruebas correspondientes. De este modo se garantiza que los requisitos se verifiquen y validen a lo largo del desarrollo. Llevar los procesos en orden y no empezar una siguiente etapa sin terminar la anterior permite la detección temprana de errores, asegurando la fiabilidad y reduciendo la cantidad de trabajo.

1.9. Git

El control de versiones es una herramienta fundamental en el desarrollo de software moderno. Antes de 2002, el kernel de Linux utilizaba un sistema de control de versiones centralizado. Posteriormente, se adoptó BitKeeper, el cual introdujo un modelo de colaboración distribuido que cambió significativamente la dinámica de trabajo del proyecto. Sin embargo, en 2005, debido a restricciones en el uso de la licencia gratuita de BitKeeper, Linus Torvalds se vio impulsado a desarrollar un nuevo sistema de control de versiones distribuido denominado Git (Leninmhs, 2023).

A diferencia de los sistemas centralizados, un sistema de control de versiones distribuidos como Git permite que cada desarrollador posea una copia local completa del historial del proyecto. El manejo de los proyectos en Git se basa en el uso de ramas (Véase figura 1.5), las cuales representan líneas de desarrollo independientes donde se modifica el código sin afectar el trabajo principal. Aunque el modelo de Git es altamente flexible y no impone una estructura rígida, es una práctica común mantener una rama principal que sirva como la versión estable del proyecto, donde se fusionan los cambios que han superado las pruebas de calidad.

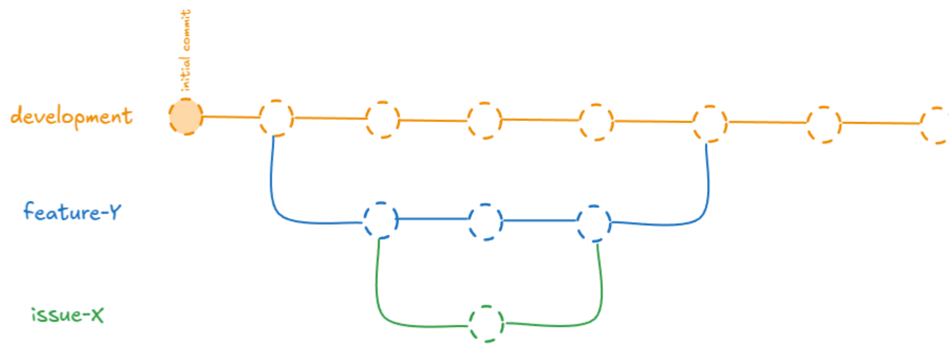

Figura 1.5: Gráfico de ramas de Git

Existen diferentes plataformas en línea para la gestión de repositorios remotos que utilizan Git, entre las cuales destacan por su amplio uso en la comunidad:

- GitLab
- GitHub

1.10. ROS 2

El Sistema Operativo de Robots (ROS Robot Operating System), es el conjunto de bibliotecas y herramientas disponibles en diferentes lenguajes de programación para el desarrollo de software para robots (ROS 2 Developers, s.f.). ROS es libre y de código abierto, desde su lanzamiento en 2007 ha cambiado la comunidad robótica, actualmente se tiene ROS2 y su última versión Jazzy Jalisco.

La principal idea con la que nos encontramos en este tipo de programación con ROS2 son los nodos, la estructura de todo el software se separa en nodos y cada nodo se enfoca en una tarea específica. Si bien cada uno tiene su propia tarea específica, se tienen que comunicar para cumplir una tarea general. Ahí es donde tenemos diferentes técnicas para comunicar los nodos (Véase figura 1.6) como los son servicios y tópicos.

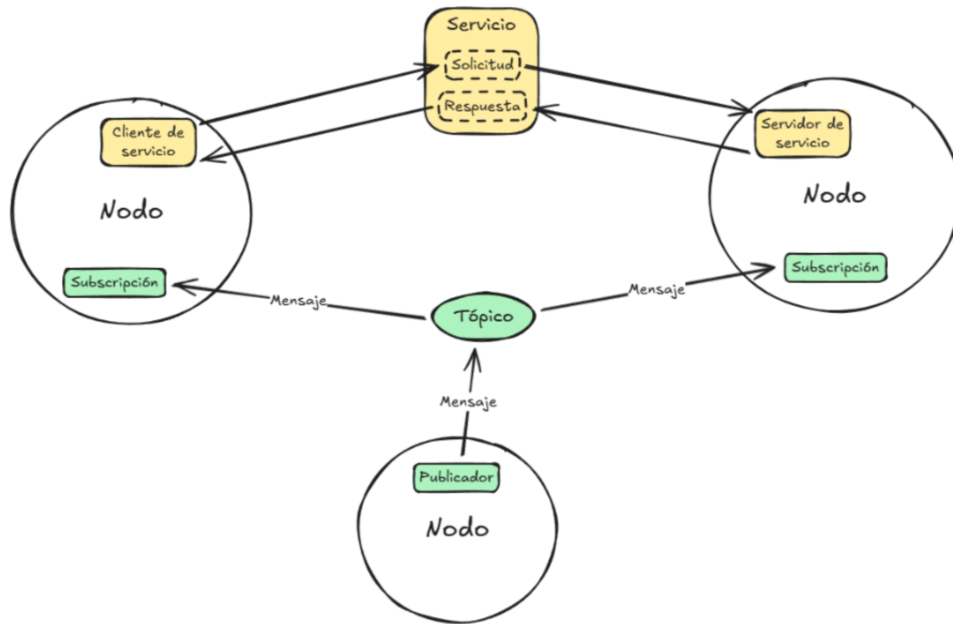

Figura 1.6: Comunicación entre nodos de ROS2

1.11. Docker

Docker llegó a revolucionar el desarrollo de software, en 2013 Solomon Hykes presentó por primera vez Docker en el evento PyCon (Johnston, 2024). Docker es una plataforma de software desarrollada con base al kernel de GNU/Linux junto con semántica de Git, dando resultados el poder crear, ejecutar y compartir aplicaciones en contenedores de forma más eficiente. La diferencia con contenedores como lo son LXC (Véase figura 1.7), es que este último se enfoca más en virtualizar un sistema operativo completo y Docker se centra en solo abstraer la aplicación junto con sus dependencias, haciendo más livianos los contenedores de Docker y resolviendo problemas como la portabilidad para ejecutarse en diferentes máquinas de forma más sencilla.

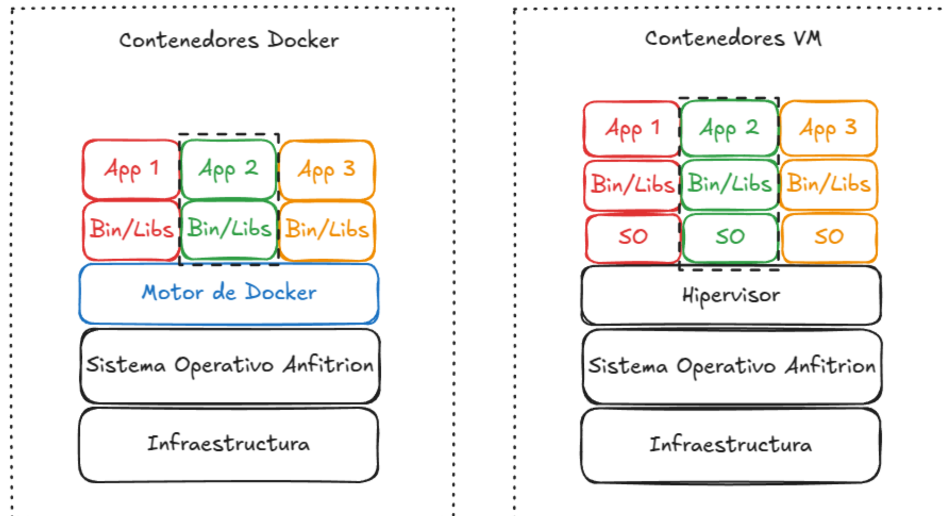

Figura 1.7: Comparativa de contenedores Docker y LXC

1.12. Pruebas de software

La pruebas en el desarrollo de software son esenciales para verificar y validar que un software funcione correctamente, cumpla con los requisitos y no tenga errores. Es un proceso que busca asegurar la calidad del software.

1.12.1. Tipos de pruebas

1.12.1.1. Unitarias

El enfoque de las pruebas unitarias es probar fragmentos pequeños de código, para asegurar que cada unidad funcione correctamente de forma aislada. Así se puede detectar errores y comprobar la funcionalidad de cada componente antes de integrarlo al sistema completo.

1.12.1.2. Integración

Las pruebas de integración se centran en verificar el correcto funcionamiento de diferentes módulos, revisando una correcta interacción entre ellos para la correcta unión de las nuevas funcionalidades.

1.12.1.3. Sistema

Estas pruebas se realizan con el sistema completo, se prueba que todas las funciones integradas trabajen juntas correctamente y se mantenga el flujo esperado del software.

1.12.1.4. De rendimiento

En este tipo de pruebas se evalúa la estabilidad, velocidad y capacidad del software bajo diferentes condiciones de carga. Se hacen pruebas de carga, midiendo el

rendimiento o también se hacen pruebas de estrés llevando el software a su límite.

1.13. MQTT

MQTT es un protocolo de mensajería ligero basado en estándares, diseñado para el intercambio de mensajes entre dispositivos («¿Qué es MQTT?», 2024). Es ampliamente utilizado en proyectos de Internet de las Cosas (IoT) para establecer comunicación a través de la red, ya que está optimizado para funcionar en entornos con bajo ancho de banda y dispositivos con recursos limitados. Estas características lo convierten en una buena opción para este tipo de proyectos.

Tiene diferentes agentes como lo son:

1.13.1. Cliente

El cliente es cualquier dispositivo ejecutando la biblioteca MQTT, con la capacidad de emplear el protocolo MQTT, se puede suscribir a diferentes tópicos para recibir mensajes de ellos o también enviar mensajes en otro tópicos.

1.13.2. Broker

El broker es un software de *back-end* que su trabajo principal es la administración de la comunicación entre dispositivos. Su rol es recibir y filtrar mensajes, identificar hacia donde van los mensajes, a que tópicos están suscritos los clientes, puede autentificar los clientes, entre otras actividades.

1.14. AWS S3

Amazon Simple Storage Service (Amazon S3) es un servicio de almacenamiento de objetos que ofrece escalabilidad, disponibilidad de datos, seguridad y rendimiento («Amazon S3», 2025). Cuenta con sus propios SDK (*Software Development Kit*) para poder implementar mediante código los servicios de S3 a los diferentes proyectos donde lo quieras usar.

1.15. UNIX Sockets

Los *sockets* del dominio de UNIX permiten una comunicación eficiente entre procesos que se ejecutan dentro de un mismo sistema, tienen diferentes configuraciones, se pueden usar con protocolos UDP o TCP («Sockets de dominio UNIX», 2024). Una característica, a diferencia de otros *sockets* que se enlazan mediante direcciones IP y puertos, es que estos *UNIX sockets* se vinculan a una ruta de archivo.

Capítulo 2

Antecedentes

Durante tres años me he desempeñado como Ingeniero de Software para sistemas embebidos, me encargo principalmente en la evolución y mejora continua del código del proyecto. Dentro de mis responsabilidades está el desarrollo de nuevas funcionalidades (*features*) y la resolución de incidencias (*issues*), como lo sería la optimización de código, refactorización para aplicar patrones de diseño más eficientes y mitigar deudas técnicas para asegurar la escalabilidad y funcionalidad del software.

Mi rol se centra en el desarrollo de software embebido para la completa autonomía del dispositivo no tripulado, así como el de su centro de carga, que apoya al dispositivo para su resguardo, carga, actualización de configuraciones y protocolos.

Mis conocimientos adquiridos en la Facultad de Ingeniería me han ayudado mucho, el pensamiento analítico que se me inculcó en la Facultad es la herramienta más importante como ingeniero al resolver problemas, he tenido que enfrentar problemas en un proyecto real y aunque he aprendido cosas que no sabía, pude comprobar que tengo bases muy fuertes. Los conocimientos cobran más sentido al momento de aplicarlos, por ejemplo, toda la teoría de programación; pude entender la importancia de la programación orientada a objetos, crear código limpio, documentación, entre otros, aun cuando somos estudiantes no le damos la importancia en los proyectos de la Facultad, encajan al incorporarte en un proyecto de carácter profesional y con *stakeholders* de alto valor y sobre todo al utilizar y aprender código heredado (*legacy code*).

Aparte del pensamiento analítico, conocimientos de programación o ciclo de vida, como programador, tienes que ir más allá de solo la generación de código, ya que, tienes que entender lo que engloba el proyecto final y que generalmente aplica a las circunstancias, el entorno laboral o inclusive los requerimientos del cliente lo que será finalmente la programación en sí, ahí también me ayudó mucho los conocimientos de la Facultad, he tenido que enfrentar problemas en donde están involucrados: motores a pasos, sistemas embebidos, circuitos básicos, entre otros.

2.1. Sobre la empresa

Durante los últimos 7 años, han sido la empresa líder en tecnología, desarrollo y operación de vehículos aéreos no tripulados (UAV) para una amplia variedad de aplicaciones.

Desde video y fotografía aérea hasta mapeo, modelos 3D y supervisión, brindan a los clientes las soluciones y servicios que demandan. Este proyecto puede realizar servicios de topografía aérea, agricultura de precisión y mucho más, utilizando UAVs de alta calidad.

Algunos años después, cambiaron su modelo de negocio y se dieron cuenta que los vuelos tienen que ser completamente autónomos para alcanzar el verdadero potencial de esta tecnología, al volar constantemente podemos recopilar muchos más datos y brindar información de alta calidad.

El proyecto está conformado por diferentes áreas (Véase figura 2.1), al frente está el dueño de la empresa. El área de la que nos interesa hablar es TI, la cual está conformada por 3 personas. Contamos con un *Maintainer* que se encarga de tomar las decisiones de alto impacto del proyecto, tanto en la página web como en los embebidos. El equipo se divide en 2 áreas principales: el área de embebidos, donde me desempeño y asumo mis funciones principales, y el área de desarrollo web. Mi incorporación a la empresa se dio por medio de un becariado; tras postularme y ser aceptado en el área de embebidos, al pasar un periodo de seis meses como becario tomaron la decisión de contratarme formalmente.

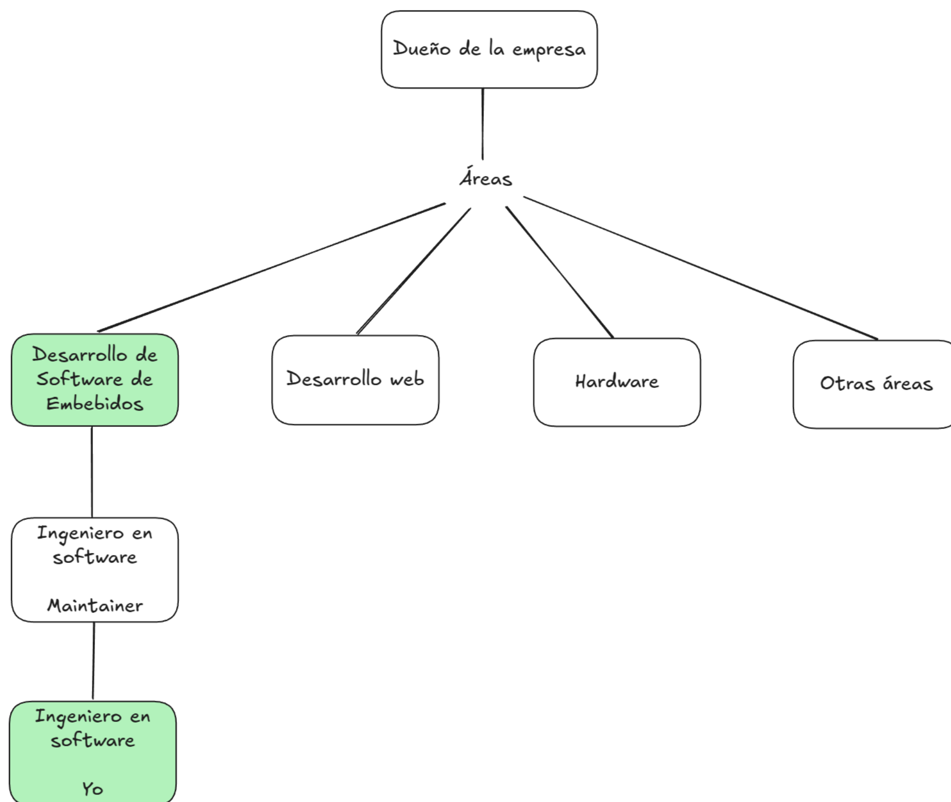

Figura 2.1: Organigrama de la empresa

Capítulo 3

Proyecto a desarrollar

Dentro de la empresa algunas de mis funciones más significativas fue el desarrollo de software para la autonomía de UAVs (Véase figura 3.1) y su centro de carga (Véase figura 3.2). La autonomía principalmente enfocada al levantamiento topográfico teniendo la toma de video, fotografía aérea, mapeo y supervisión así como otras funcionalidades. Para este trabajo me voy a enfocar en el desarrollo de software para la manipulación del UAV y su centro de carga.

Figura 3.1: Sketch de un UAV

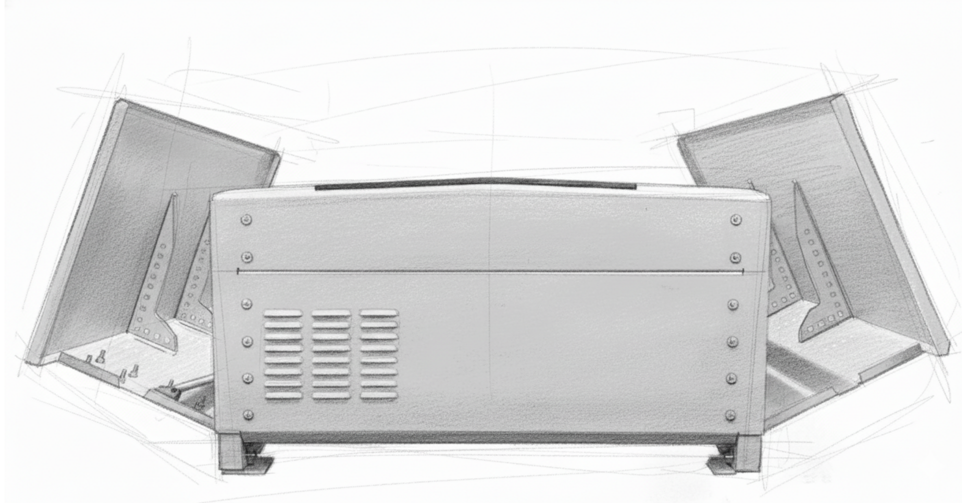

Figura 3.2: Sketch de un centro de carga para UAV

Se abarcaran *issues* para ambos sistemas embebidos, como lo es carga automática de archivos a servicios en la nube, desarrollo de un *forking server*, uso de patrones de diseño y algoritmos para manejo de motores a pasos. La autonomía del UAV conlleva la realización de diferentes tipos de misiones aéreas teniendo el manejo de fotografía, *streaming*, comunicación remota y realización de diferentes tareas para llevar a cabo una experiencia con la menor interacción humana posible.

Para el desarrollo de este proyecto se usan herramientas como lenguajes de programación, software, buenas prácticas de programación, metodologías de desarrollo, manejo de versionado, y demás tecnologías.

3.1. Configuraciones implementadas

Para este proyecto y con base a las recomendaciones que nos hace el líder de proyecto con su experiencia, se utilizan herramientas FOSS.

Para el procesamiento central del software tanto para el UAV, como su centro de carga, se usa el software propietario de la empresa corriendo sobre GNU/Linux en un sistema embebido.

Un sistema embebido es conveniente en este proyecto por el tamaño del UAV y que tiene que interactuar con diferentes dispositivos como la cámara, el manejador de la batería, entre otros, o como en el caso del centro de carga, que igual se utiliza un sistema embebido, manipular los diferentes motores para el cierre y apertura del mismo. Contar con un sistema operativo como lo es GNU/Linux, corriendo junto con nuestro software nos da muchas herramientas para usar software complementario para ser funcional y escalable el proyecto.

A continuación describiré cuales fueron las herramientas seleccionadas y el porqué de dicha elección.

3.2. Entorno de desarrollo

El desarrollo de este proyecto, se lleva a cabo sobre un sistema embebido con arquitectura ARM y montando un sistema operativo GNU/Linux en su distribución Ubuntu.

3.3. C++

Usar C++ es ideal para sistemas embebidos, ya que como mencioné en el marco teórico, combina la velocidad y control eficiente de hardware del lenguaje C con la capacidad de abstracción de la programación orientada a objetos, dándonos un lenguaje eficiente para el propósito del proyecto.

3.4. ROS 2

El uso del software ROS2 nos resulta conveniente para el desarrollo del software del proyecto pudiendo aislar el funcionamiento de los diferentes sectores del UAV. Se centra en dividir en secciones el sistema para que trabajen en sintonía cada uno con su propósito especifico como lo requiere un UAV revisando paralelamente diferentes tareas.

3.5. GNU/Linux - Ubuntu

La elección de usar Ubuntu como ambiente de trabajo, es por su versatilidad en el mundo de creación de software, su gran estabilidad y gran comunidad de desarrollo. Esta distribución tiene diferentes usos como lo son la computación en la nube, servidores, computadoras de uso general y dispositivos IoT. Siendo un sistema operativo muy liviano y pensado para poder correr con muy poco poder de cómputo, resulta una excelente elección para un ambiente de desarrollo en un sistema embebido.

3.6. Python

La elección de Python se debe a su flexibilidad y a la gran cantidad de bibliotecas que facilitan la implementación de funcionalidades útiles para el manejo de software. Python permite la modificación y ejecución directa de scripts, lo que resulta ideal para configurar parámetros del sistema de manera ágil. Esto elimina la necesidad de procesos de compilación intermedios tras cada cambio de configuración. Además, cuenta con una integración nativa y eficiente con el ecosistema de ROS2.

3.7. Docker

Los desarrolladores de software con el uso de Docker se despreocupan del sistema operativo que utilizan o la configuración del servidor donde se ejecutara su aplicación

y se pueden centrar en el desarrollo de software. El uso de este software es clave para poder tener versatilidad y que diferentes desarrolladores que se sumen al equipo puedan tener fácilmente su área de trabajo lista para empezar a desarrollar.

3.8. MQTT

La implementación del protocolo MQTT al proyecto es para lograr la intercomunicación entre el UAV y su centro de carga, siendo liviano viene bien para un sistema embebido. Logrando también poder obtener información en tiempo real como lo es la telemetría de los dispositivos.

Capítulo 4

Implementación y Pruebas

Para comenzar con la implementación de mis desarrollos se tuvo que pasar primero por una validación y verificación por parte de mi jefe directo, el cual, no solo me corregía sino también me orientaba a llevar por buen camino las ideas y plasmarlas en código, de tal manera que se pudieran llevar hasta la parte tangible dentro de la UAV o del centro carga. A continuación presento el proceso de simulación y ejecución del software desarrollado por mi.

4.1. Simulación y Ejecución del Software

Para poder realizar verificaciones del correcto funcionamiento del UAV se utiliza un software de simulación en conjunto con el software de la empresa, en donde se realizan pruebas funcionales de la operación de acuerdo a la solicitud de *features*, ya que, si estás pruebas fueran realizadas directamente en el UAV físico, el costo se elevaría considerablemente, por lo que usamos el software de simulación para desarrollo, posteriormente y al finalizar las configuraciones necesarias y pruebas relevantes, se procede a realizar la implementación en el UAV físico.

Durante el desarrollo de cada *issue*, para probar la implementación realizada ejecuto un *script* que inicia dos contenedores Docker, el primero es el software de la empresa y el segundo es el software especializado de piloto automático (Véase figura 4.1).

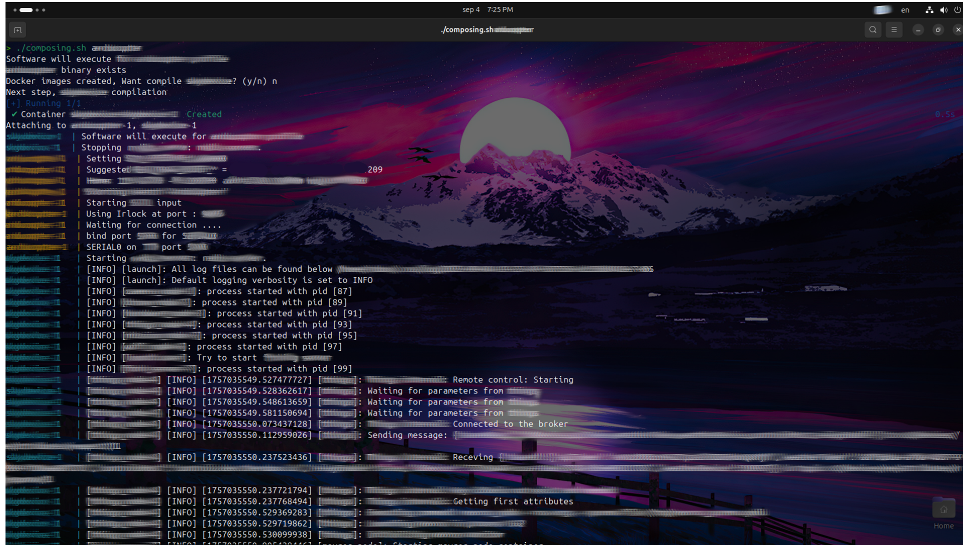

Figura 4.1: Ejecutando el software de piloto automático y el software de la empresa para simulación

Para el monitoreo de los vuelos se cuenta con un portal exclusivo para desarrolladores, que es diferente al portal al que tienen acceso los clientes, donde podemos mandar vuelos al dispositivo que estoy ejecutando en simulación, pero también se podría mandar a uno real (UAV físico).

Ya teniendo el software corriendo en simulación, empezamos creando una ruta para nuestro UAV en el portal (Véase figura 4.2).

Figura 4.2: Creación de ruta para el UAV en el portal de desarrollo

Y en la sección de vuelos (Véase figura 4.3), se agenda la ruta previamente creada para poder hacer pruebas de comportamiento.

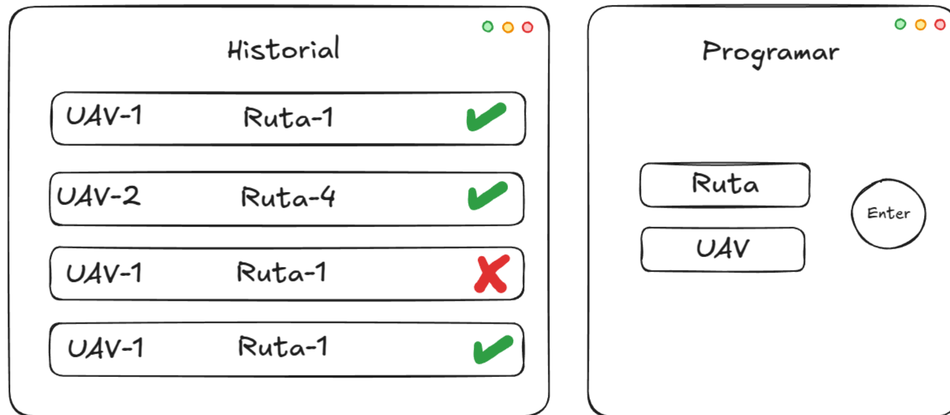

Figura 4.3: Ilustración de programación del vuelo

Posteriormente monitoreamos en tiempo real el vuelo en el portal (Véase figura 4.4), mientras también podemos revisar comportamiento en la terminal del software, tanto los *logs* del UAV como algunos errores que puedan surgir en las pruebas.

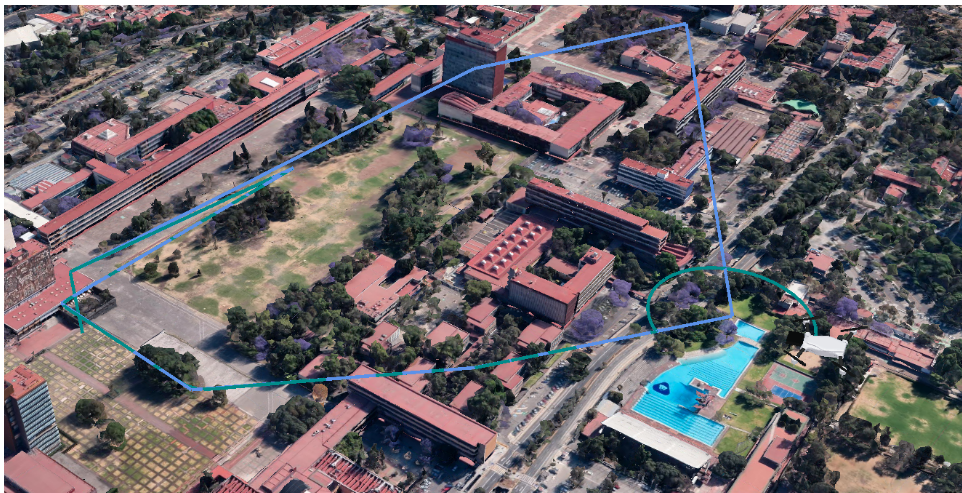

Figura 4.4: Monitoreo del vuelo

4.2. Buenas prácticas del desarrollo de software

Para ser participe del proyecto se necesita ser más consciente de las buenas prácticas de desarrollo para garantizar calidad, orden, y eficiencia del software. Así como se necesita disciplina, responsabilidad, enfoque profesional y saber colaborar con el equipo para lograr los objetivos del proyecto.

4.2.1. Mantras

Entre los desarrolladores de software a lo largo de los años han surgido filosofías para tener código limpio y eficiente que se han vuelto cultura, me gustaría mencionar los que más me han ayudado a ser buen desarrollador de software:

- KISS (*Keep It Simple, Stupid*) - Mantén las cosas simples.
- YAGNI (*You Aren't Gonna Need It*) - No programes lo que no necesitas aún.
- DRY (*Don't Repeat Yourself*) - No repitas código. Reutilízalo.
- SoC (*Separation of Concerns*) - Cada módulo/clase/función con una sola responsabilidad.
- *Defensive Programming* - Programa esperando errores, siempre validar.
- *Don't Make Me Think* - No me hagas pensar. El código debe ser fácil de entender, explicito, no usar nomenclaturas que nadie entienda.
- *Clean code* - Código limpio. Mantenlo fácil de leer.

UNIX Philosophy

Código simple, compacto, claro, modular y extensible.

4.2.2. Modelo V

De las principales bases que me fueron inculcadas fue el modelo V, como mencioné anteriormente es una guía para garantizar un buen desarrollo de software.

Contrario a lo que normalmente se piensa sobre el desarrollo de software, que solo es sentarte a programar, no es así, primero tenemos que conceptualizar los requerimientos del cliente o la operación, listar los requerimientos para una buena gestión de las expectativas posteriores a la implementación del nuevo desarrollo, de lo que esto conllevará en cuestión tiempo y cambios sobre el software actual, el siguiente paso sería diseñar la arquitectura para la implementación del *issue*, *feature* o proyecto, se tiene que diseñar en primera instancia a grandes rasgos, para depués pasar a diseñarlo a bajo nivel, teniendo toda la investigación y diseño pasamos al último punto que es desarrollo y codificación. Enseguida tenemos que hacer pruebas unitarias de cada una de las etapas como menciona el Modelo V para comprobar su funcionalidad y para concluir hacer pruebas de usuario final para ver la aceptación.

4.2.3. Pruebas Unitarias

Las pruebas unitarias son parte importante al momento de desarrollar una nueva funcionalidad, ya que, será una herramienta vital para poder probar nuestro código sin tener que estar compilando todo el proyecto y probando en conjunto, si no se tiene el resultado esperado de la nueva funcionalidad y se requiere estar haciendo cambios constantes para posteriormente probarlos, es mucho más sencillo compilar solo la prueba unitaria, mandar una nueva entrada y ver su comportamiento, esto agiliza el proceso para después implementarlo en el resto del proyecto.

4.2.3.1. Gtest

La herramienta que uso para las pruebas unitarias en una creada por Google, tiene una forma muy sencilla de usarla con C++.

El siguiente *test* lo desarrollé cuando tenía que estar realizando pruebas de la función `upload_file` que de igual manera implementé cuando se me pidió agregar la funcionalidad al UAV de poder subir archivos al servicio AWS (Amazon Web Services) de forma automática.

Para la validación de la función, la cual explicaré más adelante, implementé el caso de prueba `UploadFileTest`. En este declaré las variables esenciales que almacenan los parámetros para la prueba, y la funcionalidad de leer un archivo de configuración en donde puedo modificar los parámetros fácilmente sin necesidad de volver a compilar. La prueba consiste en probar la función 'a secas' y evaluar el resultado mediante `ASSERT_TRUE` que verifica que el resultado de la función sea *True*.

El Gtest se puede programar con diferentes herramientas para probar diferentes casos, pero en este en particular todas las pruebas las realicé cambiando los parámetros en el archivo de configuración. Primero probé con los parámetros correctos y el archivo que quería subir, lo cual resultó exitoso y empecé a darle diferentes entradas a la función, probando con credenciales falsas (se colocan en un archivo de configuración en el sistema), *buckets* inexistentes, *buckets* en donde no tenia acceso, escribiendo mal la ruta del archivo o hasta cambiando la extensión del archivo destino dentro del *bucket*. Teniendo las pruebas de todos los casos y verificando el comportamiento esperado, pude realizar la integración de la función al proyecto.

A continuación se puede observar el código ejemplo que desarrolle para la carga de videos automáticos.

```
1  #include <gtest/gtest.h>
2  #include <libconfig.h++>
3  #include "aws.h"
4
5  TEST(UploadFileTest, BasicTest) {
6
7      libconfig::Config cfg;
8      Aws::String bucket;
9      Aws::String region;
10     Aws::String source;
11     Aws::String remote_name;
12     bool result = false;
13
14     try {
15         cfg.readFile("tests.ini");
16     } catch(const libconfig::FileIOException& fileIoEx) {
17         FAIL() << "Error opening test config file: " <<
               fileIoEx.what();
18     } catch(const libconfig::ParseException& parseEx) {
19         FAIL() << "Error parsing test config file: " <<
               parseEx.what();
20     } catch(const std::exception& e) {
```

```
21          FAIL() << "Error while reading file" << e.what();
22      }
23  
24      const libconfig::Setting &root = cfg.getRoot()["
           UPLOAD_FILE_TEST"];
25  
26      root.lookupValue("bucket", bucket);
27      root.lookupValue("region", region);
28      root.lookupValue("source", source);
29      root.lookupValue("remote_name", remote_name);
30  
31      if(bucket.empty() || region.empty() || source.empty() ||
           remote_name.empty()) {
32          FAIL() << "Not all variables were assigned a value.
               Missing parameters.";
33      }
34  
35      result = upload_file(bucket, region, source, remote_name)
           ;
36  
37      ASSERT_TRUE(result);
38  }
39  
40  int main(int argc, char **argv) {
41      ::testing::InitGoogleTest(&argc, argv);
42      return RUN_ALL_TESTS();
43  }
```

Listing 4.1: Gtest en C++ para prueba de subida de archivos a AWS

4.2.4. Manejador de versiones

El uso de un manejador de versiones en el desarrollo de software es una herramienta esencial, ya que, te permitirá poder desarrollar software de forma organizada y trabajar colaborativamente teniendo todos su área de trabajo actualizado.

4.2.4.1. Git

En la empresa se tiene como manejador de versiones estrella, Git, se cuenta con un repositorio central y a cada uno de los desarrolladores nos corresponde un *branch* para trabajo local, aunque anteriormente ya trabajaba con dicho versionador, el uso en el ejercicio profesional, es diferente, en este caso, el poder colaborar con mi equipo de trabajo facilita el rápido crecimiento del software y elimina ciertos temas delicados al momento de conjuntar todos los cambios del software.

Para este proyecto era de suma importancia realizar el detalle del *commit* y posteriormente subirlo a la rama principal, la idea es que conforme se iba terminando el día laboral y se realizaron *commits* el mismo día, dichos cambios queden plasmados en el repositorio en la nube.

Git cuenta con una serie de comandos para poder manejar las versiones que tiendes a familiarizarte rápido con ellos. Aparte es soportado en las plataformas mas populares de alojamiento de proyectos de software como lo son GitHub y Gitlab.

De las principales costumbres que tienes que adoptar al momento de incorporarte a un proyecto en donde mas de una persona puede estar trabajando en una misma rama es tener actualizado tu proyecto local actualizado, siempre llegar a jalar los cambios del proyecto del servicio de la nube.

4.2.4.2. Issue/Feature

Al usar un manejador de versiones normalmente se usan servicios web de Git para albergar tu proyecto de software, en nuestro caso, se tiene un historial de issues/features pendientes y al momento de registrar uno nuevo necesitamos ser muy explícitos al momento de explicar de forma general el problema o mejora (Véase figura 4.5), poner el comportamiento actual, poner las posibles soluciones y si aplica los beneficios esperados.

Summary

(Summarize the bug encountered concisely)

Steps to reproduce

(How one can reproduce the issue - this is very important)

What is the current bug behavior?

(What actually happens)

What is the expected correct behavior?

(What you should see instead)

Relevant logs and/or screenshots

(Paste any relevant logs - use code blocks (```) to format console output, logs, and code, as it's very hard to read otherwise.)

Possible fixes

(If you can, link to the line of code that might be responsible for the problem)

/label bug ~reproduced ~needs-investigation

/cc @project-manager

/assign @QA-tester

Figura 4.5: Template para registro de un nuevo *Issue*

4.2.4.3. Ramas

Para el proyecto se tiene una rama protegida donde se alberga el software estable, donde se va agregando las nuevas funcionalidades desarrolladas, a partir de esa se

crean las ramas en donde se desarrolla individualmente ya sea un *issue* o un *feature*, nunca se usa una rama para diferentes propósitos de cambios, al terminar el objetivo de la rama se hace fusión a la rama estable de desarrollo. Para un historial limpio se puede ir etiquetando en diferentes puntos la versión que corresponda.

4.2.4.4. Merge Request

Al momento de terminar un cambio en una rama, se tiene que realizar un *Merge Request* (Véase figura 4.6) con las siguientes características: Descripción concisa de los cambios, su tipo (*bug fix*, *feature*, *refactoring*, etc.), como se prueban los cambios, un *check list* que aseguran un óptimo desarrollo (*coding standar*, *local testing*, etc), y relevante información. Para que posteriormente el encargado del área lo revise y te lo apruebe para que se haga la fusión de los cambios.

Summary

(Provide a clear and concise description of the changes you are proposing. This should include the purpose of the changes and any issues they are addressing)

Type of Change

- New feature
- Bug fix
- Refactoring
- Documentation
- Other (specify)

How to Test the Changes

(Describe the steps reviewers should follow to test your changes. This includes any setup that needs to be done beforehand, how to run the changes, and how to verify that the changes work as expected)

Verification Checklist

(Set of tasks or criteria that you confirm you have completed before submitting this request for review)

- Coding Standards
- Local Testing
- Unit/Integration Tests
- OTA requirements (Package needed, Roll back, etc)
- Documentation
- Change Log Entry

Relevant logs and/or screenshots

(Paste any relevant logs - use code blocks (```) to format console output, logs, and code, as it's very hard to read otherwise)

References

(Link to related task/ticket or any other relevant external reference for the changes (e.g., links to external resources that justify a library change))

Figura 4.6: Template para registro de un *Merge Request*

4.3. Desarrollo de *Issues/Features*

4.3.1. Software del UAV

4.3.1.1. Cargar archivos a S3 AWS

De mis primeros *features* relevantes fue implementar una funcionalidad para que el UAV, al finalizar cada misión, cargara de forma automática las fotos y videos capturados a los servicios S3 de AWS. Para la resolución de esta funcionalidad primero tuve que entender lo que era AWS, acudí a videos propietarios de Amazon en donde explicaban su servicio y como estaba estructurado. Teniendo los conceptos básicos, configuraciones del servicio, así como información y credenciales necesarias para usarlo, procedí a leer la documentación de AWS SDK para C++ y usar el servicio S3.

Teniendo los requerimientos y la información del servicio comencé a programar la función, recuerdo que tuve problemas porque fue el primer *feature* que me puso a prueba en habilidades fuera de lógica de programación, como por ejemplo, revisar que bibliotecas, funciones y configuraciones necesitaba, hasta que al final pude terminar con la funcionalidad.

Lo siguiente es la declaración de la función con los parámetros necesarios para tener una buena configuración de la carga de archivos en los diferentes casos:

```
bool upload_file(Aws::String bucket, Aws::String region, Aws::String source, Aws::String remote_name);
```

Listing 4.2: Declaración de la función

Así como podemos ver algunos fragmentos de código de la función:

```
if (!bucket.compare("") || !region.compare("") || !source.compare("") || !remote_name.compare(""))
    return false;
```

Listing 4.3: Asegurar parámetros validos

```
auto s3_client = Aws::MakeShared<Aws::S3::S3Client>("S3Client", clientConfig);
transfer_config.s3Client = s3_client;
transfer_config.uploadProgressCallback = UploadProgressCallback;
```

Listing 4.4: Variables de configuración

```
while (uploadHandle->GetStatus() != Aws::Transfer::TransferStatus::COMPLETED && (current_time - start_time) < 2)
{
    transfer_manager->RetryUpload(source, uploadHandle);
    uploadHandle->WaitUntilFinished();
    current_time = std::chrono::duration_cast<std::chrono::hours>(std::chrono::system_clock::now().time_since_epoch()).count();
```

```
}
```

Listing 4.5: Esperar a que se carguen los archivos y en caso de fallo reintentar

4.3.1.2. Promesas inválidas

Se llegó a un punto en el proyecto donde era momento de actualizar paqueterías, bibliotecas, archivos, entre otros, que son complementarios para aprovechar nuevas funcionalidades, tener actualizado el proyecto y poder seguir entregando mejoras en general al proyecto.

Para esto se actualizaron las dependencias, se hicieron las respectivas ramas de trabajo, etiquetado para diferenciar las versiones de compatibilidad, y al momento de compilar el proyecto, como era de esperarse, salieron muchos errores. Dieron algunos errores de bibliotecas que ya no existían, que habían cambiado el nombre, etc.

La actualización del proyecto se llevó a cabo primero analizando e investigando las actualizaciones y qué posiblemente conllevaría cambios en el código. Pero claro, no siempre se ve todo el panorama de cambios leyendo las documentaciones oficiales, si no hasta que ya estás en la travesía de cambio.

Después de resolver todos los errores de compilación, no servía nuestro software, al menos no como debería de funcionar. Existen los servicios de ROS2, es una forma de petición de un nodo a otro para que realice una acción y te regrese información, se les puede conocer como "promesas" de petición. Habiendo mencionado eso, el error que salía en la ejecución de nuestro software era *invalid future*. En ese punto se me encomendó investigar el porqué del error, este al ser de mis primeros *issues*, mi jefe inmediato me oriento y me dio algunos consejos de como abordar esta investigación.

La solución al error la encontré en foros, lo que sucedía era que al tener tu promesa, si tratabas de utilizar el método "get" de la promesa más de una vez se invalidaba (Ver el código 4,6), era algo que en anteriores versiones no sucedía. Para resolver este problema, en los casos donde se necesitaba usar la información, primero se creaba una copia de la información usando el método "get" y ya después se usaba cuantas veces fuese necesario.

Aquí se observa la creación un *request* asíncrono, se espera hasta 10 segundos a que el otro nodo realice la solicitud y se obtenga una respuesta:

```
auto future_result = fake_node->client_cam_status->
    async_send_request(std::make_shared<interfaces::srv::
    CameraStatus_Request>());
auto status = future_result.wait_for(10s);
if (status != std::future_status::ready)
    return false;

execute_other_function(future_result.get()->found);
std::cout << "Camera found: " << future_result.get()->found;
```

Listing 4.6: Ejemplo de error de invalidación de promesa

4.3.1.3. Cancelación de vuelo

Se me pidió tener la posibilidad de que los administradores del servicio pudieran cancelar vuelos antes del *takeoff*. En nuestro software teníamos todas las configuraciones del UAV antes del *takeoff* en una sola función. Este *feature* lo realicé cambiando la estructura de estas configuraciones mencionadas, hice un análisis e investigación para llegar a la decisión de emplear un patrón de diseño para una mejor estructura de estas configuraciones y poder cancelar fácilmente la misión. El patrón de diseño que emplee fue *command*, consideré que para modularizar el código vendría bien.

Para emplear el patrón de diseño, desarrollé una nueva función que recibe como parámetro la referencia a otra función, el propósito de esta función es ejecutar la función referenciada y revisar en cada ejecución si llegó el comando de cancelación de misión.

```
bool MainNode::checking_step_before_takeoff(bool (MainNode::*
    step_to_checking)(), bool *cancelled) {

    if (!((this->*step_to_checking)())) {
        return false;
    }

    if (mission_command == COMMAND_FLIGHT_CANCELLATION) {
        RCLCPP_WARN(this->get_logger(), "Canceling mission");
        *cancelled = true;
        return false;
    }

    return true;
}
```

Listing 4.7: Manejo de funciones

4.3.1.4. Forking Server

Por decisiones internas en la arquitectura del software se empezó a trabajar en un *forking server*. Se me pidió investigar qué tipo de *sockets* nos convenía en este *feature*, y se tomó la decisión de usar UNIX sockets, ya que al desarrollar en un ambiente GNU/Linux era la opción más eficiente y nativa para la comunicación entre procesos locales. Seguido de eso empecé a desarrollar el código.

La funcionalidad principal del *forking server*, como lo menciona la teoría, es crear un nuevo proceso hijo, haciendo `FORK`, para que atienda cada conexión de cliente entrante. De esta forma, el proceso hijo se encarga de toda la comunicación con ese cliente en particular, mientras que el proceso padre queda libre para volver a escuchar y aceptar más conexiones de forma concurrente.

La creación de este *forking server* de inicio a fin fue mi responsabilidad, tuve que consultar la teoría e investigar. En todo este proceso mi jefe inmediato me supervisaba y me oriento dando los requerimientos así como también me indicaba

que código cambiar o mejorar. Desarrollé tanto el lado del servidor como el del cliente para una satisfactoria comunicación entre procesos.

El desarrollo de esta mejora solucionaba necesidades propias del software, y a continuación mencionaré algunos puntos de código:

Primero se realiza la configuración de los *callbacks* para obtener los códigos de salida de los procesos creados mediante `FORK`, así como la configuración del *socket*.

```
void launch_server() {
    signal(SIGINT, signal_handler);
    signal(SIGCHLD, signalchld_handler);

    int server_socket;
    fd_set active_sockets, read_sockets;
    const char *socket_path = SOCKET_PATH;

    server_socket = socket(AF_UNIX, SOCK_STREAM, 0);

    ...
}
```

Listing 4.8: Configuración inicial del *forking server*

Cuando se leen mensajes mediante *UNIX sockets*, se tiene que recibir primero la cantidad de bytes a leer y enseguida, ya con esa información saber hasta que cantidad de bytes leer. Lo programé para que estuviera leyendo la petición hasta que recibiera `FORK_CODE_END_OF_CMD`, que es el que definí como el final de la petición.

```
    while (true) {

        ...

        bytesRead = recv(descriptor, &msg_len, sizeof(size_t)
            , 0);
        if (bytesRead <= 0)
            goto response;

        if (static_cast<int>(msg_len) == FORK_CODE_END_OF_CMD
            )
            break;

        char* message = new char[msg_len + 1];
        bytesRead = recv(descriptor, message, msg_len, 0);
        if (bytesRead <= 0)
            goto response;

        exec_args.push_back(message);
    }
```

Listing 4.9: Lectura de un tipo de petición

El siguiente *callback* se ejecuta cuando se recibe un código de salida de algún proceso generado en el servidor, entonces se recorre los PIDs de los procesos lanzados,

se solicita el código de salida de todos y el que diga que ya terminó se guarda en un mapa.

```
void signalchld_handler(int sig) {
    ...

    for (int itr_pid : running_processes) {
        result = waitpid(itr_pid, &statusPid, WNOHANG);

        if (result <= 0)
            continue;

        if (WIFEXITED(statusPid))
            mapPidsCodeExit[itr_pid] = WEXITSTATUS(statusPid)
                ;
    }

    ...
}
```

Listing 4.10: Callback de child signal

Este procedimiento es fundamental en un *forking server* para la correcta recolección de los procesos hijos y la prevención de "procesos zombis". Cuando un proceso hijo termina, envía una señal `SIGCHLD` al padre. Sin embargo, si múltiples hijos terminan casi al mismo tiempo, puede que se agrupen estas notificaciones en una sola señal. Entonces en el callback `signalchld_handler` no se puede asumir que solo un hijo ha muerto, se debe iterar en el registro de todos los procesos e ir preguntando el estado de cada uno.

4.3.1.5. Control de streaming

El UAV cuenta con la funcionalidad de tener *streaming*, se puede activar por diferentes medios:

- Comando vía MQTT
- Prioridad
- Configuración de la misión

La situación es que el *streaming* se puede prender y apagar por los diferentes medios anteriormente mencionados. Para resolver el problema emplee un manejo de *bitmask* (Véase figura 4.7), donde se crea una variable de tipo byte, en el que le asignas un bit a cada componente que puede prender y apagar el *streaming*. Cuando se prende vía MQTT es porque hay una persona viendo el *streaming*, se prende el bit que le corresponde, y así con todos los bits. Esto hace mas fácil el manejo, si todos los bits están apagados y llega el comando "prender stream", prendes el bit y prendes el *streaming*, si algún bit ya esta prendido, solo prendes el bit que toca y no mandas a prender *streaming*. Y para apagar, se quita el bit y si todos los bits estan en cero, se puede apagar el *streaming*.

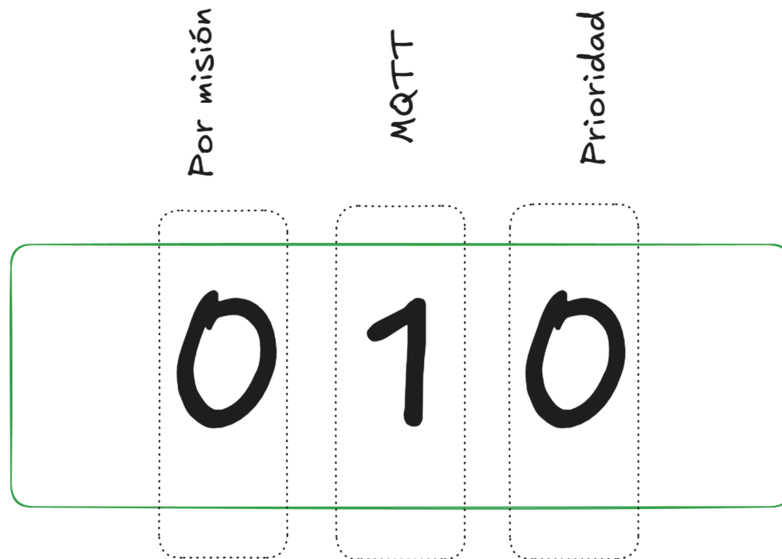

Figura 4.7: Bitmask

4.3.2. Software del centro de carga

4.3.2.1. Algoritmo de manejo de motor a pasos

Este *feature* para la caja autónoma del UAV es sobre un mejor manejo de motores con *encoder* y realizar la apertura de unas compuertas, a diferencia de otros motores dentro de la caja, estos que manejaban dichas compuertas solo tenían *switch* de cerrado que nos indicaba cuando parar el motor (las demás etapas con motores tenían *switch* de cerrado y apertura), pero no teníamos referencia de cuando dejar de mover el motor. De aquí surgen 2 *issues* relevantes:

- Adaptación de algoritmo de Python a C++
- Algoritmo de apertura de compuertas

Se tenia un *script* de Python para mover estos motores, con el que se estuvo probando en etapas tempranas de la implementación de los motores con *encoder*. Este *issue* conllevó pasar el *script* de Python a C++, enfocado a la parte de las bibliotecas que se usaban en Python, las cuales no existían en C++. Más en especifico fue crear el algoritmo de obtención del control PID (Proporcional-Integral-Derivativo) para un mejor manejo de los motores, es un sistema de retroalimentación que ajusta la velocidad del motor conforme a la posición de dicho motor dependiendo de la configuración de los parámetros. Para la transcripción de este algoritmo, tuve que revisar Python y recurrir al código fuente de la biblioteca y extraer la parte del algoritmo para posteriormente programarla en C++. El cambio de lenguaje no fue directo y tuve que indagar un poco en el algoritmo original para resolver unos puntos ya que en la biblioteca de Python resolvía este algoritmo usando otras funciones y bibliotecas, entonces tuve que realizar una investigación para poder completar satisfactoriamente el algoritmo en C++.

```
1 double PID::calculate_pid(int speedRpm, double currentTime) {
```

```
2       double deltaTime = (currentTime - lastTime) / 1000;
3
4       if (deltaTime == 0)
5           deltaTime = 1e-16;
6
7       ...
8
9       if (firstRound) {
10          d_speedRpm = speedRpm - speedRpm;
11          d_error = error - error;
12
13          firstRound = false;
14      } else {
15          d_speedRpm = speedRpm - lastSpeedRpm;
16          d_error = error - lastError;
17      }
18
19      double proportional;
20
21      ...
22
23      double derivative;
24
25      ...
26
27      else
28          derivative = kd * d_error / deltaTime;
29
30      double pidOutput = proportional + integral + derivative;
31
32      ...
33
34      return pidOutput;
35  }
```

Listing 4.11: Partes del algoritmo para control PID

Junto a la par investigué como trabaja un motor con *encoder*. Se tiene el canal A y B (Véase figura 4.8), dependiendo de las lecturas que hagas puedes determinar de forma sencilla hacia donde esta girando el motor, por ejemplo, si lees que el pin A esta en alto y el pin B esta en alto, esta girando hacia un sentido pero si estuviera en bajo el pin B estaría girando hacia el otro lado. Pero como se controló hacia donde se esta girando, en esta investigación que realice solo es relevante el conteo de los pulsos. Ya que con esto pude resolver la parte de la apertura y cerrado de las compuertas. Desarrolle un algoritmo donde se llevaba un conteo en tiempo real de las vueltas, con esto, se definió hasta qué número de vuelta era abierto y ya no se tuvo problema por la ausencia del *switch* de apertura.

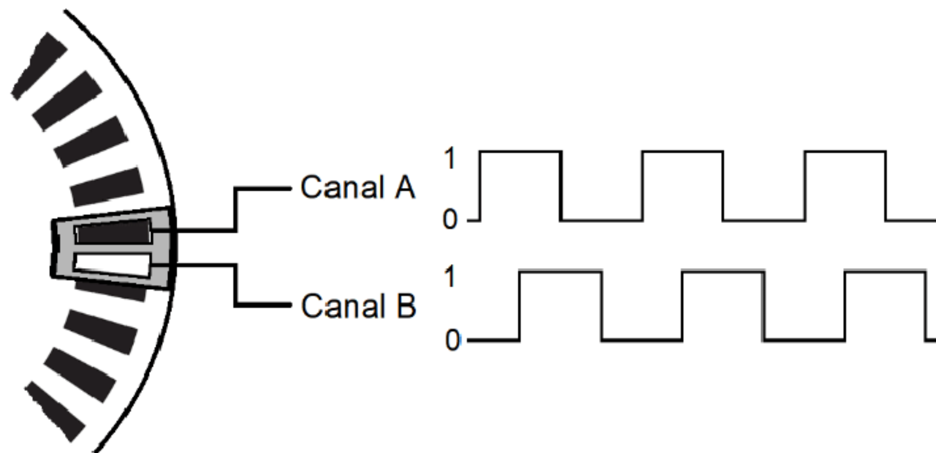

Figura 4.8: Señales de los canales A y B de un encoder de cuadratura («Señales de los canales A y B de un encoder de cuadratura», 2020)

Capítulo 5

Resultados obtenidos y Conclusiones

Este proyecto me ha ayudado a crecer como Ingeniero en Computación especializado en Software. La Facultad me dio tanto las herramientas como la disciplina para ser parte de este gran proyecto, las diferentes asignaturas y profesores me han enseñado bases sólidas del desarrollo de software y este proyecto ha dado forma y sentido a todos los conocimientos adquiridos en la Facultad de Ingeniería.

Como todo, al inicio de incorporarme al proyecto tenía responsabilidades básicas que no implicaran gran impacto en el software. Conforme ha pasando el tiempo he sido más consciente de la importancia de la calidad, el orden y la colaboración en el desarrollo del software. Esto ha sido fundamental en mi desarrollo como Ingeniero en Computación, me ha permitido adquirir y poder aplicar buenas prácticas de programación, así como fortalecer mi disciplina, responsabilidad y enfoque profesional en el desarrollo de software de calidad.

Cabe destacar que gracias a las buenas prácticas que se fomentan durante el desarrollo de los proyectos en la empresa donde laboro, puedo tener un mejor desempeño en la Facultad y como ejemplo puedo citar a mi clase de Computación Gráfica, que gracias al proyecto final encomendado, pude implementar la metodología de desarrollo del modelo V, obteniendo resultados de gran calidad.

Algo que he aprendido en el trabajo es gestión de tiempos y aprender a ser el puente entre las peticiones del cliente con la conversión a lo que implica tanto impacto en el software como el tiempo que tomarán ciertos cambios. Me han pedido realizar cambios y lo que tengo que hacer es analizar todos los cambios que conlleva dicha solicitud y poco a poco he logrado afinar ese sentido de estimar plazos y poder decir a mi superior en cuánto tiempo aproximado podría tener los cambios.

Mi perfil como desarrollador de software se ha fortalecido notablemente. Los fundamentos de lógica, estructuras de datos y algoritmos adquiridos en la Facultad se han vuelto de carácter profesional gracias a este proyecto. He comprendido que la calidad del software reside en los detalles: desde la nomenclatura de una variable hasta la implementación de metodologías y algoritmos complejos.

Mis responsabilidades han ido escalando y he tenido que empezar a pensar fuera de mis límites que sólo resolver *issues*, sino hacer que los cambios vayan más allá y mantengan el software escalable, lo cual antes el maintainer se encargaba más en

revisar esas partes en mis issues o features, he pasado de ejecutador a diseñador.

Conocimientos de gran valor obtenidos en el proyecto puedo empezar mencionando Git, si bien el uso de Git se fomenta dentro de la Facultad de Ingeniería, en mi experiencia solo tuve acercamiento al manejo con interfaz gráfica, y sin lograr entender el porqué usarlo. Al ser parte de este proyecto fue forzoso el aprendizaje a nivel medio del manejo de Git en consola, así como las buenas prácticas para su uso y trabajar colaborativamente con otros desarrolladores en un mismo proyecto.

El código heredado es algo que fue un trabajo importante para mi, no estaba acostumbrado a leer código que otras personas desarrollaron y menos proyectos tan grandes, me hizo poner a prueba habilidades que no sabía que se tenían que tener en la Ingeniería de Software, o al menos no lo tenía tan presente, había técnicas para leer código heredado, cosa que aprendí con este proyecto.

Los issues mencionados o en general mi experiencia profesional, me han enseñado que ser Ingeniero de Software no es ser el mejor programador, es emplear diferentes conocimientos para encontrar una solución astuta y eficaz a ciertos problemas enfocados al software.

Este proyecto me ha enseñado que tengo bases fuertes gracias a la Facultad de Ingeniería, ya que entrar a un proyecto de carácter profesional es ese ultimo "paso" donde terminas de darte cuenta la razón de cada escalón dado durante la carrera universitaria, el porqué pasaste por eso, y terminas desarrollando ese sentido de como usar todos los conocimientos obtenidos para ser Ingeniero y resolver problemas. Y aun así nunca terminas de aprender nuevas o afinar habilidades.

El conjunto de todas las herramientas y conocimientos mencionados en este trabajo escrito y los que están implícitos en mi carrera universitaria son igual de vitales en mi ser, convirtiéndome en una mejor versión de mi como Ingeniero.

Glosario

- Bit: La unidad de datos más pequeña en informática (0 o 1).
- Byte: Conjunto de 8 bits.
- Callback: Función que se pasa como argumento a otra, para ser ejecutada más tarde.
- Commit (Git): Operación que guarda un conjunto de cambios en el historial de un proyecto de forma permanente.
- Consola: Interfaz basada en texto para interactuar con un sistema operativo mediante comandos.
- Feature: Una funcionalidad o capacidad nueva que se añade a un programa o sistema.
- Fork (Creación de procesos): Acción en la que un proceso crea una copia exacta de sí mismo para ejecutar una nueva tarea.
- IoT (Internet de las Cosas): Red de dispositivos físicos cotidianos conectados a Internet para intercambiar datos.
- IP (Internet Protocol): Protocolo para dirigir y enrutar paquetes de datos en una red.
- Issue: Registro de una tarea, error o mejora en un proyecto de software. Sirve para organizar el trabajo.
- PID (ID de proceso): Número único que el sistema operativo asigna a cada proceso en ejecución.
- Procesador: Componente principal de un ordenador que ejecuta instrucciones y procesa datos.
- Protocolo: Conjunto de reglas que definen cómo deben realizarse ciertas acciones en un contexto específico.
- Script: Conjunto de instrucciones o comandos que se ejecutan de forma secuencial para automatizar una tarea.
- Sistema Operativo: Software principal que gestiona el hardware y los recursos de un dispositivo.

- Streaming: Consumo de contenido multimedia directamente desde Internet sin necesidad de descargarlo por completo.
- Takeoff: Momento de despegue, es la acción de elevar la aeronave desde el suelo o una superficie para comenzar el vuelo.
- TCP (Transmission Control Protocol): Protocolo de red que asegura la entrega completa y ordenada de datos.
- TI (Tecnologías de la Información): Campo que abarca el uso de sistemas informáticos, redes y software para gestionar y procesar información.
- UDP (User Datagram Protocol): Protocolo de red rápido que envía datos sin verificar la entrega.

Bibliografía

¿Qué es MQTT? [Recuperado el 15 de octubre de 2025]. (2024). https://aws.amazon.com/es/what-is/mqtt/

Aguinaga, Á. (2020). *Lenguajes de programación de bajo nivel vs. alto nivel* [Recuperado el 09 de septiembre de 2025]. https://cipsa.net/lenguajes-de-programacion-de-bajo-nivel-vs-alto-nivel/

Alonso, R. (2025). *Arquitectura Von Neuman: qué es, cómo funciona y para qué sirve* [Recuperado el 10 de septiembre de 2025]. https://hardzone.es/tutoriales/rendimiento/von-neumann-limitaciones/

Amazon S3 [Recuperado el 15 de octubre de 2025]. (2025). https://aws.amazon.com/es/s3/

Barahona, J. M. G. (2021). *A Brief History of Free, Open Source Software and Its Communities* [Recuperado el 08 de septiembre de 2025]. https://www.computer.org/csdl/magazine/co/2021/02/09353517/1r8kwgBjU9W

Carvalho, C. (2023). *¿Qué es Python? Historia, sintaxis y una guía para iniciarse en el lenguaje* [Recuperado el 09 de septiembre de 2025]. https://www.aluracursos.com/blog/que-es-python-historia-guia-para-iniciar

Corvo, H. S. (2024). *Arquitectura Harvard* [Recuperado el 10 de septiembre de 2025]. https://www.lifeder.com/arquitectura-harvard/

González, I. G. G. (2006). *Levantamientos Topográficos* [Material educativo interno, Palacio de Mineria]. http://www.ptolomeo.unam.mx:8080/xmlui/bitstream/handle/132.248.52.100/15279/decd_4757.pdf?sequence=1&isAllowed=y

Hat, R. (2022). *Procesadores ARM* [Recuperado el 10 de septiembre de 2025]. https://www.redhat.com/es/topics/linux/what-is-arm-processor

Johnston, S. (2024). *11 Years of Docker: Shaping the Next Decade of Development* [Recuperado el 10 de septiembre de 2025]. https://www.docker.com/blog/docker-11-year-anniversary/

Josh Schneider, I. S. (2024). *¿Qué es un microprocesador?* [Recuperado el 05 de septiembre de 2025]. https://www.ibm.com/mx-es/think/topics/microprocessor

Leninmhs. (2023). *La historia de Git: el sistema de control de versiones que cambió el mundo de la programación* [Recuperado el 10 de septiembre de 2025]. https://www.docker.com/blog/docker-11-year-anniversary/

Luna, J. I. V., González, R. S., Guzmán, G. S., & González, L. A. S. (2022). Arquitectura RISC y sus aplicaciones. *Revista de Ingeniería.*

ROS 2 Developers. (s.f.). *ROS 2 Documentation: Jazzy* [Consultado el 04/09/2025]. https://docs.ros.org/en/jazzy/index.html

Sami, M. (2018). *The Validation and Verification Model – The V-Model* [Recuperado el 09 de septiembre de 2025]. https://melsatar.blog/2018/08/27/the-validation-and-verification-model-the-v-model/

Sandoval, V. D. (2017). *Evidencias de hornos alfareros en pedernales: una interpretación de la producción cerámica* [Recuperado el 04 de septiembre de 2025]. https://www.researchgate.net/publication/362954185_Evidencias_de_hornos_alfareros_en_pedernales_una_interpretacion_de_la_produccion_ceramica

Señales de los canales A y B de un encoder de cuadratura [Recuperado el 11 de octubre de 2025]. (2020). https://www.researchgate.net/figure/Figura-1-Senales-de-los-canales-A-y-B-de-un-encoder-de-cuadratura_fig1_348374766

Sockets de dominio UNIX [Recuperado el 15 de octubre de 2025]. (2024). https://www.ibm.com/docs/en/ztpf/1.1.2023?topic=considerations-unix-domain-sockets

Stallman, R. (2021). *Linux and the GNU System* [Recuperado el 09 de septiembre de 2025]. https://www.gnu.org/gnu/linux-and-gnu.es.html

TRBL Services. (2021). *Sistemas embebidos y sus características — Conceptos fundamentales* [Recuperado el 04 de septiembre de 2025]. https://trbl-services.eu/blog-sistema-embebido-caracteristicas/

van den Beuken, F. (2023). *Una breve historia de C++* [Recuperado el 09 de septiembre de 2025]. https://www.perforce.com/blog/qac/misra-cpp-history