ESCUELA DE INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Desarrollo de una estación meteorológica y un sistema orientador para una plataforma robótica de toma de datos en un Sistema de Posicionamiento en Interiores (IPS)

Titulación: Máster en Ingeniería en Informática

Curso: 2019-2020

Autor: David Suárez Suárez

Tutores: Gabriel de Blasio García y Alexis Quesada Arencibia

Fecha: 12 de noviembre de 2020

Índice de Contenido

Índice de Figuras

Agradecimientos

En la elaboración y desarrollo de este Trabajo de Fin de Máster (TFM), he tenido la suerte de contar con el notable apoyo de varias personas o entidades. No solo me han animado en la realización de mi trabajo, sino que me han facilitado herramientas, materiales y procedimientos. Por todo lo anterior y lo que me queda por comentar, cuentan con mi agradecimiento.

A mi familia, que, bajo su margen de actuación limitado, expresa siempre su mayor esfuerzo para estar presente siempre que lo necesito, animándome a continuar y apoyando mis decisiones vitales.

A todo el Instituto Universitario de Ciencias y Tecnologías Cibernéticas (IUCTC), no solo por brindarme la oportunidad de realizar mi TFM con ellos y facilitarme el material necesario para ello, sino por las buenas relaciones y prácticas que fomentan.

Dentro del IUCTC, especiales agradecimientos a mis dos tutores: Alexis Quesada Arencibia y Gabriel de Blasio García. No solo son dos grandes profesionales, sino que además son dos grandes personas.

A todos los profesores y personas involucradas en mi proceso de formación, tanto a los profesores del Grado en Ingeniería Informática como a los del Máster en Ingeniería Informática.

A las diferentes compañías en las que he podido adquirir bastante material para proceder con el proyecto. Sin ellas, me hubiera costado un mayor tiempo y esfuerzo.

Resumen

A continuación, se expone el resumen de este TFM, tanto en versión española como inglesa:

- Este proyecto se enmarca en el ámbito de los Sistemas de Posicionamiento en Interiores (IPS) mediante tecnologías *Bluetooth Low Energy* (BLE) y Wi-Fi. Una de las líneas de trabajo en este campo requiere de un procedimiento que haga un barrido exhaustivo del entorno mediante una plataforma robótica recopilando información de la intensidad de la señal recibida (RSSI) de las balizas BLE.

  Este proyecto se centrará en el análisis, diseño e implementación de una estación meteorológica y de un módulo orientador a incorporar en una plataforma robótica. La importancia de estos dos subsistemas se debe a que tanto las variables meteorológicas como la orientación juegan un papel clave en la naturaleza de las señales recibidas en los sistemas IPS.

- This project belongs to the field of Indoor Positioning Systems (IPS) using Bluetooth Low Energy (BLE) and Wi-Fi technologies. One of the lines of work in this field requires a procedure that makes a detailed scanning of the environment through a robotic platform collecting information about the intensity of the received signal (RSSI) of the BLE beacons.

  This project will focus on the analysis, design and implementation of a weather station and an orientation module which will be integrate to the robotic platform. These two subsystems are very important, due to the fact that both the meteorological variables and the orientation are key elements of the nature of received signal in IPS systems.

1. Introducción

En este capítulo se presenta la vista global donde se enmarca este TFM, pasando por su contexto, sus objetivos esenciales y la estructura global de este documento.

1.1. Contexto

Entre la multitud de cosas que hacen únicos a los seres humanos, tenemos la tecnología, cuyo crecimiento en las últimas décadas ha sido abrumador a la vez que atractivo y beneficioso. Naturalmente, dentro del concepto de *tecnología* podemos encontrar infinidad de elementos, por lo que se centrará el foco en ciertos aspectos de interés para este proyecto.

No es nada raro que, en la actual sociedad, las personas usen su dispositivo móvil (*smartphone*) para realizar tareas cómoda y eficientemente. Dejando volar la imaginación, se pueden realizar muchas labores de forma *online*, desde la compra de productos hasta el manejo de citas previas con nuestro médico de cabecera.

Aterrizando en territorio propio de este TFM, otra de las mil y una utilidades que podemos utilizar en nuestro *smartphone* es el posicionamiento y la navegación. Estos conceptos permiten llegar de un origen a un destino de una forma rápida y eficiente, principalmente cuando no se conoce la ruta objetivo. No solo nuestro dispositivo móvil nos brinda esta utilidad, tenemos: el ordenador de a bordo de los coches, los sistemas de navegación tradicionales (GPS), entre otros.

Si nos paramos a pensar un segundo, se puede llegar a varias conclusiones. Las señales GPS y los dispositivos que se han comentado permiten, con un grado de precisión razonable, llegar de un origen a un destino más o menos lejano. Sin embargo, esta afirmación cubre el ámbito de exteriores, es decir, un entorno de *techo descubierto*.

La pregunta que se nos plantea ahora es: ¿qué pasa si se está en el interior de un edificio cualquiera (museo, instituto, universidad, teatro) o incluso en ambientes bajo tierra como en el metro y no se sabe cómo llegar a nuestro destino (obra concreta, aula concreta, sala concreta)? Obviamente, no podemos hacer uso del GPS, principalmente porque en la mayoría de los casos no disponemos de señal GPS. Aunque se dispusiera de ella, no brindaría la precisión adecuada ni se conocería el contexto del interior del edificio en el que nos encontramos (organización del lugar a lo largo del tiempo, entre otros). En definitiva, es necesario el uso de otro tipo de tecnología más adecuada y preparada para este escenario.

Este es el punto álgido de este apartado, sobre el cual se desarrolla nuestro TFM. A grandes rasgos, se pretende desarrollar parte de un Sistema de Posicionamiento en Interiores o IPS (*Indoor Positioning System*), que cuenta principalmente con varias balizas Bluetooth, una plataforma robótica y diferentes subsistemas que aportan información relevante de cara a mejorar el resultado del sistema planteado. Los módulos o subsistemas particulares en los que se centran nuestros esfuerzos son: estación meteorológica y módulo orientador. Por la esencia propia de nuestros objetivos de trabajo, el GPS pierde el protagonismo y pasa a un segundo plano.

1.2. Objetivos

Durante la realización de este TFM, se pretende alcanzar una serie de objetivos clave. El objetivo esencial y que realmente se sobreentiende es el simple hecho de aportar y contribuir al avance del proyecto en cuestión (IPS), desarrollando algunos de sus módulos, ya introducidos en el anterior apartado. El resultado de este objetivo es, naturalmente, el producto esperado.

Básicamente, los objetivos son el análisis, diseño e implementación de una *estación meteorológica* y de un *módulo orientador*. La importancia del desarrollo de estos módulos se verá a lo largo de este breve apartado, sin entrar en demasiados detalles para evitar robar contenido a futuros capítulos y/o apartados.

Como se puede intuir a raíz de todo lo anterior, así como el posicionamiento en exteriores está prácticamente solucionado y estandarizado, el posicionamiento en interiores está aún en proceso de maduración. Actualmente existen diferentes líneas de investigación que pretenden dar solución a este problema, cada una de las cuales expone un planteamiento razonable y lógico a pesar de las diferencias que plantean.

De forma muy escueta, nuestra solución toma la vía que consiste en la recopilación exhaustiva de datos y variables (en diferentes puntos del entorno) seguida de una fase de procesamiento. Cada punto del entorno presentará variaciones frente al resto y frente a sí mismo en diferentes puntos temporales. Un ejemplo sencillo para esto último podría ser la diferencia de temperatura en un mismo punto a primera hora de la mañana (más fresca) y al mediodía (más cálida).

En la primera fase de recopilación de datos, se seleccionan unos puntos estratégicos del entorno y se realiza, sobre ellos, una lectura de toda la información que se crea relevante (temperatura, orientación, intensidad de la señal Bluetooth recibida…). Con esa información en nuestro poder, le llega el turno a la fase de procesamiento. En esta fase, se realiza una lectura de datos (exacta a las anteriores) para la posición actual y, mediante alguno de los métodos de procesamiento disponibles, se logran estimar los valores de nuestra posición actual. En definitiva, estos métodos comparan los datos recabados previamente y los obtenidos para una posición concreta en busca del mayor grado de *coincidencia* de los datos.

Debido a que nuestro IPS se basa en una plataforma robótica automatizada encargada de la lectura de señales Bluetooth provenientes de diferentes puntos del entorno, es lógico querer contemplar cada una de las variables que pueden afectar a la naturaleza de esas señales Bluetooth para maximizar la precisión del proceso.

Esta es la causa por la que la estación meteorológica y el módulo orientador son necesarios en nuestro IPS, ya que algunas variables ambientales y la orientación en la que se realiza la lectura de las señales Bluetooth pueden afectar considerablemente a la naturaleza de los datos, principalmente a la intensidad de la señal recibida. Considerando esto, se pueden interpretar los datos de una manera más precisa para garantizar el mejor resultado posible.

La mayoría de los trabajos que se realizan durante la vida de un estudiante de informática consisten en plasmar unas especificaciones predefinidas en un producto final (usando algún tipo concreto de tecnología). En este proyecto, sin embargo, nosotros tenemos el testigo para analizar la situación y aportar una solución que se crea oportuna (con ciertos límites, evidentemente). Aparte de aplicar los conocimientos aprendidos a lo largo del Grado y del Máster en Ingeniería Informática, se experimenta parte del *sabor* que sienten los profesionales e investigadores del sector.

1.3. Estructura de la memoria

Una vez introducido el TFM y sus objetivos, pasamos a detallar las diferentes partes que componen esta memoria.

En el capítulo 2 se introduce el estado del arte de este TFM, incluyendo información actual de relevancia para el proyecto, que ayude a comprender y abordar sus objetivos esenciales. Se tratan el posicionamiento en exteriores y el posicionamiento en interiores.

En el capítulo 3 se incluye toda la información sobre las herramientas utilizadas durante la realización de este TFM, la mayoría son aplicaciones auxiliares para poder ejecutar todas nuestras tareas.

En el capítulo 4 se incluye todo el contenido relacionado con Normativa y Legislación, fuertemente relacionado con el anterior capítulo.

En el capítulo 5 se incluyen todas las competencias que están "*grabadas a fuego*" en la especificación y ejecución de este TFM.

En el capítulo 6 se incluye la información directamente relacionada con la metodología empleada y la planificación de tareas seguidas a lo largo del desarrollo.

En el capítulo 7 se presenta la columna vertebral de nuestro trabajo, es decir, las diferentes fases de desarrollo del producto esperado. En este caso particular, se presentan varios subapartados para cubrir los detalles de cada uno de los dos módulos a desarrollar.

En el capítulo 8 se presentan las conclusiones y las perspectivas de futuro del trabajo realizado.

Por último, se presentan cinco anexos como complemento informativo para este documento: API, Manual de instalación, Especificación de casos de uso y Patrones de diseño y *clean code*.

2. Estado del arte

En este capítulo se presentará el estado en el que se encuentra el ámbito de desarrollo de este TFM.

2.1. Posicionamiento en exteriores

Antes de comenzar con este apartado, tenemos que hacer referencia a dos conceptos: *nomadismo* y *sedentarismo*. Con el tiempo, el ser humano ha evolucionado y ha pasado de ser nómada a ser sedentario. Dicho de otro modo, de vivir de la caza a vivir de la agricultura y la ganadería, aunque la caza siempre sigue presente (complemento).

El nomadismo (1) se puede definir como *la condición de nómada, que es característica de algunos pueblos o grupos étnicos*. Ser nómada (2) consiste básicamente en no tener un lugar de asentamiento estable, sino estar en continuo desplazamiento (asentamientos temporales en determinados lugares).

Por otro lado, el sedentarismo (3) se puede definir como la *actitud de la persona que lleva una vida sedentaria.* Básicamente, ser sedentario (4) significa todo lo contrario a ser nómada, es decir, tener por asentamiento un lugar fijo. Es cierto que podemos cambiar ese lugar en determinados momentos, pero una vez escogido un lugar, se establece como nuestro *hogar*.

En cualquier caso, la orientación, la navegación y el posicionamiento juegan un papel clave en las dos posturas. Sin el uso de la más mínima orientación, es probable que, si alguien quiere dirigirse a un destino concreto (o regresar a su hogar en el caso del sedentario), no lo logre. En el caso de los primeros humanos, dirigirse en la dirección incorrecta puede suponer la muerte, por lo que no hay que restar importancia a esto.

Es aquí donde entramos a la miga del asunto. El ser humano siempre ha generado soluciones a este problema. Por ejemplo, dejar marcas en los árboles por los que se pasa para saber regresar, es una forma sencilla de orientarse y posicionarse, pero no es la única. A continuación, se presentan soluciones más complejas (inventos y conocimientos):

- **Astronomía** (5). Es la ciencia que estudia la estructura y la composición de los astros, su localización y las leyes que rigen sus movimientos. Con estos conocimientos, es posible determinar no la ubicación, pero sí hacia donde nos movemos. Entre otras cosas, nos permite determinar el norte y determinar a dónde debemos dirigirnos.

  Existen varias constelaciones, pero algunas destacan sobre otras. Las principales son la *Osa Mayor* y la *Osa Menor*, gracias a las cuales podemos encontrar la *Estrella Polar* (estrella del norte), estrella hacia la que casualmente apunta el eje de rotación de la Tierra (casi a la perfección) (6). Sin exponer más detalles al respecto, nos quedamos con esta *pincelada de conocimiento* y dejamos una muestra gráfica en la figura 2.1.

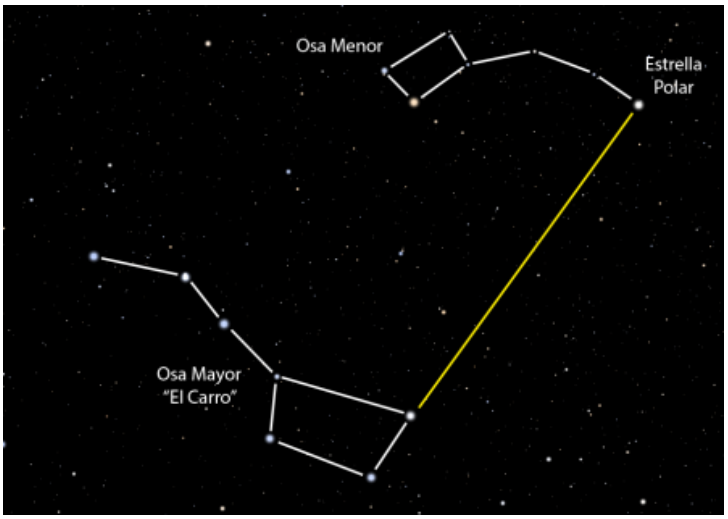


Figura 2.1. Ilustración de los astros principales en el ámbito de la navegación.

Alternativamente y durante el día, podemos deducir nuestra posición por el sol, puesto que sabemos que sale por el este y se esconde por el oeste. Para esto, existen diferentes métodos, variantes en complejidad y fiabilidad (7).

- **Astrolabio y Brújula** (8). Son dos instrumentos físicos partícipes en los conceptos comentados hasta ahora.

    - **Astrolabio**: Uno de los principales inventos de navegación que permite determinar la posición y altura de las estrellas sobre el cielo. Le siguieron otros instrumentos como el cuadrante, la ballestilla o el sextante, todos con el mismo propósito. Algunas de las funciones del astrolabio son determinar la hora usando la latitud (y viceversa) y determinar distancias mediante un procedimiento de triangulación.

    - **Brújula**: Es un instrumento que permite determinar el norte y el sur magnéticos de la Tierra (basándose en los campos magnéticos).

Estos instrumentos junto a los mapas, cartas de navegación y conocimientos básicos en astronomía y trigonometría, permitieron al ser humano dar un avance clave en este ámbito, pero evolucionaría a un nivel más avanzado con las mejoras tecnológicas inminentes (soluciones más modernas e innovadoras).

Con la explosión de la tecnología, el posicionamiento y la navegación fueron evolucionando hasta alcanzar el estándar actual: el Sistema de Posicionamiento Global o GPS. A diferencia de los artefactos/métodos existentes, el GPS funciona mediante satélites distribuidos alrededor del planeta (figura 2.2). Además de esos satélites, se requieren receptores GPS encargados de recibir y decodificar la señal de los satélites. Estos se integran en diferentes dispositivos electrónicos como *smartphones.*

El GPS funciona mediante la aplicación del famoso *efecto Doppler* a las señales de radio (emitidas por estos satélites). Aplicando el *efecto Doppler* se estima la velocidad relativa del satélite (respecto a nosotros), que junto a nuestra posición nos permiten averiguar su posición relativa. Si supiéramos la posición relativa de los satélites y

aplicamos la misma lógica se podría obtener nuestra posición.

Naturalmente esta tecnología comenzó en algo sencillo, evolucionando hasta convertirse en lo que conocemos hoy en día. Hoy, cada satélite integra un ordenador y un reloj atómico, emitiendo constantemente su posición y hora actuales. Con la distribución satelital actual, es posible tener conexión con al menos cuatro satélites simultáneamente en cualquier lugar de la Tierra. Aunque estos sistemas pueden llegar a ser muy precisos, pueden experimentar errores debido a:

- El atravesamiento de las diferentes capas de la atmósfera, que afectan a las ondas emitidas y recibidas.
- La calidad del receptor de las ondas emitidas por los satélites.
- La naturaleza de las señales, que pueden rebotar en grandes obstáculos (sobre todo en ciudades).
- Pequeños errores del reloj atómico del satélite y/o de su trayectoria, que se reflejan en errores de gran magnitud.

A raíz de lo anterior (sobre todo el último punto), se plantean estaciones de control en la superficie terrestre con el objetivo de seguir al satélite y corregir su órbita y su reloj si fuera necesario. Esto se resume en el concepto de GPS diferencial (DGPS), que consiste en contar con una estación en Tierra con una posición exacta y conocida con el objetivo de reducir errores (figura 2.3). Con las señales de los satélites y la señal de la estación (una o varias), se pueden aplicar cálculos correctivos para obtener una posición más precisa. El único punto débil de este sistema es evitar estar demasiado alejado de la estación, ya que la distancia también implica errores adicionales.

Inicialmente tendríamos el sistema de satélites TRANSIT (9) formados por diez satélites, estadounidense, cómo no. Viendo que no es suficiente, se decide aumentar la rapidez, precisión y cobertura del sistema, dando lugar a una aplicabilidad global y continua. El resultado de todo este trabajo se llamó originalmente NAVSTAR GPS (10), destinada a propósitos militares. Más tarde comenzó a llamarse GPS y se permitió su uso a civiles.

Figura 2.2. Sistema satelital sobre la órbita terrestre.

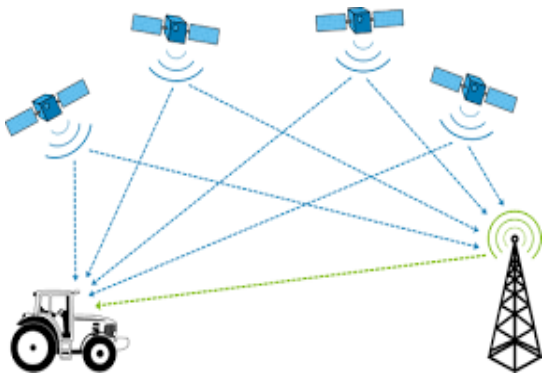


Figura 2.3. Funcionamiento del GPS diferencial (DGPS).

Similares a este sistema de satélites, cada entidad fue creando el suyo, por lo que tenemos GLONASS (11), perteneciente a Rusia, Beidou (12), perteneciente a China, y Galileo (13), perteneciente a la Unión Europea. Sin embargo, en mayor o menor medida, todos plantean los mismos principios.

A grandes rasgos, esto es todo lo que se puede comentar sobre el posicionamiento en exteriores y el GPS. Es un sistema que hoy en día cumple bastante bien su propósito, aunque por su naturaleza es evidente que en interiores no realice bien la labor de posicionamiento. Para ello, tenemos el siguiente apartado de este documento dedicado exclusivamente al posicionamiento en interiores.

2.2. Posicionamiento en interiores

Los avances realizados en el ámbito del posicionamiento en exteriores han demostrado su eficacia en entornos abiertos (navegación, turismo, rescate de personas desaparecidas…), incluso en entornos urbanos no demasiado densos, ya que comentamos con anterioridad que los edificios pueden afectar bastante a las señales.

Sin embargo, para proporcionar información detallada y guiada sobre nuestro paso por algún tipo de interior (sean museos, ayuntamientos, fábricas, centros médicos, metros…), esta tecnología es imprecisa y poco recomendable. En otras palabras, requerimos de sistemas de posicionamiento locales (en vez de globales) para calcular nuestra posición en estas circunstancias. Naturalmente, estos solo serían capaces de actuar dentro de una región limitada, concreta y exclusiva del planeta (14).

2.2.1. Contexto tecnológico

A diferencia del enfoque planteado en el posicionamiento en exteriores, para el posicionamiento en interiores las posiciones absolutas no cobran tanto protagonismo. En su lugar, se tienen en cuenta mecanismos basados en localizaciones relativas. Normalmente el objetivo no suele ser averiguar la latitud y longitud global del individuo (como sí se pretendía con el GPS), sino determinar su posición respecto a diferentes *puntos de referencia* de nuestro sistema (local y exclusivo). De este modo, aparecen conceptos interesantes como la creación de zonas y sectores específicos.

Actualmente, existe un amplio abanico de alternativas tecnológicas que podemos usar para posicionarnos en un entorno local concreto. Algunos ejemplos de los más conocidos pueden ser puntos de acceso Wifi, torres de telefonía o balizas Bluetooth, aunque tenemos también tecnologías ópticas, magnéticas y acústicas (15).

Independientemente de la tecnología, existe un concepto clave: la intensidad de las señales (naturaleza variada). Partiendo de este concepto, el posicionamiento del individuo usualmente se determina en función de la intensidad de señal recibida desde los diferentes puntos de control establecidos (origen de las señales transmitidas). Esto conforma una de las líneas más relevantes en el posicionamiento en interiores.

A raíz de este último concepto, surge otro similar: la fuerza o intensidad de la señal recibida o RSS (del inglés, *Received Signal Strength*). Este a su vez deriva en el

indicador de fuerza de la señal recibida o RSSI (del inglés, *Received Signal Strength Indicator*) (16) (17), que resulta ser una escala de referencia conocida y habitual en el ámbito del posicionamiento en interiores.

El posicionamiento en interiores está en avance constante y brinda suficientes tecnologías como para elaborar soluciones aceptadas y divergentes entre sí. A continuación, nos centraremos en describir las dos tecnologías más populares: Tecnologías Wifi y Bluetooth. Adicionalmente, se definirán algunas de las técnicas más conocidas y muchas de las utilidades potenciales de los IPS (aparte de la navegación). La mayor parte de la información que se presentará para estos apartados se ha extraído de varios artículos oficiales publicados por el IUCTC, perteneciente a la Universidad de Las Palmas de Gran Canaria, del cual forma parte el proyecto vinculado a este TFM. Entre esos artículos destacamos:

- Study on an Indoor Positioning System for Harsh Environments Based on Wi-Fi and Bluetooth Low Energy (18).
- Beacon-Related Parameters of Bluetooth Low Energy: Development of a Semi-Automatic System to Study Their Impact on Indoor Positioning Systems (19).
- A Survey on Bluetooth Low Energy Indoor Positioning Systems (20).

2.2.2. Tecnologías protagonistas

Las tecnologías Wifi (21) y Bluetooth (22) aplicadas al caso particular del posicionamiento en interiores dan lugar a dos alternativas bastante diferentes, que pueden ser vistas desde el punto de vista competitivo o complementario (pueden usarse ambas para contrarrestar los puntos débiles de la otra).

Puntos de acceso Wifi

Acorde con varios de los artículos revisados, esta tecnología suele utilizarse para estimar la localización del individuo con pocos requisitos de exactitud, es decir, para distinguir sobre qué gran área se encuentra un individuo, pero no para posicionarlo con exactitud sobre esa área. Las desventajas principales de esta tecnología son:

- La escasa precisión respecto a otras tecnologías similares. Por ello, se suele combinar con tecnologías Bluetooth para mejorar la precisión. No se descarta que esta tecnología evolucione aún más para lograr satisfacer esta *carencia.*
- Los costes que plantea un router Wifi, bastante altos y no comparables a otras alternativas.
- Las señales Wifi son bastante susceptibles a elementos del entorno como interferencias con otros dispositivos, personas, condiciones climatológicas adversas u obstáculos.

Sin embargo, no todos los aspectos son inconvenientes, algunas de sus ventajas son:

- Un mayor alcance de las señales. Esto permite cubrir más terreno con facilidad.

- Una mayor compatibilidad con la cotidianidad. La mayoría de las personas que se encuentran en un edificio con Wifi suelen tener activada esta característica en su dispositivo móvil.

Las bases de funcionamiento son bastante parecidas a la tecnología Bluetooth, por lo que se verán en conjunto en el siguiente apartado.

Puntos de acceso Bluetooth (Balizas Bluetooth)

A pesar de las tecnologías alternativas como el Wifi, el ZigBee, la luz visible, el campo magnético y otros tipos de señales, nos atrevemos a afirmar que el Bluetooth, concretamente el *Bluetooth Low Energy* (BLE), destaca sobre el resto.

La tecnología Bluetooth es una tecnología que ha evolucionado de una manera sorprendente a la vez que curiosa, mejorando cada vez más en versatilidad, rendimiento, eficiencia y seguridad. Esta evolución podría resumirse con el concepto de BLE (23).

Nos remontamos al Bluetooth 4.0, pues el BLE nació con objetivos destinados al *Internet de las cosas* (IoT) y como una extensión dentro de esta versión de Bluetooth. El BLE es usado para transmitir pocos datos en conexiones de aproximadamente 1 ms (a diferencia del Bluetooth clásico, donde se transmiten muchos datos en conexiones de 100 ms). Con ello, las señales BLE logran alcanzar mayores distancias, transmiten más información y mantienen conexiones seguras. Todo esto deriva en un menor consumo energético (sin descuidar las tasas de transferencia), que varía en función de algunos parámetros fundamentales:

- *Transmission Power (dBm)*. Representa la potencia de la señal transmitida y determina el alcance de la señal.
- *Advertising Interval (ms)*. Representa cada cuánto tiempo la baliza emite las señales.

Actualmente el BLE ha derivado en productos portadores de esta tecnología como balizas Bluetooth, *Smartwatches*, *Smartbands*… que presentan un consumo mínimo y amplían el mercado notablemente.

Con la llegada del Bluetooth 5.0 (y extensiones) los rangos, velocidades y capacidades de transmisión se incrementan notablemente (por cuatro, por dos y por ocho, respectivamente). Además, estas versiones introducen poco a poco el cálculo de distancias y orientaciones de otros dispositivos Bluetooth, lo cual podría significar un aporte interesante a este sector.

Entrando un poco más en detalles técnicos, los paquetes que se mandan en la transmisión siempre dependen del protocolo o estándar que se utilice (iBeacon de Apple, Eddystone…). Al margen de eso, una característica de funcionamiento común es el *modo publicación* (en inglés, *advertising mode*), que consiste en un estado en el que la baliza envía mensajes constantemente a posibles receptores a través de los tres canales primarios de transmisión para ganar redundancia y fiabilidad.

Es turno de pasar al planteamiento de ventajas y desventajas de la tecnología

Bluetooth respecto al resto. Aunque puede plantear ciertas ventajas diferenciales respecto al resto (mejor precisión y susceptibilidad), no debemos olvidar que asienta sus bases en señales de radiofrecuencia y que está expuesta a una degradación y fluctuación condicionada por el entorno (solapamiento de señales, variaciones…). La tecnología Bluetooth en su estado actual aporta los siguientes beneficios:

- Una mayor precisión, siempre teniendo en cuenta el concepto RSSI comentado con anterioridad.
- Es más barata. Estas balizas Bluetooth plantean un coste reducido respecto a otras alternativas como los puntos de acceso Wifi. Esto permite preparar un entorno rico en balizas y más eficaz.
- Menos susceptible a elementos del entorno. Aunque no es inmune, presenta menos inconvenientes que las señales Wifi. Además, no plantea complejidad en la lectura de la fuerza de la señal.

En definitiva, con la tecnología BLE, se obtienen soluciones portables (funcionan por batería), pequeñas, ligeras, eficientes (bajo consumo) y altamente precisas. Hasta ahora solo se han expuesto ventajas, pero nada más lejos de la realidad, esta tecnología también presenta sus flaquezas:

- Menor alcance. Aunque esta carencia se compensa con su reducido precio, es una realidad que plantea un menor alcance que otras alternativas.
- Menos cotidiano que otras tecnologías en nuestros dispositivos móviles. Puede parecer algo anecdótico, pero si no disponemos de un dispositivo que *nos obligue* a tener activo el Bluetooth, es una característica que solemos tener desactivada (al contrario que con la tecnología Wifi). Por suerte, cada vez incorporamos más dispositivos Bluetooth a nuestro día a día.

Es cierto que algunas tecnologías como *Ultra Wide Band* (UWB) (24) pueden llegar a ofrecer precisión con un error de centímetros, pero con la tremebunda evolución del Bluetooth y su amplia aceptación en una amplia gama de dispositivos, queda expuesto que es una alternativa muy atractiva. De hecho, en este ámbito, la tecnología Bluetooth no solo plantea servicios basados en localización y proximidad, sino que ofrece funciones y servicios avanzados a los usuarios que se expondrán en el siguiente apartado *Aplicaciones complementarias del Bluetooth*.

Aplicaciones complementarias del Bluetooth

A continuación, se describirán algunas aplicaciones adicionales o complementarias del BLE, avistadas también en contenido bibliográfico ya expuesto anteriormente (14) (15).

- Creación de rutas y navegación. Esto es evidente en exteriores, pero en interiores es también una necesidad.
- Alertas al acercarnos a algunas de las balizas BLE. Esto es útil para aportar información turística, orientación en lugares nuevos, ofertas de tiendas cercanas, entrada de visitantes no deseados, entrada en zona de riesgo, llegada de

paquetes… Un sector muy beneficiado de esto sería por ejemplo el transporte público (las estaciones), como se ilustra en los artículos del IUCTC mencionados al final de la contextualización y en la figura 2.4 extraída de ellos.

- Saber dónde está ubicado un paciente, la concentración de gente en cada parte del edificio, donde está un personal de alto cargo…
- Cálculos de distancias para operar correctamente (quién está más cerca para operar con mayor rapidez…).
- Grandes aportaciones de utilidad que pueden relacionarse con el Big Data. Con la información que obtenemos de multitud de dispositivos personales, podemos detectar patrones de utilidad en multitud de sentidos.

Esto es solo una señal más de que esta tecnología es probablemente una de las mejores opciones, aunque no se desprestigia al resto de posibilidades.

2.2.3. Métodos de posicionamiento

Aunque pueden surgir nuevos métodos, se describirán algunos de los más conocidos a continuación y se ampliaran detalles sobre el que, en principio, es más relevante.

- **Proximidad**. Se calcula la posición en base a las coordenadas de la baliza más cercana, aunque no es del todo eficaz y depende de la línea visual con las balizas.
- **Trilateración** (25) **y Triangulación** (26). Podemos apreciar una representación gráfica de ambos métodos en las figuras 2.5 y 2.6. Ambos métodos son parecidos. Básicamente utilizan distancias (multilateración) o ángulos (multiangulación), respectivamente, para estimar la posición del objetivo. Cuando se usan distancias entre las balizas y el objetivo, se usan diferentes técnicas de estimación (Tiempo de vuelo de la señal o TOF, Diferencias de tiempos de llegada o TDA…), mientras que cuando se usan ángulos, se utiliza el ángulo de llegada de las señales para estimar la localización del objetivo.

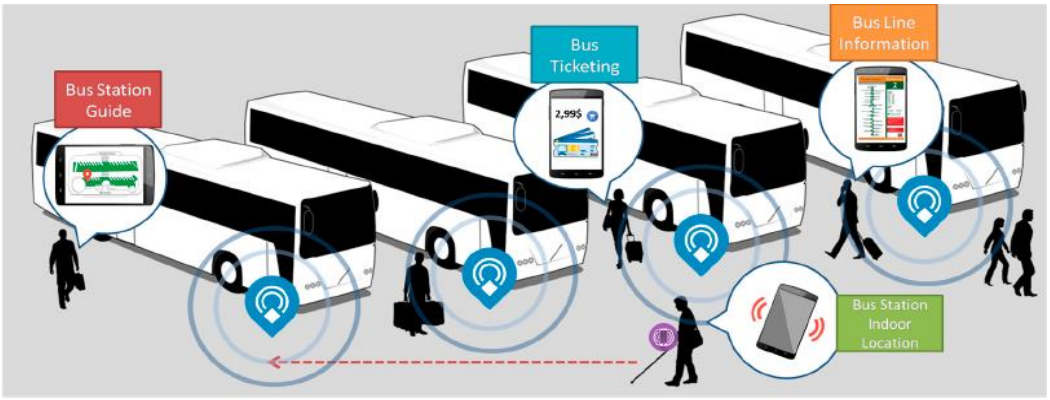


Figura 2.4. Ejemplo de servicios adicionales de las balizas Bluetooth. Fuente (18).

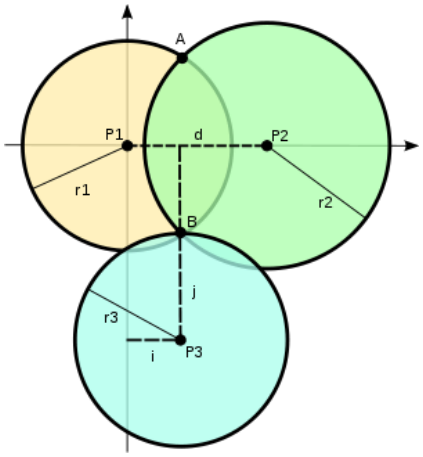


Figura 2.5. Representación del concepto de Trilateración.

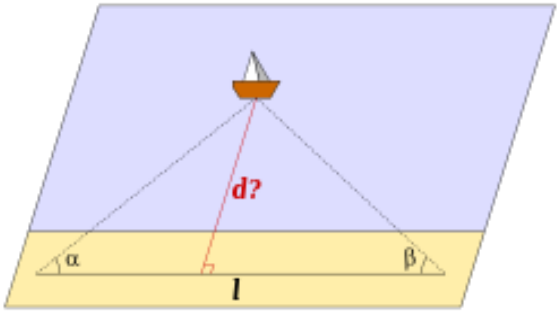


Figura 2.6. Representación del concepto de Triangulación.

- **Centroides**. Utiliza el centroide de figuras geométricas planas para calcular la posición. No nos detendremos en detalles en lo que a esta opción respecta.
- **RSSI** (16). Ya lo comentamos en párrafos anteriores y es una medida relativa de la fuerza/intensidad de la señal recibida (RSS) que, en combinación con un modelo de propagación de pérdida de ruta, permite calcular la distancia entre la baliza y el receptor.
- ***Fingerprinting* o análisis de escena**. Consiste en recolectar las señales recibidas de las balizas (RSSI) y asociar los valores con una posición particular (de ahí el nombre de *Fingerprinting*).
- **Híbrido**. Esta técnica se basa en la combinación de dos o más de los anteriores.

Fingerprinting

Este es uno de los métodos más interesantes, puesto que hace uso de la intensidad de las señales recibidas (RSSI) para identificar cada punto (que presenta valores relativamente únicos). Este nombre viene dado porque existe un claro *simil* con nuestras huellas dactilares, únicas para cada ser humano. Este tipo de métodos tiene la peculiaridad de requerir muestreos previos para poder obtener la posición actual. El concepto de su funcionamiento es sencillo:

1. Se realiza una captura de datos en los diferentes puntos para construir una base de datos que poder consultar. Esta base de datos contendrá los datos sobre las intensidades de las señales recibidas en cada punto, orientación y otros parámetros requeridos. Esta fase podemos definirla como la *fase de calibración/entrenamiento o fase offline*.
2. Se obtiene la posición actual contrastando los valores actuales con nuestra base de datos. Naturalmente, la exactitud completa significaría obtener valores en todos los puntos posibles, lo cual a veces no es factible. En estos casos, el punto

actual es obtenido en base al valor que más se le aproxime en la base de datos. Esta fase podemos definirla como *fase de posicionamiento o fase online*.

Puede quizás ser obvio, pero es necesario alcanzar un balance o equilibrio en la base de datos. Posibles situaciones son las siguientes:

- Se capturan datos sobre la mayoría de los puntos posibles. Esto puede favorecer el funcionamiento, pero también implica un sobreesfuerzo en la elaboración de la base de datos.
- Se captura el mínimo número de puntos posibles para su funcionamiento. A pesar de que reduce el esfuerzo necesario para elaborar la base de datos, se podrían manifestar carencias en el funcionamiento del sistema, aunque es cierto que se puede mejorar mediante interpolaciones.
- Equilibrio entre esfuerzo y datos en la base de datos. Una alternativa interesante es la construcción de la base de datos a medida que esta es utilizada, es decir, una especie de fusión entre las *fases offline y online.*

Dentro de este concepto, existen al mismo tiempo diferentes técnicas o métodos de comparación:

- Deterministas. El algoritmo más común es el NN (*Nearest Neighbour*) o kNN (*k-Nearest Neighbours*) (27). Atendiendo a los valores de la base de datos, el resultado será la posición con mayor coincidencia en términos de RSSI (y otros posibles factores). En el caso del kNN, la única diferencia es que no se centra en la mayor coincidencia sino en un conjunto de mayores coincidencias, realizando la media oportuna para obtener el resultado. El proceso puede ir acompañado de algoritmos de comparación de patrones para optimizar el resultado.
- Probabilísticos. Se basan en la inferencia Bayesiana (estadística) (28). La fuerza de las señales se representa como una distribución probabilística de forma que el resultado sea la posición.
- Basadas en *Machine Learning* (29). Aquí destacan las redes neuronales y las máquinas de soporte vectorial, aunque no nos detendremos en detalles.

En general, las técnicas basadas en *Fingerprinting* plantean tanto ventajas como desventajas. Centrándonos en sus ventajas, tenemos que es fácil de implementar usando infraestructuras existentes sin necesidad de introducir nuevo hardware, todo a un bajo coste. A esto se suman la baja complejidad computacional y la razonable exactitud y precisión que ofrecen.

Al mismo tiempo, plantea desventajas como el elevado tiempo empleado para construir el mapa (base de datos) o las limitaciones al rediseñar el lugar (inmobiliario) o el sistema (hardware), pues da lugar a la necesidad de actualizaciones en la base de datos. También expone la necesidad de muchos parámetros del entorno, es decir, que necesitamos la mayor cantidad de información posible para que nuestra base de datos sea fiable.

3. Herramientas utilizadas

Este capítulo trata de conglomerar todas y cada una de las herramientas partícipes en el trabajo realizado en este TFM.

3.1. Fritzing

Exclusivamente en la fase de diseño de la estación meteorológica y por el mero hecho de asegurar que todo estuviera correcto, se usó la herramienta Fritzing (30). Esta es una iniciativa de código abierto que permite, entre otras cosas, hacer la electrónica algo más flexible y educativa. Ofrece un ecosistema creativo donde, a través del software, las personas pueden elaborar, documentar y compartir sus propios prototipos.

Una de las características más interesantes de esta herramienta es la posibilidad de esquematizar en diferentes formatos de salida de forma rápida y automatizada. Los dos aspectos esquemáticos más útiles y conocidos son los mostrados tanto en la figura 3.1 (visión más técnica) como en la figura 3.2 (visión más fiel a la realidad). En nuestro caso, gracias a esta herramienta pudimos montar un esquema digital del circuito que se quería diseñar/implementar (de acuerdo con las variables meteorológicas y el material seleccionados). Algunos beneficios de haber usado esta herramienta son:

- Evitar trabajo de montaje innecesario en fases iniciales del diseño.
- Comprender mejor las especificaciones del material disponible para el montaje. Aunque se estudiaron las características de los sensores adquiridos (analógico o digital, precisión…), a la hora del montaje era necesario conocer mejor algunas particularidades sobre ellos, como:
    - Qué entrada/salida representaba cada PIN del sensor de cara a minimizar errores.

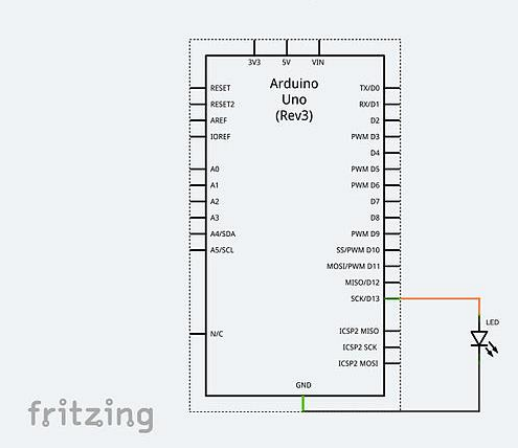


Figura 3.1. Diseño del programa Fritzing consistente en una nomenclatura formal.

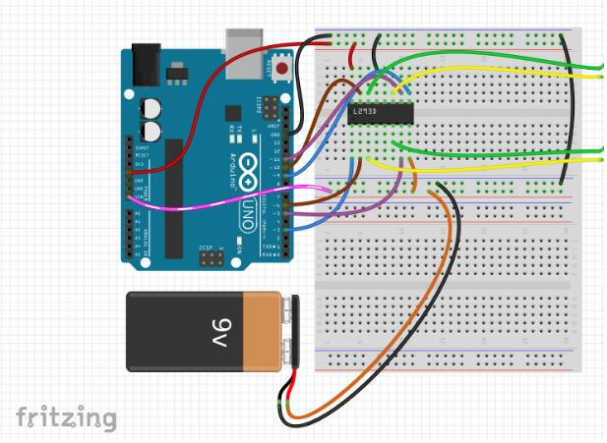


Figura 3.2. Diseño del programa Fritzing consistente en una nomenclatura más visual y gráfica.

- Necesidad (o no) de usar componentes adicionales para el montaje, como, por ejemplo, resistencias. Gracias a las resistencias incorporadas en los circuitos de los sensores usados, esto no fue necesario.

- Proporcionar documentación sobre el diseño final del circuito para su incorporación en este documento.

3.2. Arduino IDE

Debido a que Arduino está presente en nuestro módulo meteorológico, no es extraño que su entorno de desarrollo integrado o IDE (*Integrated Development Environment*) esté presente en el conjunto de herramientas usadas.

Arduino IDE (31) (32) es una aplicación multiplataforma (Windows, Mac OS y Linux) que ofrece un entorno para escribir y subir programas desarrollados a las placas Arduino a través del puerto serie. Podemos ver un ejemplo de una placa Arduino Uno (como la usada en nuestro proyecto) en la figura 3.3.

Los lenguajes base de este IDE son C y C++. De hecho, varias librerías desarrolladas para este entorno están escritas en C++. A la hora de desarrollar existen dos rutinas principales que vienen creadas por defecto en proyectos nuevos: `setup()` y `loop()`, mostradas en la figura 3.4. La primera se encarga de establecer la configuración de inicialización del programa, mientras que la segunda se encarga de ejecutar de forma repetida el código que contiene. Arduino IDE admite, como es lógico, bastantes procedimientos de E/S (por ejemplo, para establecer si un PIN es de entrada o de salida, para leer o escribir datos de/en ellos…). Es por ello por lo que fue una herramienta bastante útil en el desarrollo. Algunas de sus funciones fueron:

- Prueba o *testeo* de los sensores adquiridos mediante el uso de programas sencillos y librerías, revisando las salidas obtenidas desde el *monitor serial*.

Figura 3.3. Diseño físico de la placa Arduino UNO.

Figura 3.4. Estructura clásica de un programa elaborado en Arduino IDE.

- Carga de programas en la placa Arduino para dejarla lista para la comunicación con nuestro programa principal (elaborado en Python), empleando un protocolo sencillo y personalizado.

Arduino IDE nos facilitó bastante el trabajo por el mero hecho de que nos permitió establecer las bases sobre las que construir nuestro módulo meteorológico. No solo es de libre acceso, sino que cuenta con múltiples librerías (oficiales o de terceros). En nuestro caso, se utilizaron básicamente las librerías desarrolladas para los sensores adquiridos y otras librerías utilizadas para codificar y decodificar en formato JSON.

3.3. JetBrains PyCharm

El IDE por excelencia empleado para el desarrollo de los dos módulos (estación meteorológica y módulo orientador) ha sido JetBrains PyCharm Professional Edition (33). JetBrains (34) ofrece una amplia gama de herramientas de bastante calidad para programar en diferentes lenguajes y, sin duda, PyCharm no es una excepción.

JetBrains establece algunas limitaciones en la mayoría de sus productos. Normalmente ofrece una versión *Community* (o gratuita) y otra versión *Professional* (o de pago). Para esta última, brinda la posibilidad de probarlo durante 30 días de manera gratuita, debiendo abonar dinero tras ese periodo. En nuestro caso, usamos la versión *Professional* de PyCharm en calidad de estudiante, pues ofrecen beneficios a estudiantes de algunas instituciones educativas como la ULPGC. En la figura 3.5 mostramos gráficamente las diferencias oficiales entre la versión gratuita y la de pago.

A continuación, nombraremos algunas de las características favorables de este entorno de desarrollo, extraídas de la experiencia propia y de fuentes oficiales:

- Potencia la calidad del código. Este IDE aporta avisos y correcciones que facilitan refactorizar y optimizar el código cómodamente. Esto mejora el rendimiento y minimiza los fallos al escribir código. Aunque es una función típica y casi obligatoria, este IDE cumple las expectativas.
- Posibilidad de visualizar la jerarquía de directorios y archivos de un proyecto cómodamente. Adicionalmente, es compatible con el control de versiones usado (*git*) y colorea los elementos según su estado actual en el repositorio.
- Ofrece una consola para la ejecución eficaz de varios archivos Python simultáneamente, distribuidos en una disposición de pestañas (la parte inferior).
- Dispone de un intérprete de código en consola. Esta característica es lógica en lenguajes como Python. Otra característica interesante es la opción de elegir un intérprete de Python existente o crear un entorno virtual (*virtual environment*).
- Además de esto, presenta multitud de funcionalidades adicionales (no usadas en nuestro caso): soporte *Framework* para desarrollos web modernos, herramientas científicas (*matplotlib*…), múltiples utilidades orientadas a la integración óptima de bases de datos…

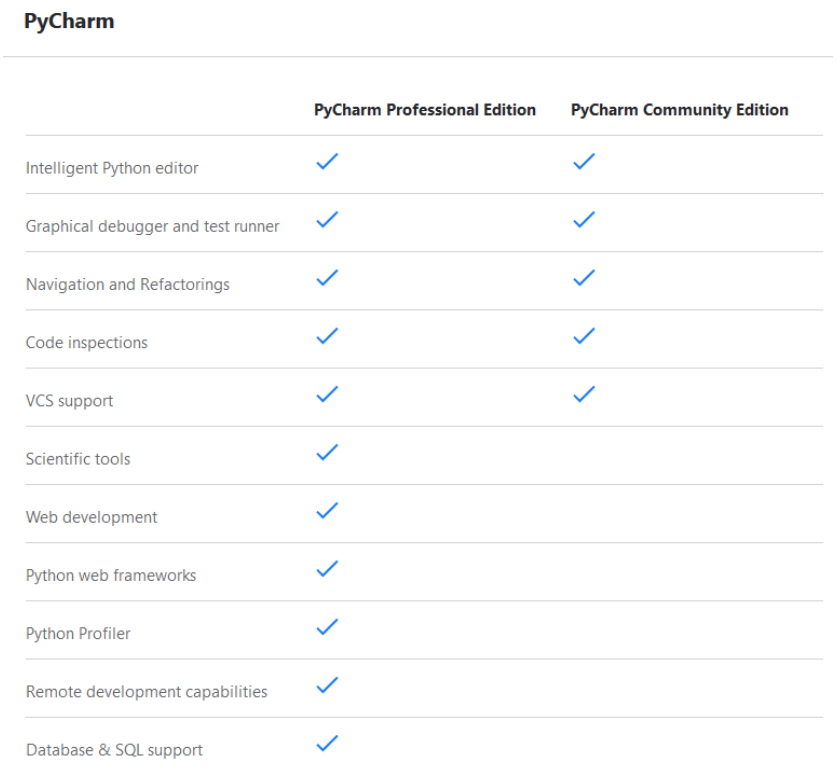


Figura 3.5. Características PyCharm *Community* and *Professional*.

3.4. Git

Un proyecto sin control de versiones hoy en día es prácticamente impensable. Es por eso por lo que empleamos Git (35) en nuestro proyecto. Las razones de su uso son:

- Es uno de los mecanismos de control de versiones más usados y populares.
- Se puede utilizar para cualquier tipo de proyecto ubicado bajo un directorio concreto (no necesariamente un producto software).
- Ha sido el que se ha estudiado y usado a lo largo de toda la carrera universitaria, por lo que es lógico querer reciclar los conocimientos aprendidos.

Un control de versiones consiste en la gestión exhaustiva del progreso realizado en un conjunto de archivos, asegurando los cambios y permitiendo entre otras cosas restaurar versiones pasadas de dichos archivos. En definitiva, aporta mucha flexibilidad a la hora de manejar cualquier proyecto.

Al descargar Git, disponemos de una herramienta para ejecutar comandos de Git desde una consola especializada (*Git Bash*). No obstante, podemos usar también programas con una interfaz gráfica (como el *Sourcetree*) tanto para ver el estado del repositorio como para ejecutar los comandos requeridos, todo de forma más visual.

Para finalizar con este apartado, vamos a describir algunos conceptos básicos de Git y las operaciones básicas que se han utilizado (las más importantes):

- Git consiste en *instantáneas* de un conjunto de archivos dentro de nuestro proyecto (directorio de trabajo). Cada *instantánea* no es más que un estado particular del proyecto en un punto concreto del tiempo.
- La sucesión de *instantáneas* que se va generando se puede entender gráficamente como una sucesión de nodos. Cada nodo tiene un identificador único y contiene un conjunto de cambios realizados (*commit*).
- Cada secuencia o línea de nodos se considera una rama (*branch*). La rama principal generada al principio del proyecto se denomina *máster.* Los nodos que se van generando pueden formar parte de esta rama *máster* o de otra rama secundaria que bifurca de *máster*. Estas ramas secundarias son bastante útiles en proyectos colaborativos y se crean para realizar cambios muy concretos en nuestro proyecto (como añadir una funcionalidad concreta en el software) y normalmente vuelven a fusionarse (*merge*) con la rama *máster*.
- El control de versiones puede realizarse exclusivamente en local, aunque lo recomendable es disponer de un repositorio remoto en el que los cambios son subidos (para evitar pérdidas en caso de avería del equipo, por ejemplo). La operación de subida de cambios al repositorio remoto se denomina *push* mientras que la operación inversa se denomina *pull*.
- Las operaciones básicas en Git son, a grandes rasgos, las siguientes:
    - *Add*: Añade un conjunto de archivos para que formen parte de la siguiente *instantánea*.
    - *Commit*: Crea un nuevo nodo almacenando todos los cambios realizados sobre un conjunto concreto de archivos.
    - *Push*: Actualiza el repositorio remoto con todas las *instantáneas* nuevas.
    - *Pull*: Actualiza el repositorio local con todas las *instantáneas* nuevas.
    - *Branch*: Crea una nueva rama que bifurca de la rama actual (normalmente, *máster*).
    - *Merge*: Fusiona una rama con otra (ya comentado anteriormente).
    - *Checkout*: Volver a versiones (*commits*) anteriores de nuestro proyecto.

3.5. Sourcetree

Previamente hablamos del uso de Git en nuestro proyecto. SourceTree (36) fue la herramienta complementaria que se usó para gestionar el control de versiones desde una perspectiva más gráfica y visual.

Particularmente se utilizaba la mayor parte del tiempo la línea de comandos debido a la baja complejidad del repositorio y de las operaciones realizadas. No obstante, el SourceTree se usó sobre todo con propósitos orientados a ver el estado del

repositorio de una forma más práctica y cómoda. Por tanto, una de las operaciones más cómodas en este entorno gráfico (respecto a la línea de comandos) es *checkout*.

Con esto y lo comentado en el apartado anterior de Git, tenemos lo básico para entender a nivel general el control de versiones. Dejamos varias capturas finales con algunas partes del SourceTree, tales como algunos nodos de la secuencia de *commits* (representación) y el detalle de un *commit* particular (figuras 3.6 y 3.7, respectivamente).

3.6. Bitbucket

Con el control de versiones establecido, Git en nuestro caso, podemos crear diferentes puntos de guardado de nuestros cambios, permitiéndonos tanto salvaguardar los últimos cambios como restaurar versiones antiguas del proyecto.

Ya se comentó ligeramente en los anteriores apartados relacionados, pero lo remarcamos nuevamente. Lo que es obligatorio es tener el repositorio con el control de versiones en local, pero… ¿y si nuestro equipo sufre algún daño y lo perdemos todo? Lo lógico sería no tener los cambios únicamente en local, sino tenerlos remotamente almacenados con herramientas como Github o Bitbucket. En nuestro caso disponemos de dos repositorios remotos, uno por cada módulo desarrollado.

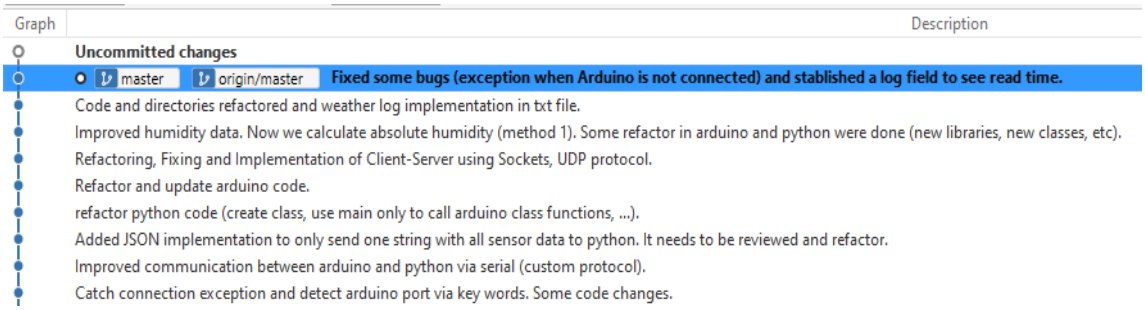


Figura 3.6. Nodos dentro del programa Sourcetree.

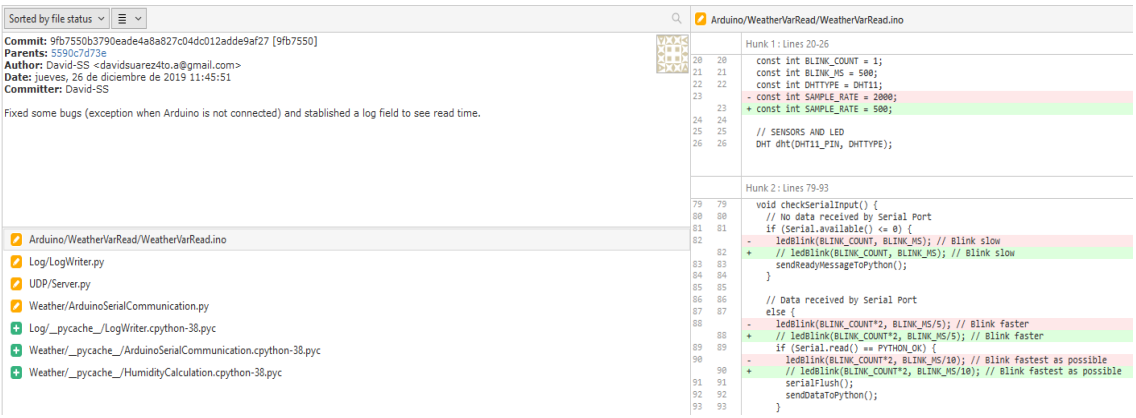


Figura 3.7. Nodos dentro del programa Sourcetree.

En este proyecto se opta por Bitbucket (37), al igual que los últimos proyectos realizados. Bitbucket es una de las muchas herramientas disponibles para el control de versiones de un producto software mediante el uso de repositorios en Git (o Mercurial) de forma remota. Aunque Github es también bastante popular, se descartó porque a nivel personal existen preferencias por Bitbucket.

Ambas herramientas comentadas ofrecen la posibilidad de mantener un repositorio colaborativo remoto (fuera del entorno local), con todas las características que ya se han repasado sobre Git. Para nuestro proyecto, el único colaborador soy yo, es decir, no existen más desarrolladores asignados a este proyecto/repositorio.

La mejor manera de aportar contenido a un repositorio consiste habitualmente en la creación de ramas que bifurquen de *máster* para crear un nuevo conjunto de aportaciones. Al acabar el desarrollo en esa rama, se fusiona a la rama máster y desaparece. A pesar de esto, en nuestro caso no existe la verdadera necesidad de crear ramas, por lo que se trabaja directamente sobre la rama *máster* para realizar cambios.

Cuando se dé el caso de que existe más de un desarrollador, el uso de ramas cada vez que uno va a realizar cambios se vuelve prácticamente indispensable. Todos trabajarían sobre la última modificación de la rama máster y al terminar su aportación, fusionan su rama a la rama *máster* para añadir las nuevas modificaciones. De esta manera, se pueden manejar conflictos (modificaciones de un mismo archivo) de forma consistente para que el proyecto crezca de forma correcta.

Como cada producto en el mercado, Bitbucket presenta tanto ventajas como desventajas frente a su competencia (en su caso, GitHub principalmente). En la parte de su Web asociada a software, concretamente la asociada a Bitbucket, comenta una serie de ventajas del producto frente a GitHub. La mayoría son interesantes y hacen pensar al lector que la herramienta que debe usar es Bitbucket. Algunas de las características que decantan la balanza a favor de Bitbucket son:

- Repositorios privados gratuitos e ilimitados. Esta siempre será una característica destacable, porque puede existir interés en ocultar un repositorio hasta que el producto esté acabado.
- Certificación *Soc 2 tipo II*: Básicamente, garantiza que el código en la nube no se perdería ni sería accedido de manera no deseada. Bitbucket ha sido una de las primeras soluciones en garantizar esto, según la información revisada.

3.7. Microsoft Office y FreeMind

Para abordar todas las labores de documentación a lo largo de este TFM, incluyendo la elaboración misma de este documento, se han tenido que usar diferentes herramientas. En este apartado comentaremos todas esas herramientas.

En primer lugar, tenemos el conjunto de programas de Microsoft (Office 365) (38), principalmente Microsoft Word para la elaboración de documentos de texto y Microsoft PowerPoint para la elaboración de la presentación de este TFM. En lo que a estas herramientas respecta, tampoco conviene comentar demasiados detalles más,

pues son simples y conocidas, de forma que sería incluso redundante seguir comentándolas.

Abandonando el conjunto de herramientas de Microsoft, en la elaboración de documentos de textos también se ha utilizado una herramienta llamada FreeMind (39). Incluimos esta herramienta en el mismo apartado que las herramientas de Microsoft por su alta vinculación al uso de Microsoft Word para generar documentación. Se muestra una captura de pantalla de esta aplicación en la figura 3.8.

Esta herramienta se ha usado para elaborar un mapa conceptual de las partes que forman esta memoria. Tiene características muy útiles como la vinculación de archivos a cada nodo del esquema. Con esto, se ha logrado modificar de forma aislada cada apartado de este documento (fase previa a la integración final en un solo documento).

También contamos con características menores pero que igualmente ayudan a la organización y construcción de la memoria final, como puede ser la capacidad de formatear cada nodo con diferentes colores y fuentes, añadir anotaciones para cada nodo, añadir marcas o *pegatinas* para simbolizar el estado de un nodo, entre otros.

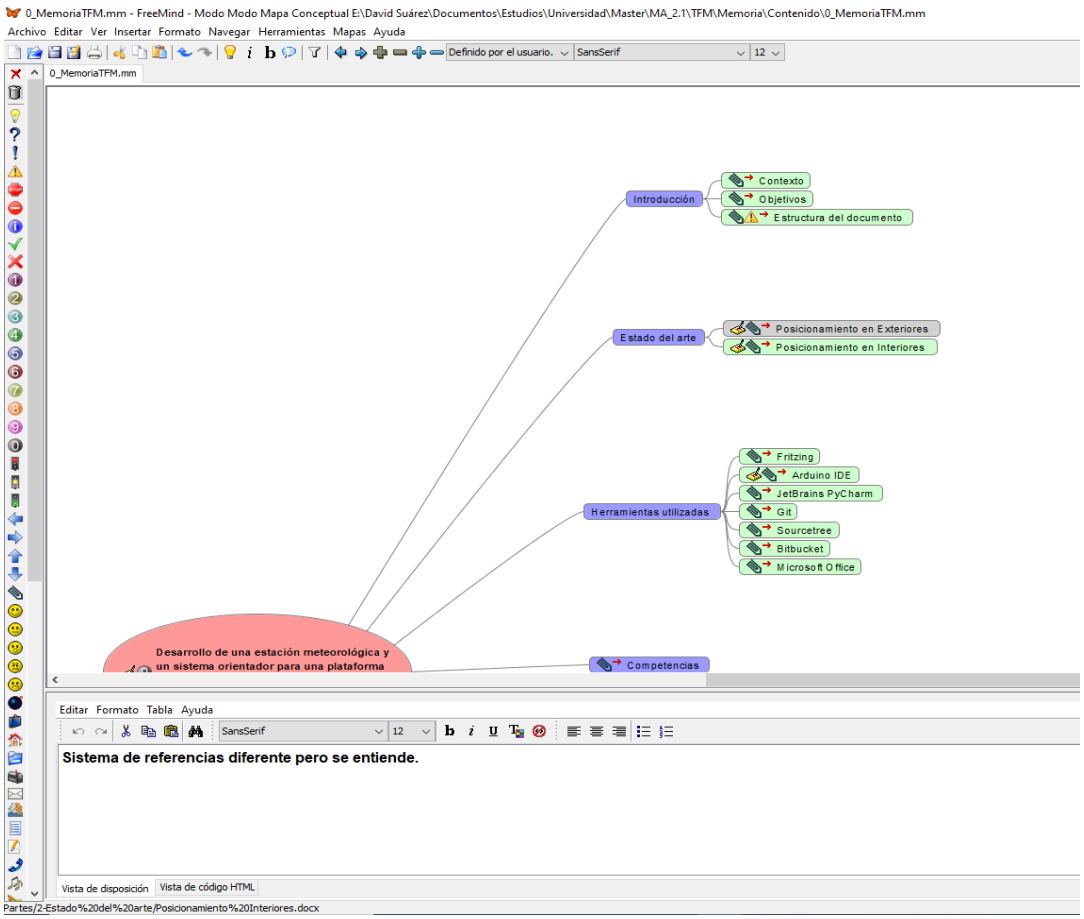


Figura 3.8. Captura del programa FreeMind durante el desarrollo de la memoria.

3.8. R+ Manager 2.0

En el caso particular del orientador, se ha empleado un motor Dynamixel para ejecutar las labores de rotación esenciales para este módulo (permiten adoptar diferentes posiciones en grados). Para afrontar el primer contacto con este motor (configuración, experimentación…) se ha empleado una herramienta oficial proporcionada por los fabricantes: R+ Manager 2.0 (40).

Esta aplicación es básicamente una interfaz que nos permite configurar y controlar a voluntad el motor Dynamixel conectado a nuestro ordenador. Con esto se han podido configurar algunos de sus valores de configuración (principalmente el ID) y experimentar las claves de su funcionamiento sin recurrir a la programación.

R+ Manager 2.0 ha sido clave principalmente en la primera toma de contacto con el motor, ya que experimentamos de primera mano las operaciones que había que realizar sobre el mismo para poder habilitarlo y realizar rotaciones, todo de manera bastante gráfica. Todos los conceptos asimilados de forma gráfica podrían luego ser trasladados a la programación.

Con el objetivo de reflejar el aspecto e idealizar los comportamientos o funciones descritas anteriormente, se muestra la pantalla principal de control del motor (modelo especifico escogido para nuestro proyecto) en la figura 3.9.

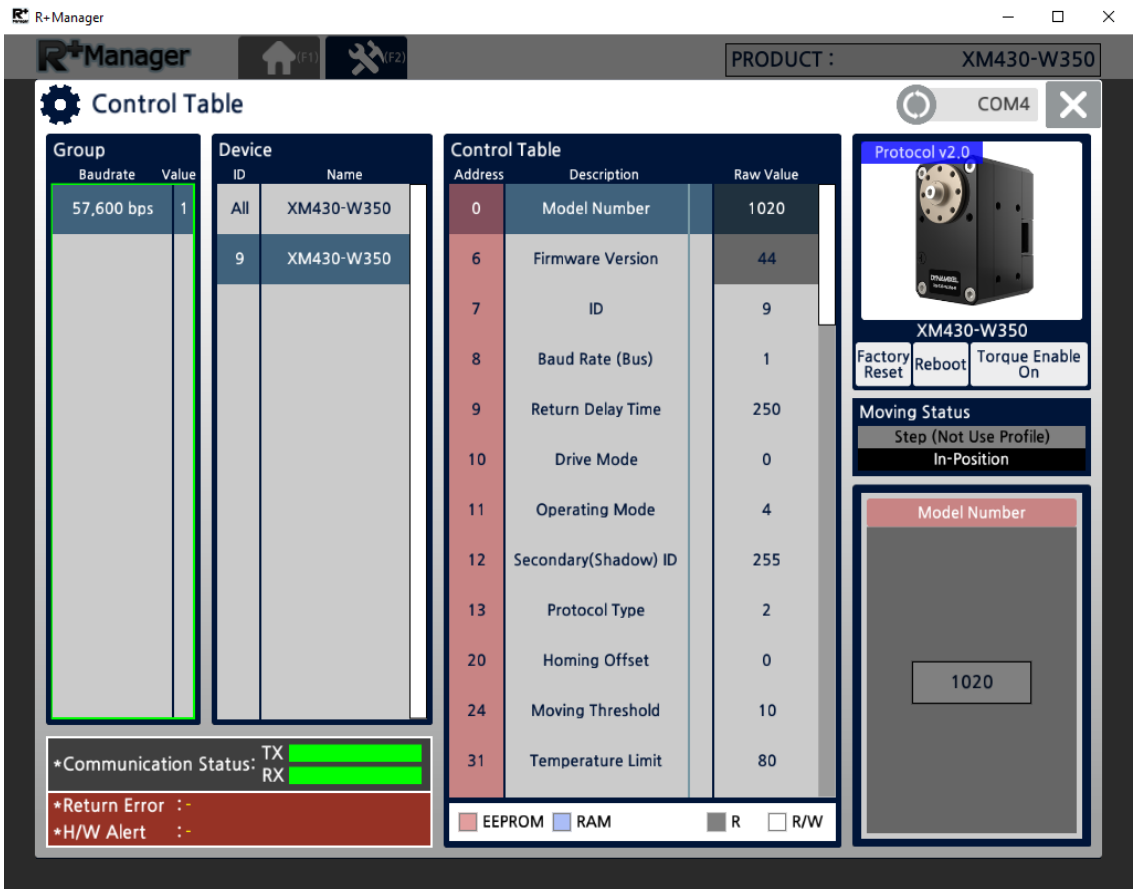


Figura 3.9. Pantalla principal de visualización y control del motor.

4. Normativa y legislación

En este anexo se tratarán diferentes aspectos legales relacionados en mayor o menor medida con nuestro TFM. Si bien es cierto que algunas partes de este apartado no tienen demasiada aplicación en nuestro TFM actual, se comentan por su gran relevancia en el campo de la Informática y la posibilidad de tener que afrontarse en posibles derivaciones de este proyecto. Adelantamos que este es el caso de la protección de datos, que comentaremos más detalladamente en su momento.

En este apartado se comentarán dos aspectos legales de vital importancia. Estos son las licencias y las leyes.

4.1. Licencias

Empezaremos por definir un poco el concepto de licencia en el ámbito del software. Una licencia de software es la que especifica los términos y condiciones de uso del software mediante un contrato entre el licenciatario y el licenciante/autor/titular de los derechos de distribución y explotación. Con esto en consideración, pasamos a comentar y explicar todas las licencias relacionadas con las herramientas utilizadas.

Licencia GNU (GPL o LGPL)

Esta licencia debe sus siglas a Licencia Pública General (General Public License) de GNU (GNU's Not Unix) y es una licencia del tipo *copyleft*. Esto último implica la total libertad para explotar dicho software en términos de uso, copia y modificación. Cualquier software que haga uso de otro con licencia GNU GPL está obligado a ceñirse al mismo tipo de licencia (mismas condiciones). Esto evita intentos de apropiación.

Uno de los objetivos que pretende esta licencia es la evolución de un producto de manera colaborativa y abierta al público. De un mismo producto y fruto de esta colaboración, pueden surgir diferentes mejoras y ramas de un mismo producto.

En nuestro caso, el producto más importante afectado por esta licencia es Arduino, tanto el IDE (código fuente) como la placa. Los productos de la compañía son distribuidos como Hardware y Software libre bajo esta licencia, tanto en su forma habitual como en su forma reducida (LGPL). El objetivo de Arduino no es ni más ni menos que el comentado en el párrafo anterior, que se resume en *progreso*.

La popular herramienta Git también hace uso de la licencia GNU GPL, en su segunda versión. En base a la información recopilada, el mantenimiento de este software está supervisado por Junio Hamano (que cuenta con el soporte de 280 programadores contribuyentes, aproximadamente). Esto puede cambiar con el tiempo, pero queda claro tanto el tipo de licencia como la gran supervisión con la Git cuenta.

Asimismo, el código fuente de la herramienta de diseño Fritzing está también bajo este tipo de licencia, la GNU GPL en su tercera versión. A pesar de esto, su documentación y sus gráficos de la vista de la placa están bajo otro tipo de licencia, CC Attribution-ShareALike (CC-By-SA).

De esta manera, puedes republicar la documentación de Fritzing mientras contribuyas en la misma y la publiques bajo esta misma licencia. Sin embargo, Fritzing sí que establece limitaciones en cuanto al uso de sus gráficos en otros sistemas software, aunque ofrecen la posibilidad de conceder excepciones si se contacta con ellos previamente y se llega a algún tipo de acuerdo.

Otro de los programas que entra bajo esta licencia GPL es FreeMind, la herramienta utilizada para elaborar mapas conceptuales. Probablemente esta herramienta sea tan simple y útil gracias a que es de código abierto.

Otras licencias

Para el resto de las herramientas faltantes, el tipo de licencia es más específico y particular.

A parte de Arduino, se trabajó con el lenguaje de programación Python, el cual posee la licencia Python Software Foundation License (PSFL), que es una licencia libre permisiva y compatible con GPL. El ser libre permisiva implica que es flexible respecto a la distribución (puede ser redistribuida como libre o privativa). De hecho, la característica más relevante de esta licencia es la ausencia de *copyleft*, evitando mantener el mismo régimen de derechos de autor que el original. Además, esta licencia está dentro de las listas de licencias aprobadas por la Free Software Foundation (FSF) y por la Open Source Initiative (OSI).

Las licencias de este tipo (libres permisivas) más conocidas son la Licencia MIT y la licencia BSD, pero ninguna de las herramientas usadas cuenta con ellas, por lo que la PSFL es probablemente la más parecida de las licencias en nuestro caso.

En otras herramientas utilizadas como Sourcetree, nos encontramos ante una licencia de software propietario, es decir, el código fuente es totalmente privado y cerrado al público, aunque desde la adquisición de Sourcetree por parte de Atlassian el uso de esta herramienta es totalmente libre y gratuito, siendo el único requisito registrarse en su página oficial para obtener un identificador de acceso a las aplicaciones de Atlassian (incluida esta).

Al contrario que la anterior, Bitbucket (perteneciente a Atlassian), contiene una licencia similar pero no es totalmente gratuita, sino que es shareware. Esto supone un uso gratuito de la aplicación bajo unas limitaciones concretas (como el número de colaboradores en un repositorio privado) a no ser que se pague.

La última de las herramientas nombradas en este apartado de licencias *especiales* es Microsoft Office, actualmente Office 365. La licencia de este conjunto de productos, incluyendo Word y PowerPoint, es Office 365 Educación. Este paquete es totalmente gratuito para nosotros al ser alumnado de la ULPGC, que está dada de alta en este servicio para que todos los alumnos y profesores puedan optar a esto.

El proceso es sencillo y consiste en aportar nuestro correo institucional para optar a la descarga de estos productos de Microsoft. Naturalmente, Microsoft se reserva todos los derechos para poder revisar la validez de la inscripción al centro educativo, en última

instancia, si seguimos cumpliendo los requisitos para seguir usando estos servicios.

4.2. Leyes

Realmente, pocas leyes se pueden destacar para los módulos tan concretos que se han desarrollado. No obstante, se comentará como es habitual la ley más importante y popular en los últimos años de avance tecnológico: La LOPD o Ley Orgánica de Protección de Datos (recientemente reformada a la RPGD).

Ley Orgánica de Protección de Datos (LOPD)

La Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos (LOPD) (41), es una de las leyes más importantes y conocidas dentro del mundo del Software. Hace unos cuatro años aproximadamente se ha reformado esta ley en España para adaptarse al reglamento europeo, que se impuso con dos años de antelación a esta reforma (hace unos seis años aproximadamente). El resultado de esto es el Reglamento General de Protección de Datos (RPGD) (42).

Al ser una reforma de la antigua LOPD, la RPGD conserva algunos de aspectos relevantes de esta. Independientemente de las novedades de la reforma, siempre ha perseguido el objetivo de garantizar y proteger las libertades y derechos de las personas físicas en lo que respecta a sus datos personales (intimidad, privacidad, …). Algunas de las reformas (novedades) de esta reforma son las siguientes:

- El derecho a la portabilidad. Exige la facilidad para portar tus datos. Un ejemplo claro se puede observar en el mundo de la telefonía, que planteaba un auténtico infierno si se decidía cambiar de compañía.
- El derecho al olvido. Plantea el derecho a que un titular de cualquier dato personal pueda borrarlo, bloquearlo o suprimirlo en cualquier momento.
- La necesidad de una solicitud explícita para garantizar el consentimiento del usuario en cualquier toma de datos que lo involucre. Este aspecto básicamente cubre ciertos vacíos legales de la antigua LOPD, garantizando el máximo control sobre nuestros datos personales.

Aunque solo se han expuesto tres ejemplos de reforma, esta RGPD ha traído muchas novedades beneficiosas a los titulares de datos personales. Sin embargo, en nuestro proyecto (ambos módulos) no se puede observar una actuación clara de este reglamento. Esto no quiere decir que, en un futuro, esto cambie, siendo por esto por lo que se ha decidido incluir como contenido de esta memoria. En un futuro, no es una locura pensar que este reglamento actúe en los siguientes aspectos de nuestros módulos (o el IPS en el que se incorporan):

- Anonimato de posibles datos tomados de dispositivos móviles (para contribuir a la mejora del IPS).
- Datos almacenados durante el proceso de posicionamiento: lugar, derechos del establecimiento…

Estos son solo dos de los posibles ejemplos donde podría actuar la RPGD. Para ello, se usarían, por ejemplo, métodos de encriptado, solicitudes explicitas al usuario… Grosso modo, con esto se concluye el apartado de *Leyes* y con ello el anexo *Normativa y Legislación.*

5. Competencias

Este TFM satisface todas las competencias del máster (43) y en consecuencia las competencias relacionadas del grado en cuestión (44). En este capítulo se comentarán principalmente las especificadas en la redacción de la solicitud de este TFM.

Competencia G2

Esta competencia entra dentro de la categoría de competencias Generales. Esta competencia enuncia:

> *Aplicar sus conocimientos a su trabajo o vocación de una forma profesional y posean las competencias que suelen demostrarse por medio de la elaboración y defensa de argumentos y la resolución de problemas dentro de su área de estudio.*

Esta es una de las competencias generales más importantes, que se ve reflejada con el resultado de este TFM y la predisposición a la hora de realizar todas las tareas.

Competencia N3

Esta competencia entra dentro de la categoría de competencias de la ULPGC. Esta competencia enuncia:

> *Contribuir a la mejora continua de su profesión, así como de las organizaciones en las que desarrolla sus prácticas a través de la participación activa en procesos de investigación, desarrollo e innovación.*

Ésta en concreto es una competencia satisfecha en el sentido de que se ha afrontado un problema nuevo y, además, un problema perteneciente a un proyecto de investigación de un instituto universitario, lo cual refleja su carácter investigador e innovador.

Competencia T9

Esta competencia entra dentro de la categoría de competencias de Título. Esta competencia enuncia:

> *Capacidad para resolver problemas con iniciativa, toma de decisiones, autonomía y creatividad. Capacidad para saber comunicar y transmitir los conocimientos, habilidades y destrezas de la profesión de Ingeniero Técnico en Informática.*

Ante los problemas que han surgido, hemos proporcionado nuestro punto de vista y adicionalmente hemos estado receptivos a mejoras o nuevas propuestas en las diferentes reuniones realizadas para llevar a cabo el proyecto. Con esto quedaría satisfecha esta competencia.

Competencias CII05 y CII06

Estas competencias entran dentro de la categoría de competencias comunes a la Ingeniería Informática. Esta competencia enuncia:

- *CII05: Conocimiento, administración y mantenimiento sistemas, servicios y aplicaciones informáticas.*
- *CII06: Conocimiento y aplicación de los procedimientos algorítmicos básicos de las tecnologías informáticas para diseñar soluciones a problemas, analizando la idoneidad y complejidad de los algoritmos propuestos.*

Debido a la naturaleza del desarrollo de este proyecto y de las soluciones planteadas, se han cumplido las expectativas expuestas en estas dos competencias. Todo esto se debe a que las soluciones planteadas tienen de base un análisis consistente en diferentes posibilidades, escogiéndose, de forma justificada y razonada, la mejor de ellas para la/s aplicación/es final/es.

Competencias adicionales

Si bien las competencias comentadas previamente (presentes en el documento de solicitud) hacen referencia a competencias (a) generales, (b) de la ULPGC y (c) comunes de Ingeniería Informática, se comentarán algunas competencias cubiertas que son específicas del máster cursado. Hay que aclarar que son bastante similares debido a que el máster es un complemento formativo del grado.

El impulso de ideas, así como la aplicación de conocimientos adquiridos para afrontar nuevos problemas son ejemplos de competencias del máster que hemos cumplido, ya que el problema afrontado es totalmente novedoso para nosotros y además se adentra en un campo sometido a constantes aportaciones y soluciones de diversidad variada.

Para finalizar con este apartado, hay que comentar que también hemos satisfecho la mayoría de las competencias específicas de tecnologías informáticas, si no todas. Es así ya que en la mayoría de las situaciones se han ejecutado labores relacionadas con todo el proceso de desarrollo en diferentes ámbitos (interacción persona-ordenador, funcionamiento correcto y óptimo de aplicaciones…).

6. Metodología y Planificación

En este capítulo de la memoria se describirán tanto la metodología de trabajo empleada a lo largo del desarrollo como la planificación del proyecto.

6.1. Metodología

Aunque nuestra labor en el proyecto no es dar solución total al mismo, sino centrarnos en dos módulos concretos (los ya comentados *estación meteorológica* y *módulo orientador),* se han tenido que aplicar conceptos provenientes de metodologías existentes para comprender y encaminar el transcurso de las tareas.

Hay que aclarar que nos encontramos ante un proyecto algo diferente de lo habitual, es decir, diferente a un producto software tradicional. Esta afirmación asienta sus raíces en varios puntos que comentaremos brevemente a continuación.

El primero es el carácter físico de los módulos desarrollados. Por no alargar demasiado este punto, contamos no solo con un desarrollo a nivel de software (desarrollo principal), sino con una vertiente hardware en la que tenemos que contemplar y analizar posibilidades en cuanto a diseño físico se refiere.

El segundo y último punto es el esencial carácter investigador que se brinda con este proyecto, ya que está vinculado a un instituto universitario (IUCTC). Por ello, en cada toma de decisiones es habitual estudiar algunas características del estado actual del proyecto (como el estado de otros módulos) y contemplar el estado del sector relacionado (incluyendo posibles avances). Todo lo expuesto en este párrafo es clave para procurar acertar con las decisiones en el desarrollo de nuestros módulos.

Como cualquier producto de carácter innovador, no existen certezas sobre las soluciones planteadas, por lo que los riesgos son indudables y pueden llevar a la necesidad de replantear o incluso desechar parte del trabajo. Es por eso por lo que otro esfuerzo es contemplar soluciones flexibles, moldeables a esta situación. Afortunadamente esta situación no es demasiado probable en nuestro caso, gracias a la naturaleza de los módulos (claros a nivel funcional) y a la validación iterativa empleada durante el desarrollo. Lo peor que podría pasar para nuestros módulos es que las tecnologías escogidas resulten no ser las mejores o que tengamos que remodelar parte de las soluciones planteadas para ajustarnos a las especificaciones/cambios del proyecto.

Por las justificaciones expuestas en los anteriores párrafos, es difícil asemejar la metodología usada con algunas de las metodologías tradicionales. Por esto mismo, como la metodología empleada no encaja al cien por cien con ninguna de las existentes, se cree conveniente plasmar de la forma más objetiva posible la forma en la que se encauzó el proyecto. Después de esto, se podrá realizar una mención a metodologías existentes relacionadas con nuestra forma de proceder.

Metodología propia

Como en ambos módulos contamos con una parte hardware y una parte software, no parece muy adecuado *meterlas en el mismo saco*. Es por ello por lo que, en ambos casos, contamos con fases de desarrollo algo *independientes*. Al margen de estas fases, lo que está claro es que no se puede abordar la parte software sin antes tener un producto hardware funcional (no necesariamente en su diseño final).

En el caso del producto hardware, se plantean fases de análisis y diseño que definen y moldean de forma iterativa una solución que cubra todos los requisitos funcionales (incluyendo características y detalles técnicos) que se nos presentan para cada módulo. Una vez se concluye con estas fases (aceptación), se procede a la adquisición del hardware y al montaje oportuno, materializando la solución planteada.

Como parte del IPS, está claro que nuestros módulos se encapsularán de alguna manera en el diseño final de la plataforma robótica, pero no debemos confundir esto con la implementación preliminar (funcional) de nuestros módulos. Evidentemente, dependemos de otros avances ajenos a nuestro desarrollo, así como de decisiones de alto nivel para realizar el encapsulamiento final. Aunque es bastante probable que se contribuya a ello, no se incluye en los objetivos propios de este TFM.

Antes de proseguir, existe una excepción respecto al segundo módulo (módulo orientador). En este módulo nos viene planteado el hardware que usaremos, por el hecho de que se usa ya en otros módulos del IPS. Por temas de congruencia y por la buena experiencia con el susodicho hardware, no parece lógico aventurarnos al cambio en este caso. El hardware en cuestión es totalmente funcional por sí solo (una vez habilitado y configurado), por lo que la fase de desarrollo hardware para este módulo se simplifica a estudiar e investigar el hardware que se nos presenta, dedicando los principales esfuerzos al desarrollo software.

En el caso del producto software, sí que se sigue una línea más estándar del desarrollo. Se cubren todas las fases típicas de un producto software como el análisis, el diseño, la implementación y las pruebas/despliegue (donde simplemente se puede ver cómo se comporta el producto final).

Dentro de este aspecto software, se plantea un esquema iterativo bastante particular. Tomando una perspectiva general del asunto, se marcan inicialmente unos requisitos básicos que debe tener el producto (fruto de un consenso mediante diferentes reuniones). Con esos requisitos claros, el objetivo es plantear y desarrollar un producto mínimo viable que cumpla las expectativas (primera iteración).

Una vez en este punto y con el primer producto funcional validado, se rescatan (o se idean) requisitos algo más *secundarios* para incorporar al producto y hacerlo más completo. Estos se irán añadiendo y validando en sucesivas iteraciones, de forma que al final de cada iteración se obtiene un producto funcional cada vez más completo, hasta llegar al producto final (abierto a cambios y mejoras futuras, evidentemente).

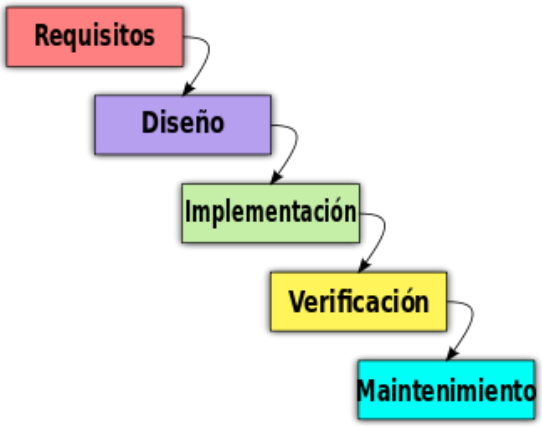


Figura 6.1. Modelo tradicional de desarrollo en cascada.

Metodologías referentes

Con toda esta información, pasamos a las metodologías referentes que han dado forma a nuestro modo de plantear el proyecto. Con los conocimientos adquiridos en la carrera y luego reforzados en el máster, se ha planteado una metodología algo propia y libre con el uso de algunos conceptos conocidos.

Aunque puede parecer inconsistente, al incluir un desarrollo *a dos bandas* (hardware y software), no es una locura pensar que en parte se recogen fragmentos de la esencia del modelo tradicional en cascada (figura 6.1) (45). En nuestro caso, esta similitud es simplemente una barrera, una restricción casual: no poder pasar al siguiente punto de desarrollo (software) hasta que se termine en anterior (hardware). Conocemos de sobra las flaquezas de este modelo (variación de requisitos durante el desarrollo), pero como comentamos antes, son solo partes esenciales que nos recuerdan a esta metodología.

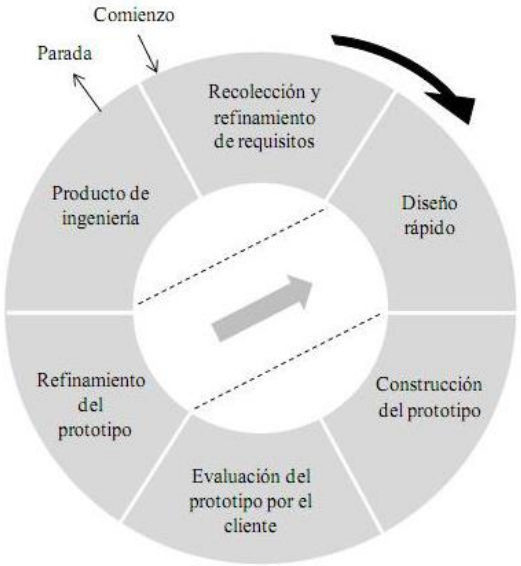


Figura 6.2. Pantalla principal de visualización y control de nuestro motor (a nivel de interfaz).

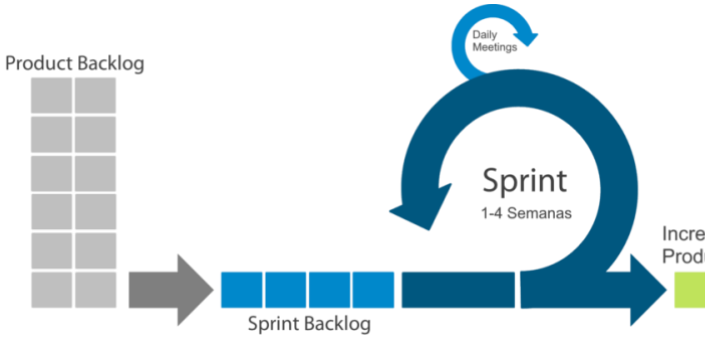


Figura 6.3. Pantalla principal de visualización y control de nuestro motor (a nivel de interfaz).

Al igual que nos ocurre con el modelo en cascada, tenemos el modelo de prototipos (figura 6.2) (46). Si nos paramos a pensar, se pueden encontrar similitudes con algunos principios de este modelo. Inicialmente no se tiene certeza sobre la solución que se desea aportar y son la investigación y las sucesivas reuniones (validación de decisiones) las que van conformando el producto. No se pasa a la siguiente fase hasta que el *prototipo* elaborado sea validado y aceptado por el cliente (en nuestro caso, el autor de este TFM y el grupo de investigación del IUCTC vinculado al proyecto).

Realmente, la principal metodología referente se verá a continuación, pero se estimó oportuno mencionar también este modelo de prototipos (sobre todo al tratarse de un producto bastante *práctico* que integra hardware y software). Siguiendo las líneas de una metodología iterativa e incremental pura, la principal metodología referente ha sido Scrum (47). Es casi imposible no mencionarla en nuestro proyecto, puesto que es una de las metodologías más destacadas en nuestro ámbito y también de las más estudiadas a lo largo de nuestro proceso formativo. En la figura 6.3 se observa una perspectiva global de esta metodología, aunque se enumeran más conceptos relacionados a continuación:

- *Pila de producto (Product Backlog)*. Recoge el conjunto global de historias de usuario que se implementarán a lo largo del desarrollo. Esta pila de producto se establece inicialmente y puede variar a lo largo del tiempo (excepto las historias que han pasado a implementarse).
- *Sprints e incrementos*. Un *sprint* consiste en el desarrollo de un conjunto de historias de usuario (funcionalidades que definen lo que se conoce como *Sprint Backlog*) durante un periodo de tiempo relativamente corto (entre dos y cuatro semanas normalmente). Al realizar y completar un *sprint* se obtiene un incremento, que es un producto funcional y entregable al cliente, que será incrementado en futuras iteraciones, de ahí el origen del nombre.
- *Roles principales*. Aunque no es relevante en nuestro caso, ya que es un proyecto limitado al ámbito académico y, por lo tanto, limitado también en las condiciones de ejecución, mencionaremos algunos de los roles principales en esta metodología. Esos roles son el *Product Owner* (representante del cliente), el *Scrum Master* (aunque no es el término exacto, es una especie de *líder* del equipo

de desarrollo que se encarga del cumplimiento de la metodología Scrum y brinda apoyo en diferentes tareas del desarrollo) y el propio equipo de desarrollo.

Naturalmente, ni de lejos nuestro desarrollo se puede asociar directamente a esta metodología, ya que no se han definido historias de usuarios en ningún momento (simplemente una lista de requisitos a raíz de diferentes reuniones) ni se han tenido o marcado unos plazos estándar para cada iteración (como ocurre con los *sprints* en Scrum). Eso sin contar con la imposibilidad de tener personal con los roles participantes en esta metodología. Sin embargo, sí que se pueden encontrar similitudes con Scrum en los siguientes aspectos:

- Se acordaron un conjunto de requisitos mínimos para desarrollar en la primera iteración, recordando algo parecido a la pila de producto.
- Se agruparon requisitos para desarrollar durante cada una de las iteraciones, tanto nuevas funcionalidades como posibles correcciones de imperfecciones o errores.
- Se obtiene un producto funcional después de cada iteración, que el cliente puede ejecutar y valorar de forma libre.

6.2. Planificación

En este apartado se comentará una perspectiva general de la planificación seguida. No obstante, no se entrará en demasiado detalle técnico ni se comentarán los resultados de cada una de las fases establecidas, esto se reserva para apartados posteriores.

Antes de comenzar con las fases de desarrollo para cada módulo, conviene aclarar que se destinaron horas de trabajo de la planificación a realizar un estudio y una formación propia en el contexto del proyecto (*Fase de estudio previo / análisis*). Esto incluye un *paseo* por el estado del arte, plasmado en el correspondiente apartado de esta memoria, así como asimilar el contexto actual en el que se encontraba el proyecto (estado de la plataforma robótica, estado de otros módulos…). Aunque estas tareas no presentan un resultado *funcional*, es innegable que ayudan a direccionar el desarrollo completo en una línea más cauta y certera. Destacamos tareas como: la lectura de artículos del grupo de investigación relacionados con los IPS y la contextualización sobre la tecnología Bluetooth.

Satisfecha esta etapa, proseguimos a elaborar primeramente la estación meteorológica y luego el módulo orientador (cada cual con sus fases de desarrollo). Por coherencia a la metodología iterativa planteada, dejar el módulo en un estado final *aceptable* no implica que este quede libre de adaptaciones y cambios posteriores (dentro y fuera del ámbito de este TFM). A continuación, se expondrán más detalles de las tareas desarrolladas para cada uno de estos módulos.

Estación meteorológica

Comenzando con la parte hardware, durante la fase de análisis se estudiaron qué factores meteorológicos eran los que más afectaban a las señales Bluetooth. Tras la lectura de varios artículos y la información recopilada se llegaron a unas conclusiones,

que fueron plasmadas en un informe.

Al análisis le sigue la fase de diseño, donde se realizó una amplia búsqueda de componentes candidatos para nuestro sistema en función de los requisitos planteados. El resultado fue también un informe agrupando los componentes candidatos. Hay que recalcar que el diseño fue plasmado sin problema, obteniendo el producto físico y completando el *hueco en blanco* que nos impedía comenzar el desarrollo software.

Zanjados los aspectos físicos del módulo, quedan los aspectos programáticos. En este caso, aunque se plantean algunas claves iniciales en el análisis (requisitos básicos para un producto inicial funcional) y diseño (planteamiento de una arquitectura básica coherente con el resto del IPS), cada una de las fases de desarrollo está presente en todas las iteraciones ejecutadas, en mayor o menor medida.

Casualmente el resultado del análisis es totalmente dependiente del diseño físico, es decir, previamente se definieron las variables que nuestra estación meteorológica contemplaría, así como los componentes que usaría para ello. En consecuencia, las variables para tener en cuenta en el desarrollo software son exactamente las mismas que las planteadas para la parte física.

Durante el diseño se plantearon dos claves iniciales que marcarían las bases del diseño durante el resto de las iteraciones: (a) arquitectura del sistema desarrollado de cara a la integración del módulo en el sistema IPS y (b) lenguaje de programación que se usaría.

La implementación y las pruebas/despliegue consistirán básicamente en desarrollar y exponer el producto en cada una de sus iteraciones.

Módulo orientador

Damos un giro de ciento-ochenta grados para centrarnos en el segundo módulo, el módulo orientador. Seguiremos una línea similar al anterior módulo para exponer la planificación.

A diferencia de la estación meteorológica, este módulo nos brinda una restricción en cuanto al análisis, diseño y montaje de la arquitectura hardware. Como se adelantó en el apartado de *Metodología,* se usarán los mismos motores usados para el direccionamiento y tracción de las ruedas de la plataforma robótica de nuestro IPS, por simple coherencia y por la positiva experiencia con este sistema hardware.

A pesar de todo esto, dedicaremos algunos párrafos de nuestra memoria a comentar algunos aspectos técnicos del sistema hardware integrado en este módulo antes de proceder con la parte software. Olvidando estos detalles, consideramos el motor listo para usarse en el desarrollo software.

Respecto a la parte software de este módulo, no cambia demasiado respecto a la estación meteorológica. Se plantean unas claves iniciales en el análisis y el diseño (conceptualmente las mismas que las comentadas en la estación meteorológica), pero al margen de esas claves, se integran todas las fases del desarrollo en cada iteración.

Los detalles de las claves iniciales del análisis difieren considerablemente respecto al anterior módulo. En este caso el hardware es relativamente sencillo porque se trata de un eje con capacidad de rotación, pero esta vez los requisitos no vienen marcados exclusivamente por el hardware. El análisis cobra más protagonismo en este módulo, ya que la definición de requisitos marcará el funcionamiento concreto y correcto del eje del motor.

En cuanto a las anotaciones relativas al resto de fases del desarrollo, son exactamente las mismas que se tuvieron en cuenta para el anterior modulo, por lo que no hay necesidad de repetirlas. El único aspecto que destacar es que este módulo planteó una mayor complejidad que el anterior y, en consecuencia, más iteraciones.

Dedicación final

Para cerrar el aspecto de planificación, se expone la planificación oficial estimada para la realización de este TFM, sobre la que se realizarán comentarios relacionados con las desviaciones.

En la fase de Estudio previo / Análisis, la tarea que sin duda supuso un mayor esfuerzo fue el análisis del módulo meteorológico (iteraciones para validar el conjunto final de variables). Sin embargo, en conjunto con las otras tareas, la dedicación final resultó en menos horas de las previstas, por lo que esta tarea se sobreestimó.

El conjunto de horas estimadas para el diseño del orientador (nivel hardware) supuso una carga inesperada de trabajo (alternativas, adquisición, montaje…). Igual ocurre en el caso de la implementación del módulo orientador, que, aunque parecía exponer unos requisitos triviales, derivó en la dedicación de más horas de las previstas. Sin embargo, esto se compensó con el resto de las tareas y, aunque se subestimó esta fase, la dedicación final no se desvió excesivamente.

Por el flujo natural que ha seguido el desarrollo, las fases de evaluación, pruebas y validación se define como una fase sobreestimada. Las reuniones y las demostraciones de funcionamiento fueron claves para simplificar bastante este proceso.

La documentación ha sido una tarea que ha consumido algunas horas extra en su realización. Siempre es un aspecto que se subestima, pero la redacción y la constante revisión y reestructuración del contenido son tareas de gran peso.

A pesar de estas desviaciones, el cómputo global de la dedicación final se aproxima a las 300 horas estimadas en la planificación.

Fases	***Duración Estimada*** ***(horas)***	***Tareas*** ***(nombre y descripción, obligatorio al menos una por fase)***
Estudio previo / Análisis	60	Tarea 1.1: Estudio (Revisión) del contexto de los IPS.
		Tarea 1.2: Análisis del módulo meteorológico. Se determinarán y confirmarán las variables que afectan a nuestro Sistema de Posicionamiento en Interiores.
		Tarea 1.3: Análisis del módulo orientador. Se plantearán los parámetros con los que se deberá jugar para el correcto funcionamiento de este módulo y, por tanto, del Sistema de Posicionamiento en Interiores.
Diseño / Desarrollo / Implementación	150	Tarea 2.1: Diseño e implementación del módulo meteorológico. En base al análisis realizado y en función de los componentes disponibles, se diseñará este módulo tanto a nivel físico como lógico. Esto también supondrá la final integración del módulo con la plataforma robótica.
		Tarea 2.2: Diseño e implementación del módulo orientador. Al igual que la tarea 2.1, en base al análisis realizado y en función de los componentes disponibles, se diseñará este módulo tanto a nivel físico como lógico. Esto también supondrá la final integración del módulo con la plataforma robótica.
Evaluación / Validación / Prueba	50	Tarea 3.1: Pruebas del módulo meteorológico. Se diseñarán todos los test posibles y se llevarán a cabo pruebas de funcionamiento en un entorno realista para determinar la validez del módulo diseñado en el ámbito de las variables meteorológicas contempladas. Se comprobará si la exactitud del Sistema de Posicionamiento en Interiores es la correcta en función a los datos adicionales que aporta el módulo meteorológico.
		Tarea 3.2: Pruebas del módulo orientador. Pruebas del módulo meteorológico. Se diseñarán todos los test posibles y se llevarán a cabo pruebas de funcionamiento en un entorno realista para determinar la validez del módulo diseñado en el ámbito de la orientación del dispositivo de comunicación (en diferentes orientaciones). Se comprobará si la exactitud del Sistema de Posicionamiento en Interiores es la correcta en función de los parámetros contemplados.
Documentación / Presentación	40	Tarea 4.1: Elaboración de la documentación final para la culminación del proyecto descrito (memoria, presentación, entre otros).

Figura 6.4. Definición oficial de la planificación estimada de este TFM.

7. Descripción del proyecto

En este capítulo se presenta la descripción del desarrollo del proyecto para ambos módulos por separado (segmentados en varios apartados), pasando primero por una introducción general.

7.1. Introducción

Como se ha podido comentar en ocasiones anteriores, el tema base de este proyecto es el posicionamiento en interiores, aunque nuestro trabajo se centra en dos de sus módulos: estación meteorológica y módulo orientador.

Ya se ha explicado de forma descriptiva la motivación e importancia de estos módulos. Aun así, se remarcarán y detallarán a continuación junto a todas y cada una de las fases del desarrollo ejecutadas para ambos módulos. En estas explicaciones se descartará contenido ya comentado como la metodología o la planificación.

7.2. Módulo A. Estación Meteorológica.

Dentro del posicionamiento en interiores este módulo cobra una importancia considerable. Siempre ha existido cierta incertidumbre en lo que respecta a los factores meteorológicos que afectan a la transmisión de señales de radiofrecuencia como las señales Bluetooth. Como esta es la tecnología protagonista en nuestro proyecto, es lógico estudiar y considerar las variables meteorológicas que afectan en la transmisión y recepción de estas señales para mejorar la precisión nuestro IPS.

Basándonos en el análisis y el estudio, contemplaremos los factores ambientales que deberían considerarse en nuestro IPS. El resultado final de este módulo también tiene como objetivo oculto reforzar el conocimiento existente en este ámbito, aportando evidencias (futuras) de si los datos medioambientales contemplados verdaderamente afectan a las señales Bluetooth. Dicho de otro modo, se podrá valorar el verdadero impacto de estas variables meteorológicas.

7.2.1. Hardware

En este apartado hardware de la estación meteorológica se describirán en detalle sobre las fases ejecutadas, desde el estudio de variables meteorológicas que pueden afectar a las señales Bluetooth hasta el diseño y montaje final.

7.2.1.1. Análisis

Durante esta fase de análisis de la estación meteorológica, se profundizan conceptos sobre la tecnología Bluetooth (y relacionadas) con el objetivo de determinar qué factores repercuten en mayor o menor medida en la transmisión y recepción de las señales. Se asume la tecnología Bluetooth como protagonista por la propia arquitectura de nuestro IPS (plataforma robótica y grupo de balizas Bluetooth desplegadas estratégicamente por el entorno).

A continuación, se cubren diferentes aspectos como la naturaleza de las señales Bluetooth, el espectro electromagnético y la aclaración de algunos conceptos relacionados. En definitiva, lo que se pretende es definir el conjunto de variables ambientales que nuestro IPS debe considerar, explicando cómo y en qué medida afectan a las señales comentadas (e incluso relaciones entre variables). Hay que aclarar que el conjunto final de variables no está cerrado a ampliaciones/modificaciones.

Naturaleza de las señales Bluetooth

Aunque seguramente ya se ha comentado en apartados anteriores de esta memoria, las señales Bluetooth comparten un amplio rango de frecuencia con otras muchas señales procedentes de diferentes aparatos/tecnologías (como la Wifi). Este rango de frecuencia es el de los 2.45 GHz. Los siguientes elementos son ejemplos de tecnologías que transmiten en este rango de frecuencia (48) (49) (50):

- Tecnologías estándar (Bluetooth y Wifi básicamente): Bluetooth clásico, BLE, IEEE 802.11, IEEE 802.11b, IEEE 802.11g, IEEE 802.11n y IEEE 802.15.4. Consecuentemente, los dispositivos móviles o teléfonos inalámbricos se incluyen en este *saco.*
- Dispositivos de suministro de energía no interrumpible (UPS).
- Televisión de difusión al aire libre utilizada por cámaras de televisión móvil.
- Dispositivos particulares como: ordenadores, aires acondicionados o incluso dispositivos lumínicos.
- Otros elementos como: repetidores de video, hornos microondas, alarmas inalámbricas, teclados y ratones, luces fluorescentes, farolas, transformadores…

A todos estos elementos, se le suman materiales que dificultan la transmisión de señales, como, por ejemplo, el agua. El agua tiene una gran capacidad para absorber este rango de ondas sobre los 2.4GHz (51). Esto supone que el propio cuerpo humano (las grandes multitudes) también es un impedimento para estas ondas, pues estamos compuestos mayoritariamente por agua (52).

En el artículo *Network+ Exam Cram: Wireless Networking,* por Mike Harwood (53), se comenta una larga lista de los elementos que afectan en mayor o menor medida a las señales en estos rangos de frecuencias, visibles en la figura 7.1 y resumidos a continuación:

- Con un bajo impacto: madera, paneles de madera, paneles de yeso, inmobiliario y cristales transparentes (ventanas, por ejemplo), entre otros.
- Con un medio impacto: Cristales tintados, gente (en multitud, sobre todo), baldosas de cerámica y bloques de cemento/hormigón.
- Con un alto impacto: Bloques de cemento/hormigón (al igual que antes), espejos (debido al reflejo de las ondas), metales y agua (en acuarios, lluvia, fuentes…).

Table 7.4. Wireless Obstacles Found Indoors

Obstruction	Obstacle Severity	Sample Use
Wood/wood paneling	Low	Inside a wall or hollow door
Drywall	Low	Inside walls
Furniture	Low	Couches or office partitions
Clear glass	Low	Windows
Tinted glass	Medium	Windows
People	Medium	High-volume traffic areas that have considerable pedestrian traffic
Ceramic tile	Medium	Walls
Concrete blocks	Medium/high	Outer wall construction
Mirrors	High	Mirror or reflective glass
Metals	High	Metal office partitions, doors, metal office furniture
Water	High	Aquariums, rain, fountains

Figura 7.1. Elementos que afectan a la transmisión de señales de radiofrecuencia.

Para hacer frente a esas dificultades, los dispositivos que usan la tecnología Bluetooth implementan por defecto un salto de frecuencia adaptativo o AFH (*Adaptative Frequency Hopping*), que a grandes rasgos minimiza interferencias cambiando entre diferentes frecuencias dentro de los 2.45 GHz (evitando las más saturadas). Gracias al AFH las interferencias se minimizan razonablemente, aunque no por ello podemos ignorar todos estos factores que afectan a la transmisión de señales.

Aparte de los ya nombrados, existe un tipo de factor que afecta a las señales de radiofrecuencia y que no podemos controlar (o es más difícil). Este es el factor medioambiental, que integra variables medibles e incontrolables que pueden afectar a nuestro IPS. A continuación, consideraremos las variables ambientales más importantes que existen y que nos podrían afectar.

Principales variables ambientales

Las variables ambientales condicionan la transmisión de señales de radiofrecuencia, sobre todo en entornos exteriores. A continuación, nombraremos las variables que afectan en entornos exteriores, remarcando cuales afectan también en interiores.

Como bien se comenta en el breve artículo publicado por Raymond L Johnson, *¿Afecta el tiempo a su señal Wifi? Conoce la verdad* (54), existen múltiples factores climatológicos que afectan a una señal Wifi, que realmente es análoga a la Bluetooth por

desenvolverse en el mismo espectro de frecuencias:

- *Lluvia*: Este elemento está compuesto de agua, que como comentamos antes, afecta bastante en la transmisión de señales (absorción de señal), generando constantes interrupciones en la señal. Adicionalmente, la lluvia afecta también en términos de visibilidad (cuento menos visibilidad, más dificultad de transmisión). La lluvia afecta solo a entornos exteriores.
- *Niebla*: Este elemento causa dificultad de transmisión debido a la perdida de visibilidad que conlleva. Quizás la niebla afecta solo en exteriores, pero en caso de que se diera en interiores pasaría a ser una de nuestras variables.
- *Humedad*: Este elemento es un fenómeno climático parecido a la lluvia, ya que, de una forma u otra, existe presencia de agua. La humedad perjudica la intensidad de transmisión y afecta negativamente (provoca interrupciones, alteraciones en la señal…). Afecta tanto en exteriores como en interiores.
- *Temperatura*: Este elemento tiene un impacto indirecto sobre los aparatos que integran el manejo de señales. Estos aparatos están diseñados para actuar a ciertos rangos de temperatura y cuando se supera/rebaja ese límite, el funcionamiento puede no ser el correcto. De una forma u otra, la temperatura si afecta a la transmisión de señales, aunque no afecte directamente al medio de transmisión (55). La temperatura afecta tanto en exteriores como en interiores.
- *Electricidad*: Ejemplos de este elemento podrían ser las tormentas eléctricas (en exteriores) o los aparatos electrónicos (principalmente en interiores). A todo esto, en relación con esta categoría tenemos la luz.

El resumen a los puntos comentados es que tanto la humedad como la temperatura representan variables que nos afectan y, por tanto, son variables de interés. A este conjunto se le suma la luz (medible), que, aunque se encuentra en *terreno pantanoso*, el hecho de que muchas balizas Bluetooth la contemplen nos anima a incluirla. Aunque la presencia de dispositivos electrónicos también puede afectar, no entra en la categoría de variable climatológica, por lo que se descarta.

En siguientes apartados, se profundizará más en la variable de la luz ambiente (que es la más dubitativa), comentando para ello el espectro electromagnético. Asimismo, se tratará de estudiar un poco la relación entre las dos variables más importantes: la humedad y la temperatura.

Ondas y espectro electromagnético

A lo largo de la documentación revisada (56) (57) (58) (59) (60), hemos estudiado bastantes conceptos clave. A continuación, exponemos uno en particular: el concepto de onda. Este será necesario para valorar la variable de luminosidad y se define así:

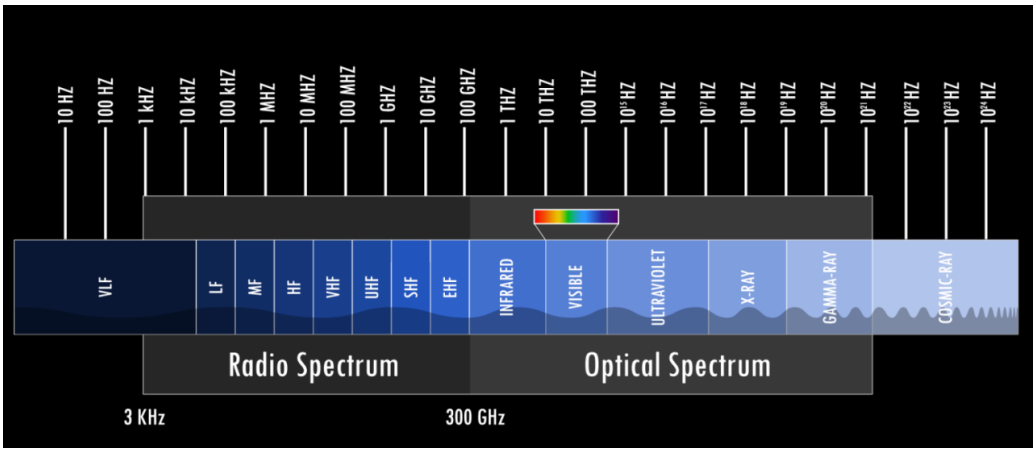


Figura 7.2. Espectro electromagnético con las diferentes frecuencias.

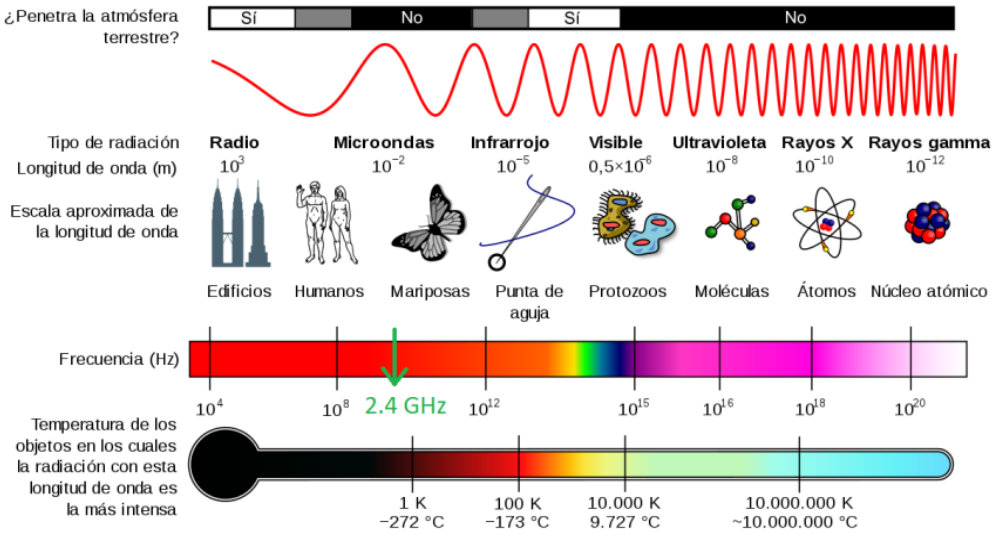


Figura 7.3. Bandas de frecuencias, longitudes de onda, escalas y radiaciones.

> *Propagación de una perturbación de alguna propiedad del espacio, por ejemplo, densidad, presión, campo eléctrico o campo magnético, implicando un transporte de energía sin transporte de materia en un espacio que puede o no contener materia (aire, agua… o el vacío si no contiene materia). En el caso del sonido, el espacio perturbado contiene materia porque si no, no se puede transmitir, mientras que, en el caso de la luz, no sería así.*

Tanto la luz como el sonido son propiedades asociadas al concepto de onda, aunque en este caso se hablará solo de la luz y del espectro electromagnético (conjunto de ondas electromagnéticas). Cuando hablamos de ondas electromagnéticas incluimos las señales de transmisión como la radio frecuencia, la luz visible…

En la figura 7.2 se muestra claramente el espectro electromagnético. De forma complementaria, en la figura 7.3 se exponen datos como la relación entre longitud de onda y la frecuencia, radiaciones, escalas... En cualquier caso, se pueden apreciar los

diferentes rangos de frecuencia y la separación entre el espectro de radio y el óptico. Las señales Bluetooth (y similares) se encontrarían en el primer grupo (radio) mientras que cualquier tipo de luz (visible, infrarroja…) se encontraría en el segundo grupo (óptico). Todo esto nos lleva a pensar en un primer momento que no existen solapamientos ni interferencias entre la luz y las señales de radiofrecuencia.

Ahora se hará un pequeño inciso para comentar los canales de frecuencia a las que suelen funcionar las señales Wifi (figura 7.4). Estas se mueven en los rangos de frecuencia de los 2.4 GHz. Concretamente pueden transmitirse por diversas frecuencias o canales, que se enumeran del 1 al 13 y ocupan aproximadamente unos 22MHz. Los canales exclusivos entre sí (sin interferencias entre ellos) serían el 1, el 6 y el 11. Todo lo comentado sería análogo para el Bluetooth.

Llegados a este punto, las conclusiones a las que llegamos son que la luz, por lo general, no genera interferencias en la transmisión de señales. En caso de hacerlo, está claro que sería porque se generan/emiten señales dentro del espectro de radio. La realidad es que existen interferencias en la transmisión de señales dependiendo del tipo de luz (natural, LED, fluorescentes, incandescentes…).

Por ejemplo, la luz natural (el sol) es protagonista en diferentes procesos de ionización en diferentes capas de la atmósfera (61), pero está lejos de afectar a las ondas de radiofrecuencia. Tan solo la luz visible (colores) y parte de la luz infrarroja y ultravioleta atraviesan la atmósfera y llegan a la troposfera (62). Esto último se puede apreciar en la figura 7.5. Aún con todo esto, la luz solar que llega a nosotros supera la banda de frecuencia de las señales de radio (63).

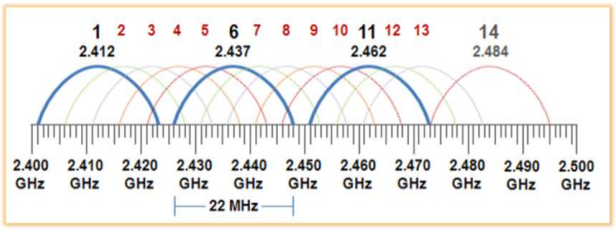


Figura 7.4. Canales y frecuencias en la transmisión de señales Wifi.

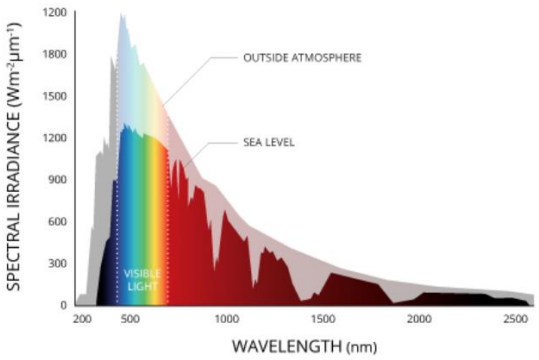


Figura 7.5. Ondas y radiación solar que traspasa la atmósfera y llega a la superficie.

Descartando la luz solar y para concluir este punto, el único tipo de luz artificial que de momento se sabe que puede afectar a las señales es la generada por los fluorescentes. Esto se debe a que emiten señales (señales residuales fruto de las reacciones químicas) en la banda de 2.4GHz (64). Por esto y por otros indicios vistos anteriormente, la luz no es una variable que se descarte a la ligera.

Temperatura y humedad

En efecto, la relación entre dos de los factores medioambientales expuestos (temperatura y humedad) es una realidad evidente. Esto se comenta en profundidad en el artículo *Effects of Temperature and Humidity on Radio Signal Strength in Outdoor Wireless Sensor Networks* (65). Obviamente, tampoco nos interesa entrar en detalles, pero es útil e interesante exponer que existe una relación entre ambas variables. A continuación, se comentan algunos conceptos clave para entender esta relación:

- Humedad. Es la cantidad de vapor de agua presente en el ambiente (invisible). La cantidad máxima de agua en el ambiente viene determinada por la temperatura ambiente, por lo que se distinguen diferentes conceptos:
    - Humedad absoluta. Humedad presente en el ambiente sin otras consideraciones (g/m3).
    - Humedad saturada. Cantidad máxima de humedad para la temperatura actual.
    - Humedad relativa. Cantidad proporcional de humedad según la temperatura actual. Si a la temperatura actual se tiene la mitad de la humedad máxima posible, la humedad relativa seria de un 50%.

Las conclusiones destacables del artículo comentado con anterioridad son las siguientes:

- La temperatura y la fuerza de la señal transmitida presentan una correlación negativa, es decir, si la temperatura aumenta, decrece la fuerza de la señal y viceversa.
- La humedad relativa y la fuerza de la señal transmitida presentan una correlación positiva por encima de los 0ºC, es decir, aumentan y decrecen juntas.
- Por otra parte, por debajo de los 0ºC la humedad absoluta muestra una correlación negativa.

A lo largo de muchos artículos, se ha tratado de estudiar que variable (temperatura o humedad) afecta más. Aunque esto sigue planteando una incógnita, lo que está claro es que ambas son de gran importancia para el estudio de señales de radiofrecuencia y que presentan una clara relación, como se expuso previamente.

Resultados del análisis

Grosso modo, se han estudiado y visto conceptos importantes para la determinación de las variables que más nos afectan en el proceso de transmisión y captación de señales

Bluetooth (y similares). Con ese pequeño estudio y gracias a los artículos revisados, se ha llegado a las siguientes conclusiones.

Las variables que más afectan a la transmisión de señales Bluetooth son la temperatura y la humedad, pero debido a que la luz es una variable contemplada en muchas balizas Bluetooth y algunos tipos de luz artificial generan interferencias en la banda de 2.4GHz, es otra de las variables a considerar (tres variables en total). Otras variables se han descartado por ser relevantes exclusivamente en entornos exteriores (niebla, tormentas eléctricas…).

También se ha visto cómo afectan estas variables y hasta se han nombrado relaciones entre ellas. Con todo esto, se disponen las cartas para la selección de los elementos más adecuados para la estación meteorológica (de acuerdo con el conjunto de variables final), en última instancia, el abordaje de la fase de diseño.

7.2.1.2. Diseño

Durante la actual fase de diseño de la estación meteorológica, el objetivo es plasmar un conjunto diverso de tecnologías que encajarían en nuestro IPS con el fin de satisfacer la lectura de variables concluidas en el anterior análisis.

Contexto de la plataforma

Sabemos que la plataforma robótica, en el punto de comienzo del desarrollo de este módulo, está compuesta por dos unidades de procesamiento: *un portátil y una unidad de control EV3*. No se sabe si nuestra estación meteorológica necesitará estructuras adicionales para captar y procesar datos de manera independiente y, además, para agrupar los diferentes sensores.

En este apartado no solo se contemplan sensores, sino placas de desarrollo y creación de hardware que facilitarían la tarea de captación de datos de una manera personalizada y dedicada. Estas placas se creen convenientes también para obtener un mejor encapsulamiento del módulo y para extraer dependencias del *cerebro principal* de la plataforma robótica. Aparte de las placas y los sensores, se sobreentiende el uso de componentes intermedios (placa de conexiones, cables, resistencias…).

En los puntos que se expondrán a continuación se ofertará un amplio abanico de componentes (placas y sensores) que ofrece el mercado actual. Ni de lejos se pueden abarcar todos los componentes disponibles, por lo que se recogen solo los sensores que se han creído relevantes (criterio propio, disponibilidad, facilidad de adquisición…). Otro dato importante es que los sensores seleccionados no necesariamente deben ser los definitivos. Si el conjunto de variables meteorológicas cambia o se requieren especificaciones concretas (precisión, costes…), los sensores pueden ser sustituidos. Por ello, otro objetivo del diseño es obtener un diseño flexible.

Placas de desarrollo hardware. Introducción.

De acuerdo a la decisión anterior consistente en encapsular todos los sensores bajo una placa de desarrollo hardware, se nos plantean dos posibilidades: (a) que esta placa

encapsule solo el proceso de captación de datos por parte de todos los sensores y que el procesamiento se lleve a cabo por dispositivos auxiliares, principalmente el *cerebro* del IPS; (b) que esta placa se encargue de lo anterior y, además, del procesamiento de datos (cualquier operación realizada sobre los datos obtenidos de los sensores).

A pesar de que el mercado ofrece un sinfín de posibilidades, plasmaremos en este documento las dos alternativas más populares para cada escenario. En el primer escenario tenemos las placas Arduino, mientras que en el segundo tenemos la popular Rasberry Pi.

Ambas opciones son quizás las más populares del mercado por su gran versatilidad y su reducido precio. Para tomar una decisión debemos valorar diferentes aspectos como: el precio, la comunidad, el rendimiento, la fiabilidad y la durabilidad. Otro aspecto a considerar y que es algo vertical es la compatibilidad con el lenguaje de programación. Se sabe que para este proyecto (otros módulos ajenos a este TFM) predomina el lenguaje Python y C++.

Sin embargo, ambas alternativas propuestas presentan un nivel similar (y positivo) en relación con los puntos expuestos previamente, por lo que para el proceso de selección existe la posibilidad de omitir ciertas anotaciones al respecto, centrándonos en puntos que se crean relevantes en ese momento (precio, detalles técnicos, opiniones, valoraciones, mejor adaptación a nuestras necesidades, …). En ambos casos, hay que añadir que, independientemente de las versiones de cada placa, presentan una evolución bastante estable y sin excesivas novedades respecto a la versión anterior, dedicándose simplemente a mejorar el hardware existente.

A continuación, comentaremos brevemente las opciones antes nombradas. Hay que aclarar que estas opciones se seleccionaron no solo por conocerse previamente, sino por su aparición en algunos artículos y diferentes recursos web, como es el caso de un artículo de *Arrow* (66) y un recurso web donde se sugerían alternativas atractivas a Arduino (67). Es lógico conducir la búsqueda buscando alternativas atractivas a Arduino, pues se quiera o no es una de las primeras opciones que se viene a la cabeza para un sistema de este tipo.

Placas de desarrollo hardware. Arduino.

Arduino es una placa que no solo tiene buenas opiniones de los usuarios, sino que cuenta con una amplia y reconocida lista de sensores compatibles, la mayoría accesibles desde cualquier tienda de electrónica popular. Cuenta incluso con un IDE de desarrollo propio en donde el lenguaje que se usa está basado en C++.

Sin embargo, para diferentes lenguajes como Python, existen librerías que podemos usar para extraer datos de los sensores y tratarlos de la forma apropiada para los objetivos deseados. Solo con estos puntos expuestos, ya constituye una maravillosa elección.

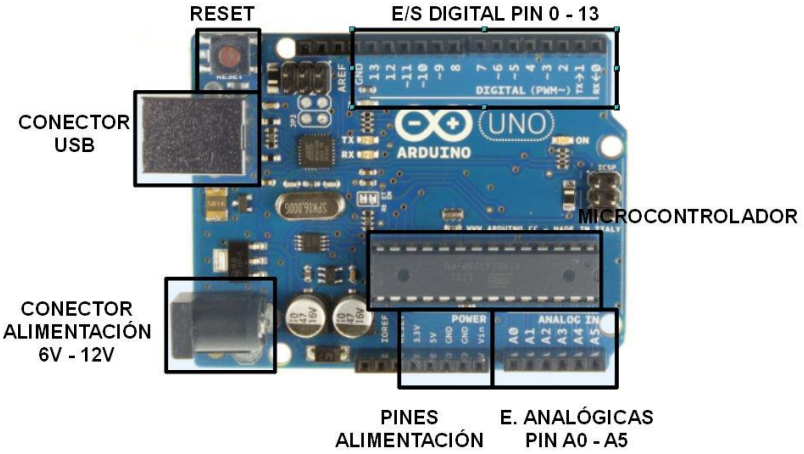


Figura 7.6. Placa Arduino UNO y sus partes (alto nivel).

A parte de lo anterior, cuenta con un evidente soporte y evolución que nos hacen reafirmarla como una de las alternativas más razonables y atractivas de acuerdo con nuestros requisitos y necesidades. Es tal que así, que cuenta con un foro personalizado y oficial (68) donde se incluyen tutoriales, resolución de dudas varias…

Por si fuera poco, Arduino es una plataforma abierta (hardware y diseños libres, expuestos a reutilizarse o mejorarse) y versátil para el desarrollo de productos electrónicos como, en nuestro caso, la estación meteorológica. Es incluso una plataforma orientada a un público no experto en electrónica ni en programación, aunque se puede combinar con lenguajes más sofisticados para elaborar productos más complicados, haciendo uso de librerías, evidentemente.

En nuestro caso particular, se podría usar Arduino Uno, pues a grandes rasgos es la placa más popular cuando hablamos de Arduino. En la figura 7.6 mostramos un desglose minimalista del aspecto y partes de esta placa, a pesar de haber ya contemplado una imagen similar en ocasiones anteriores (apartado de *Herramientas utilizadas*). Como se indica en el artículo de *Arrow* expuesto previamente (66), esta es nuestra plataforma si *se está desarrollando una aplicación de detección sencilla, una solución basada en el Internet de las Cosas, un dispositivo de adquisición de datos o un robot*.

En la figura 7.7 se puede ver de una forma mucho más detallada las especificaciones de esta placa. Sin detenernos demasiado en estos detalles, podemos observar que presenta características bastante razonables y estándar para el contexto del desarrollo de este módulo.

Otro aspecto que se menciona en el artículo de *Arrow* (66) es el siguiente: *En vez de construir una única solución para un conjunto de problemas, Arduino ha construido un ecosistema en torno a las placas Uno con el que casi cualquier producto puede ser compatible. Si bien sabemos que tal vez no sea la solución definitiva para un diseño de tipo chip-down integrado, Arduino Uno resulta ideal como punto de partida.*

Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz
LED_BUILTIN	13
Length	68.6 mm
Width	53.4 mm
Weight	25 g

Figura 7.7. Especificaciones de la placa Arduino Uno Rev3 extraída de la página oficial de Arduino (misma página que el enlace de compra).

Por todos estos motivos y como adelantamos antes, Arduino es claramente una opción a considerar. Es ideal para una solución inicial típica de un TFM. Solo faltaría considerar el precio, que tampoco es extremadamente alto. Una placa Arduino Uno REV3 presenta al público un precio de aproximadamente 20€ en su tienda oficial (69).

Placas de desarrollo hardware. Raspberry Pi.

Ahora entramos en el segundo escenario planteado. Este escenario lo protagoniza la Raspberry Pi, concretamente en su versión 4 modelo B. Podemos apreciar este modelo en la figura 7.8. Como se puede deducir es prácticamente un pequeño ordenador donde, de hecho, se integra un sistema operativo. Las opciones en este sentido van desde un sistema operativo propio hasta sistemas operativos basados en diferentes distribuciones conocidas (lógicamente adaptados a esta placa).

Por esta y otras razones, la Raspberry Pi nos brinda una mayor independencia (es *autosuficiente* a nivel de procesamiento) y presenta mayor potencia de cómputo que la placa Arduino comentada previamente. Esto se refleja en su tabla de especificaciones mostrada en la figura 7.9.

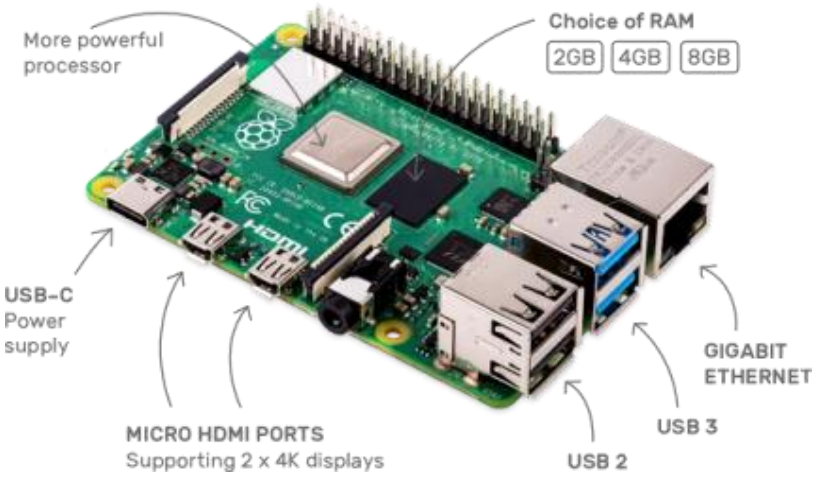


Figura 7.8. Placa Raspberry Pi (concretamente en su versión 4, modelo B) donde podemos ver gran parte de sus especificaciones.

El punto clave en la comparación entre Arduino y la Raspberry Pi radica en parte de su composición. La placa Arduino expuesta presenta un microcontrolador (70), mientras que la Raspberry Pi incorpora un microprocesador (71). Esto solo reafirma lo comentado, la Raspberry Pi es como un *miniordenador,* más potente que la placa Arduino, el cual es más comparable con un *autómata programable*.

Con esta peculiaridad conseguiríamos evitar la necesidad de un ordenador o un equipo externo para procesar los datos. Está claro que, si tuviéramos esa necesidad, una placa como la Raspberry Pi cobra más sentido y quizás merezca la pena invertir un poco más de dinero por ello.

Specifications

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- 2GB, 4GB or 8GB LPDDR4-3200 SDRAM (depending on model)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE
- Gigabit Ethernet
- 2 USB 3.0 ports; 2 USB 2.0 ports.
- Raspberry Pi standard 40 pin GPIO header (fully backwards compatible with previous boards)
- 2 × micro-HDMI ports (up to 4kp60 supported)
- 2-lane MIPI DSI display port
- 2-lane MIPI CSI camera port
- 4-pole stereo audio and composite video port
- H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
- OpenGL ES 3.0 graphics
- Micro-SD card slot for loading operating system and data storage
- 5V DC via USB-C connector (minimum 3A*)
- 5V DC via GPIO header (minimum 3A*)
- Power over Ethernet (PoE) enabled (requires separate PoE HAT)
- Operating temperature: 0 – 50 degrees C ambient

* A good quality 2.5A power supply can be used if downstream USB peripherals consume less than 500mA in total.

Figura 7.9. Especificaciones oficiales de la Raspberry Pi 4 modelo B.

Al ser un *miniordenador*, se pueden elaborar productos más completos utilizando diferentes lenguajes de programación como Python, C++ y Java. De hecho, la Raspberry Pi es utilizada en multitud de lugares con fines educativos y la cantidad de resultados que se puede obtener de ella dependen muchas veces de la imaginación.

En cuanto al soporte y evolución, no se queda atrás respecto a Arduino, ya que, de igual manera, presenta un foro propio de características similares (72), prueba de que esconde una gran comunidad y de que es un producto altamente fiable.

Claramente, estas ligeras ventajas se ven reflejadas en el precio si la comparamos con placas del estilo de Arduino. En concreto, la Raspberry Pi en su versión y modelo seleccionado, parte desde los 35$ (30€ aproximadamente), precio a partir del cual se irá sumando dinero en función de las características seleccionadas, como, por ejemplo, la memoria RAM (2GB, 4GB, 8GB).

Placas de desarrollo hardware. Alternativas.

Existen diferentes alternativas a ambas placas. Muchas de esas alternativas se nombran en las fuentes de información referenciadas al principio de este apartado. A pesar de existir suficientes alternativas, no presentan un panorama tan estable como lo presentan las dos ya comentadas. Comentaremos algún ejemplo de alternativa a Arduino y a Raspberry Pi, respectivamente.

Un ejemplo de alternativa a Arduino sería *Nanode*. Aunque las referencias encontradas a esta placa son escasas y algo anticuadas (sean oficiales o no), la poca información encontrada muestra como en su momento se presentó como la evolución y alternativa clara de Arduino. Teóricamente presenta la misma calidad y funcionalidad, incluyendo el mismo entorno de programación, el hecho de ser abierto, programable desde cualquier sistema operativo MAC, Windows o Linux…

Además de esto, permite conectarse a Internet a través de una API pudiendo utilizarse la placa como servidor, lo cual no permitía Arduino en su momento. Probablemente hoy Arduino disponga de algún *gadget* que le permita realizar esta función. Esto se intuye por la gran expansión típica del *mundo Arduino*. La verdad es que esta funcionalidad es bastante interesante, puesto que podría servirnos para publicar los datos de la estación meteorológica a través de la red si el sistema derivase en un módulo remoto, aunque es un tema que de momento no toca abordar.

La falta de bases firmes en la información recabada respecto a esta placa conforma la principal razón de su descarte y al mismo tiempo sirve para reflejar la estabilidad que ha demostrado tener Arduino a lo largo de estos años, pues sigue en pie y en constante renovación.

Toca hablar sobre el ejemplo de alternativa para Raspberry Pi, cuyo nombre es *BeagleBone/BeagleBoard*. En el caso de todas las alternativas vistas para la Raspberry Pi, muchas veces ocurre lo mismo que con las alternativas para Arduino: la imposibilidad de seguir el ritmo marcado en lo que se refiere a renovación, estabilidad y popularidad.

Aparte de eso, los precios de las alternativas a Raspberry Pi generalmente suelen ser bastante más caros. En gran medida esto es porque ofrecen prestaciones superiores, pero como se creen suficientes las prestaciones de la Raspberry Pi, es lógico que esta última pase a ser una opción más atractiva, ya que se está pagando por lo que se necesita.

Esta placa BeagleBone, al igual que la Raspberry Pi, es como un pequeño ordenador donde se pueden ejecutar sistemas operativos como Linux o Android. Al igual que la Raspberry Pi, presenta una mayor potencia de procesamiento respecto a Arduino. En esta alternativa observamos como el precio asciende a los 50€ o 60€ según el modelo que se elija. Como se venía comentando, por la naturaleza de nuestro proyecto, quizás no compensa ascender a estos precios por prestaciones similares.

Otra alternativa al uso de cualquier placa (y, por lo tanto, de sensores) podría ser el uso de un dispositivo móvil con Android incorporado. Esta es una alternativa cuanto menos interesante, pero está claro que al adquirir un *smartphone* para que actúe como estación meteorológica se estaría pagando por prestaciones que no se usarían, incluso si se opta por móviles de gama baja (elección obligatoria para evitar que el precio se dispare). Eso sin contar con que no todos los *smartphones* vienen equipados con los sensores que necesitaríamos.

Además, no sabemos si quiera si estos sensores acogen las especificaciones necesarias para la medición de las condiciones del ambiente ni si el dispositivo móvil brinda la posibilidad de obtener los datos *a conciencia* para un objetivo como el nuestro. Realmente son preguntas con sentido, pues en muchos casos estos sensores vienen orientados a mantener estable y controlado el dispositivo móvil. En conclusión, a pesar de que sería interesante contemplar esta posibilidad y usar incluso una aplicación desarrollada en Android para la recogida y transmisión de datos ambientales, queda descartada por motivos evidentes.

Sensores de Luz

Fotorresistencia o LDR. La fotorresistencia o LDR es una resistencia cuyo valor varía en función de la luz que incida sobre él, de forma que con unos cálculos sencillos se podría detectar en qué condiciones lumínicas nos encontramos (salida analógica). Podemos observar un ejemplo de fotorresistencia en la figura 7.10.

Lo bueno de esta opción es la sencillez y el precio, puesto que este tipo de fotorresistencias son bastante baratas y nunca superarían los 2€, salvo excepciones extraordinarias (73).

- **Sensor BH1750.** El sensor BH1750 es otro sensor de luz compatible con Arduino que a diferencia de los LDR nos ofrece una salida digital y un valor de salida en Lux (Lumen/m^2), es decir, el nivel exacto de iluminancia. Estos son dos argumentos bastante interesantes de cara a la elección de este sensor.

Figura 7.10. Ilustración genérica de una fotorresistencia.

Figura 7.11. Ilustración del sensor BH1750.

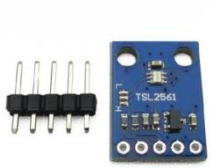


Figura 7.12. Ilustración del sensor TSL2561.

Este sensor, apreciable en la figura 7.11, es altamente preciso y configurable (74). A raiz de todo lo comentado, es una opción para considerar fuertemente debido a que parece ofrecer mejores resultados. Su precio ronda los 10€ en tiendas conocidas.

- **Sensor TSL2561.** El sensor TSL2561 es otro sensor de luz compatible tanto con Arduino como con Raspberry Pi. Podemos observar este sensor en la figura 7.12. De acuerdo con nuestras observaciones y lo expuesto con anterioridad, presenta algunas similitudes con el sensor BH1750. Se citan literalmente los detalles de este sensor a continuación (75):

  > *Sensor digital de luz de alto rango dinámico para Arduino o Raspberry Pi de Adafruit muy útil para una amplia variedad de aplicaciones.*
  >
  > *Las principales diferencias de este sensor con respecto a los basados en las LDR son su mayor precisión, permitiendo cálculos exactos de Lux entre 188 uLux a 88000 Lux, y que al incluir diodos infrarrojos y de espectro completo, permite realizar mediciones separadas de: Infrarrojos, amplio espectro y luz visible.*
  >
  > *Tiene ADC integrado por lo que se puede usar con cualquier microcontrolador, aunque este no tenga entradas analógicas.*

Sensores de Temperatura y Humedad

- **Sensor SHT7X.** El sensor SHT7X fue utilizado y mencionado en uno de los artículos comentados en el análisis (65), aunque al parecer ha sido reemplazado por un modelo renovado (76). Se muestra su apariencia en la figura 7.13.

  El sensor SHT7X es un sensor digital de tipo pin (presenta la ventaja de una fácil sustitución) que presenta dos variantes: una con buena relación calidad-precio (SHT71) y otra de gama alta (SHT75). Esto poco importa, ya que se comentaba que ha sido renovado (ambas variantes han sido sustituidas por otra: SHT85). Es un sensor que parece fiable y probado por diferentes usuarios de Arduino, por lo que el riesgo no sería tan elevado.

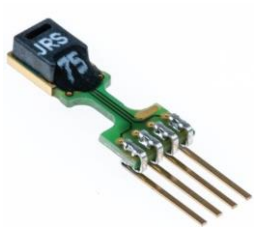


Figura 7.13. . Ilustración del sensor SHT7X.

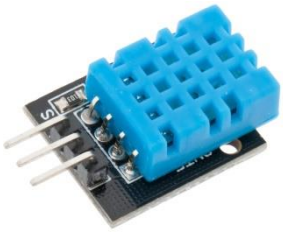


Figura 7.14. . Ilustración del sensor DHT11.

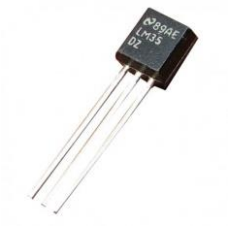


Figura 7.15. . Ilustración del sensor LM35.

- **Sensor DHT11.** El sensor DHT11 es un sensor de temperatura y humedad bastante interesante (77). Este sensor es digital, por lo que tendremos resistencia al ruido en las señales. En principio es un sensor compatible con Arduino y placas similares, presentando dos variantes sumamente parecidas (una con PCB y con LED de funcionamiento y otra sin estas prestaciones).

  Apunta a ser bastante conocido y usado por las comunidades de desarrollo de sistemas hardware. Tanto la disponibilidad como su bajo precio (5€ aproximadamente) presentan una buena baza para presentar a este sensor como una de las opciones más interesantes a considerar. Este sensor tiene una versión superior (DHT22) que presenta, entre otras características, una mayor precisión. Sin embargo, el DHT11 será nuestra opción inicial y se muestra en la figura 7.14.

- **Sensor LM35.** El sensor LM35 es un sensor con desventajas como el hecho de no ser digital (más ruido). Simplemente se expone su nombre y su información bibliográfica (78) como una vía de escape en caso de no encontrar las atractivas alternativas expuestas previamente. Se puede contemplar este último sensor de temperatura y humedad en la figura 7.15.

Sensores de Luz All-in-one

- **BlueDot BME280+TSL2591 Advanced Weather Station.** Constituye una opción interesante debido a que integra sensores de temperatura, humedad, presión, altitud e iluminación, todos bastante sensibles y precisos según las especificaciones y opiniones generales. Si tenemos curiosidad por saber cómo es este sensor, podemos observarlo con detenimiento en la figura 7.16.

  Todas las especificaciones las podemos observar en la página oficial (79). Si no se descubren evidencias claras de lo contrario, es compatible con Arduino y quizás los únicos puntos débiles en nuestro caso, serían el precio (elevado por el gran número de sensores de calidad que incorpora) y la cuestionable disponibilidad.

Conclusiones generales del diseño

Asumiendo que el diseño final contemplará una placa y diferentes sensores de apoyo para la obtención de datos, comencemos con las conclusiones relativas a las placas para luego seguir con las de los sensores.

Figura 7.16. Sensor all-in-one BlueDot BME280+TSL2591 que puede leer todas las variables requeridas.

En principio, los dos claros candidatos son Arduino y Raspberry Pi, descartando cualquier posible alternativa a ellos. Es cierto que en el mercado existen multitud de placas que quizás superan con creces a estas dos, pero muchas de ellas incrementan demasiado el precio para ofrecernos una perspectiva *más profesional y de mejor rendimiento* de la creación de *hardware personalizado*. Si bien pueden ofrecer precios competitivos, otras alternativas a estas dos populares placas carecen de un mantenimiento o renovación hardware tan positivo.

Estas placas candidatas quizás están orientadas a *labores menores* que la que se plantean en nuestro trabajo (educación, utilidades de hogar, juego…), pero debido a los requisitos planteados y la necesidad de minimizar el precio, se concluye que son las dos mejores opciones del mercado. Tampoco debemos olvidar que esta elección es ideal para el contexto del desarrollo de un TFM, donde se dan los primeros pasos y se aportan perspectivas de futuro ampliables.

Otro detalle importante en relación con esto último es que, si las variables a considerar varían o se decide sustituir el conjunto de sensores, con ambos (Arduino y Raspberry Pi) se tendría una gran flexibilidad al disponer de una amplia gama de sensores compatibles.

Además de las especificaciones y precios de las placas, la popularidad y opiniones de las personas animan más aún a esta elección y nos hacen pensar que son dos productos que no quedarán en el olvido y que dispondrán de cierto mantenimiento de calidad. De hecho, se han vuelto muy populares por su gran versatilidad en el ámbito del Internet de las cosas (IoT).

Entre la Raspberry Pi y la placa Arduino, la elección más sabia apunta a ser la placa Arduino, pues con un portátil disponible, que necesitamos de cualquier forma, no necesitamos ese *plus* de procesamiento que nos da la Raspberry Pi.

Seleccionada la placa Arduino, toca tomar una decisión respecto a los sensores. Disponemos de una amplia variedad de ellos, pero la decisión final irá marcada en función de la disponibilidad, el precio y las opiniones generales (precisión, durabilidad…) sobre cada uno de ellos.

Una conclusión clara es que merece la pena integrar los sensores de humedad y temperatura pues, al ser dos variables estrechamente relacionadas, es una ventaja disponer de dos sensores integrados en una misma pieza. De hecho, es una decisión tan evidente que todas las alternativas presentadas para estas variables cumplen esta

característica. Dentro de esta gama y por los parámetros comentados, el sensor candidato ha sido el sensor DHT11 (ofrece la posibilidad de escalar al DHT22, más caro, pero de mayor *calidad*).

De entre los sensores de luz, la opción más equilibrada y disponible parece ser el sensor BH1750. Lógicamente, en la decisión de todos y cada uno de estos sensores estamos algo limitados por la disponibilidad de estos, es decir, aunque un sensor sea mejor que otro o presente un mejor precio, la disponibilidad en tiendas cercanas ha sido clave para descartar o seleccionar cada uno de los sensores candidatos.

7.2.1.3. Montaje

Partiendo de la selección de componentes final resultado del anterior apartado de diseño, se expondrán a continuación los detalles de montaje. En el aspecto del montaje, se expondrá el diseño de manera gráfica y se procurará comentar con el mejor nivel de detalle, sin alargar innecesariamente este apartado.

Antes de comenzar a comentar, exponemos el diseño final elaborado en una de las herramientas ya comentadas en su momento, Fritzing. En este diseño exponemos dos posibles representaciones, con el objetivo de representar de forma más clara el diseño final. Estos dos diseños se pueden observar respectivamente en las figuras 7.17 y 7.18.

En el diseño digital ilustrado en estas figuras, es necesario recalcar que algunos componentes pueden no ser exactamente fieles al sensor real, como es el caso del BH1750 que presenta un pin adicional (cinco pines en total, en lugar de cuatro). Esto puede deberse a que los diseños de tales sensores pertenecen a fuentes externas, están aún en proceso de diseño o, sin irnos más lejos, representan versiones anteriores de estos mismos sensores. En cualquier caso, mientras se presenten todos los pines que se usan en nuestro diseño, estas peculiaridades no nos afectan en absoluto.

Ahora sí, toca comentar brevemente este simple pero eficaz diseño. En primer lugar, usamos la *Protoboard* para establecer la fuente de alimentación (VCC, 5V) y tierra (GND), de forma que a través de la *Protoboard* los sensores puedan acceder a ellos. Finalmente, se establecen las conexiones relativas a los datos para cada uno de los sensores conectados.

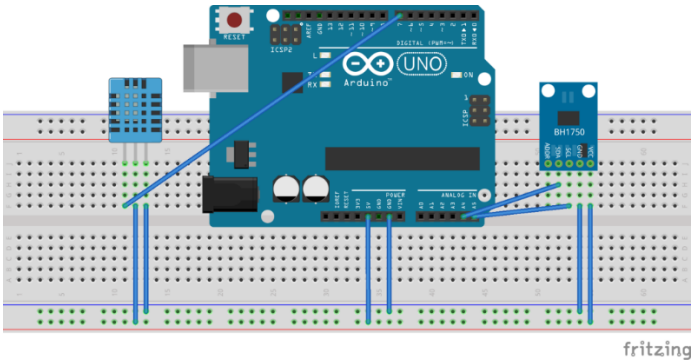


Figura 7.17. Diseño final ilustrativo y simulado de la estación meteorológica.

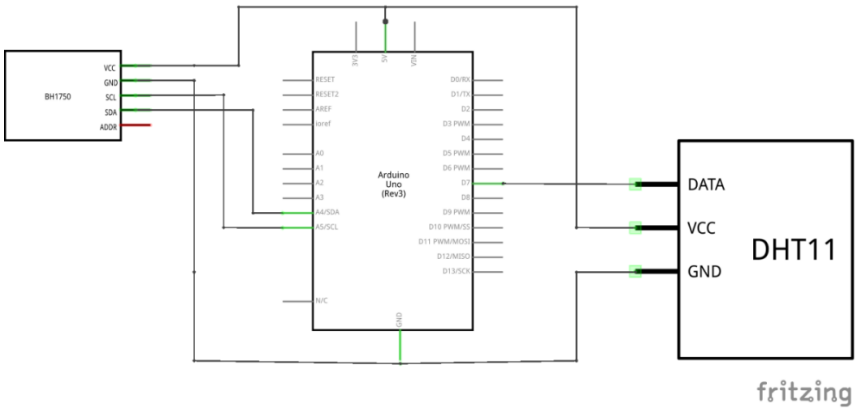


Figura 7.18. Diseño final esquemático del circuito montado para la estación meteorológica.

En el caso del sensor de temperatura y humedad DHT11, se conecta a uno de los pines digitales de Arduino (nuestra selección arbitraria fue el pin 7). En el caso del sensor de luz BH1750, se requieren dos conexiones a los pines analógicos de Arduino. Uno de estos corresponde a la señal de reloj (SCL) y el otro a la señal de datos (SDA). En Arduino, estos corresponden al pin A5 y A4, respectivamente. Sobra decir que no podemos realizar estas conexiones en otros pines analógicos.

Con el diseño montado, solo podemos mostrar una captura del sistema real para mostrar su aspecto en la realidad (figura 7.19). El uso de resistencias en nuestro diseño no fue necesario. En el caso del DHT11, por ejemplo, ya viene una resistencia *pull-up* (80) incorporada en el circuito. Para este sensor, existía una versión de cuatro pines en la cual era necesario el uso de una resistencia *pull-up* explícita (figura 7.20).

7.2.2. Software

En este apartado software de la estación meteorológica se comentarán los aspectos funcionales, organizativos y de implementación propios del desarrollo.

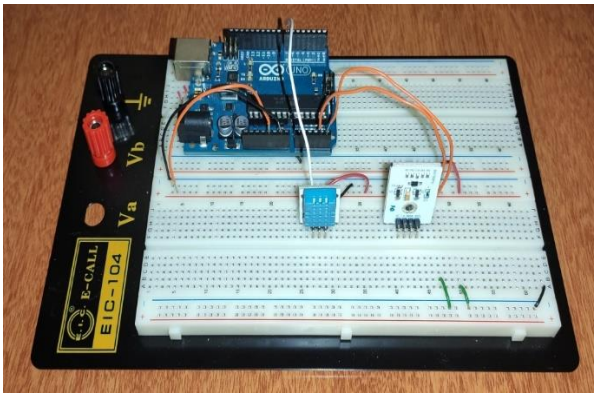


Figura 7.19. Diseño final de la estación para este TFM.

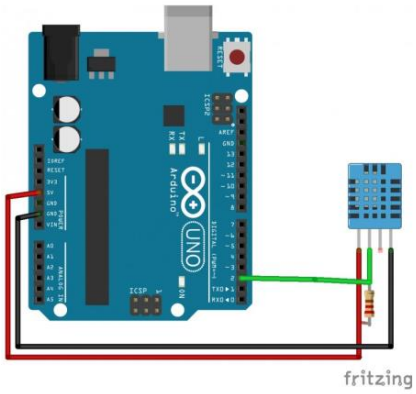


Figura 7.20. Sensor DHT11 de 4 pines con una resistencia pull-up.

De cara a una mejor organización de la memoria, se cree conveniente agrupar en el apartado funcional todo lo relativo a las iteraciones y a los requisitos (junto a otro tipo de documentación auxiliar relacionada como: diagramas de casos de uso, de actividades…). Por otra parte, se comentarán aspectos más *tangibles* en el apartado organizativo y de implementación, como pueden ser la arquitectura base del módulo (en función del diseño planteado) y los detalles del producto software. A todo esto, se le suma documentación auxiliar como diagramas de despliegue, de clases…

7.2.2.1. Funcional

Antes de proseguir a comentar las tres iteraciones de este módulo, se tiene que esclarecer que, a pesar de no usar una metodología conocida para la implementación de estos requisitos, se introducirá el tiempo aproximado consumido en la ejecución de cada iteración, aunque sea meramente ilustrativo.

Primera Iteración

Esta iteración, además de contemplar los requisitos básicos para elaborar un primer sistema funcional y entregable, constará de un proceso interno que se podría considerar desde un punto de vista abstracto, una *iteración cero* en la que se selecciona el lenguaje principal que se usará y se realizan algunas comprobaciones para verificar que el desarrollo puede llevarse a cabo con el lenguaje seleccionado (en función del diseño final del hardware). Todo esto, en definitiva, forma parte del conjunto de requisitos de esta iteración.

Los requisitos de esta iteración surgen a raíz de las necesidades más esenciales de este módulo y a raíz del acuerdo con el cliente (nosotros mismos y el grupo de investigación del IUCTC encargado del proyecto). Sin más dilación, las funcionalidades principales (requisitos) de la primera iteración se enumeran a continuación:

- Se debe implementar un sistema cliente-servidor que use el protocolo UDP como base de la comunicación para la correcta y futura integración del módulo en el sistema IPS final.
- El programa principal (servidor) y el software programado para la parte hardware (Arduino) deben ser capaces de comunicarse para enviar y recibir datos (en principio en formato JSON).
- El software programado para la parte hardware (Arduino) debe ser capaz de realizar las lecturas de todos los sensores existentes.
- El software programado para la parte hardware (Arduino) debe ser capaz de enviar los datos extraídos de la lectura de sensores al programa principal (servidor) en formato JSON.
- El programa principal cliente debe ser capaz de mandar una orden de conexión, que el programa principal servidor ejecutará para conectarse al puerto serie (COM) en el que se encuentra conectado nuestro sistema hardware.

- El programa principal cliente debe ser capaz de mandar una orden de desconexión, que el programa principal servidor ejecutará para desconectarse de nuestro sistema hardware.
- El programa principal cliente debe ser capaz de mandar una orden de lectura de variables meteorológicas, que el programa principal servidor ejecutará para obtener y formatear los datos solicitados (comunicándose para ello con el sistema hardware).

En resumen, el producto final de esta iteración debe ser un sistema cliente-servidor en el lenguaje elegido. En este sistema, el cliente empleará comandos concretos que el servidor deberá cumplir para realizar acciones como la conexión, desconexión y lectura de variables del entorno. Para esto, el servidor requiere comunicarse con la parte hardware (concretamente, con el software programado para ella).

Esta primera iteración, a pesar de presentar una lista algo extensa de requisitos, consumió aproximadamente dos semanas de trabajo intenso.

Segunda Iteración

Esta iteración va destinada a mejorar el producto obtenido en la anterior iteración. Para captar estos requisitos, se parte del mismo concepto de acuerdo con el grupo de investigación del IUCTC encargado del proyecto, tras haberse expuesto y mostrado el producto final de la primera iteración. A raíz del producto actual, los requisitos que se consideraron oportunos fueron los siguientes:

- El programa principal (servidor) debe ser capaz de transformar la humedad relativa (dato obtenido de un sensor) al formato de humedad absoluta.
- El programa principal (servidor) debe ser capaz de registrar, con su correspondiente marca de tiempo, todas las lecturas de datos en un fichero log personalizado.

En principio, el producto final de la primera iteración cumplía bastante las expectativas funcionales, siendo el resultado de esta iteración la implementación de estos dos requisitos.

Aunque a nivel programático parecen requisitos sencillos, esconden peculiaridades que alargaron la dedicación final a una semana aproximada de trabajo.

Tercera iteración (extraordinaria)

Aunque el sistema cumplía con todos los requisitos solicitados, se realizaron mejoras de optimización, adaptación y presentación (*code refactoring*) en el código elaborado. En parte, estas mejoras surgieron a posteriori, a raíz del desarrollo del otro módulo (módulo orientador). Es por ello por lo que consideramos está tercera iteración como extraordinaria. En esta iteración, los requisitos estipulados fueron los siguientes:

- El sistema debe leer los parámetros de configuración de un archivo de configuración.

- El usuario debe ser capaz de configurar el nombre asignado a cada comando.
- El usuario debe ser capaz de habilitar y deshabilitar el registro log para la lectura de valores ambientales.

En un cómputo global, el resultado final (ignorando cuestiones no funcionales) ha sido la incorporación de un nuevo archivo de configuración. Este archivo de configuración actúa ajustando los valores predefinidos del programa principal (tanto cliente como servidor) y permitiendo la personalización de ajustes de comportamiento del programa (a decisión del usuario). Ejemplo de lo primero puede ser la dirección IP y puerto usados para la conexión UDP, mientras que, en el segundo caso, un ejemplo es la posibilidad de habilitar/deshabilitar los logs.

Esta última iteración resulto en una dedicación aproximada de una semana de trabajo, considerando que muchas adaptaciones tomaban como referencia resultados del posterior módulo (sistema orientador), reduciendo el nivel de esfuerzo final necesario para esta iteración.

Diagrama de casos de uso

En la figura 7.21 se muestra el diagrama de casos de uso para este módulo meteorológico. Este diagrama representa de forma muy visual y resumida lo que el módulo proporciona al usuario, escondiendo todos los detalles técnicos del desarrollo.

Evidentemente, la parte funcional se refleja en la cara visible del programa, es decir, el cliente y las órdenes que se pueden ejecutar. Es por ello por lo que este diagrama de casos de uso omite la perspectiva del servidor y la comunicación con el sistema hardware. Los casos de uso principales a vista del usuario general son:

- *Conectar el Arduino*. Este es un comando necesario para que cualquier operación de lectura o escritura se pueda realizar sobre Arduino.

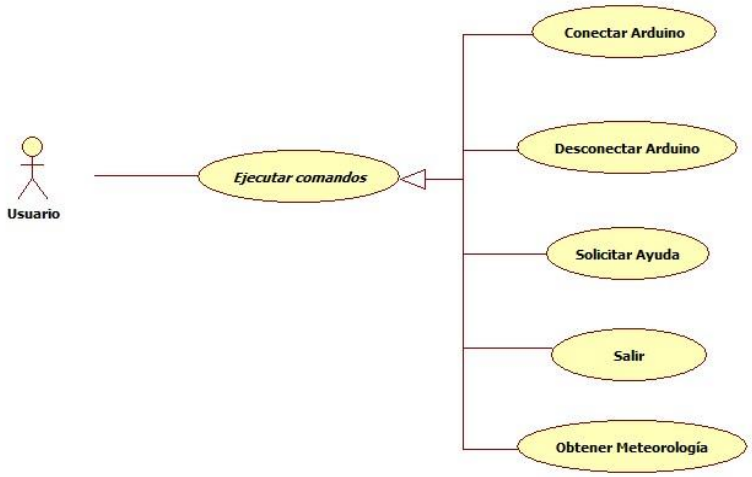


Figura 7.21. Diagrama de casos de uso de la estación meteorológica (cliente).

- *Desconectar el Arduino*. Es un comando sencillo simplemente para desconectar Arduino (programáticamente) cuando se finalicen las tareas estimadas o incluso si por causas extraordinarias surgieran errores de conexión/ejecución.
- *Obtener meteorología*. Este comando consiste en la lectura agrupada de todas las variables meteorológicas estipuladas para nuestra estación meteorológica.
- *Solicitar Ayuda*. Este comando permite al usuario solicitar la lista de comandos disponibles.
- *Salir*. Este comando permite cerrar el cliente y cerrar de forma transparente al usuario la entrada de peticiones (nuevos comandos) al servidor.

Hay que aclarar que el usuario puede ejecutar cada uno de estos comandos haciendo uso de nuestro programa cliente dedicado (exclusivamente a ello) o un programa cliente externo encargado de usar la API planteada en el lado servidor de nuestra aplicación. Un ejemplo de cliente externo podría ejecutar acciones más *personalizadas* como ejecutar una secuencia de comandos de forma programática.

Por ejemplo, podemos tener un programa encargado de conectarse, realizar lecturas programadas de las variables meteorológicas cada minuto durante una hora y finalmente desconectarse (que será algo parecido a lo que se tendrá cuando este módulo esté implementado en el IPS final).

Como complemento al diagrama de casos de uso, se dispone de la especificación completa para cada caso de uso en el anexo titulado *Especificación de casos de uso*.

Diagrama de actividades

Aunque hemos visto las funcionalidades que ofrece nuestra estación, es hora de plasmar en un diagrama de actividades el flujo de acciones que se ejecutan al interactuar con nuestro programa. Este diagrama de actividades se puede observar en la figura 7.22.

El diagrama de actividades propuesto es sencillo y minimalista, con el objetivo de plasmar la interacción del usuario con nuestro servicio. En este caso obviamos tareas de preparación obvias para no complicar innecesariamente el flujo general. Estas tareas obviadas son: *tener conectado físicamente el Arduino al ordenador donde se ejecuta el servidor* y *la ejecución del programa principal servidor*. Sin las tareas comentadas hace un momento, sería imposible ejecutar cualquier comando y, por tanto, hacer uso de nuestro producto.

Muy brevemente, el diagrama consiste en ejecutar el programa cliente, conectar nuestro Arduino programáticamente (si no lo está) y ejecutar sucesivas lecturas ambientales hasta que, cuando no se vayan a realizar más, se desconecte el Arduino.

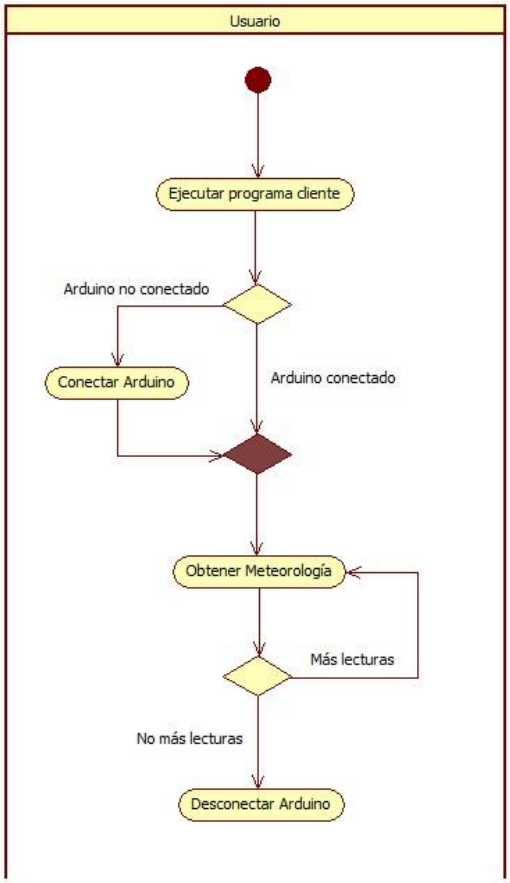


Figura 7.22. Diagrama de actividades de la estación meteorológica (perspectiva de usuario).

Hay que aclarar que se han omitido en este diagrama las acciones de solicitar ayuda y de salir con la intención de simplificar el flujo principal de acciones por parte del usuario al usar nuestro sistema. La primera acción (solicitar ayuda) tendría lugar en cualquier punto del diagrama en el que el usuario no tenga clara la lista de comandos, normalmente al principio o después de la ejecución de un comando. Por otro lado, la segunda acción (salir) tendría lugar al final del flujo, aunque a nivel de usuario es incluso recomendable no ejecutarla si no se tiene la certeza de que no se van a ejecutar más comandos, ya que implicaría volver a habilitar el servidor para ello.

7.2.2.2. Organizativa

Asumidas las funcionalidades y flujo de acción de la aplicación, es momento de abordar detalles más centrados en la organización estructural de nuestro proyecto, antes de entrar de lleno con toda la lógica en el aspecto de implementación. Para ello, comentaremos detalles acerca de los componentes que conforman nuestra estación meteorológica.

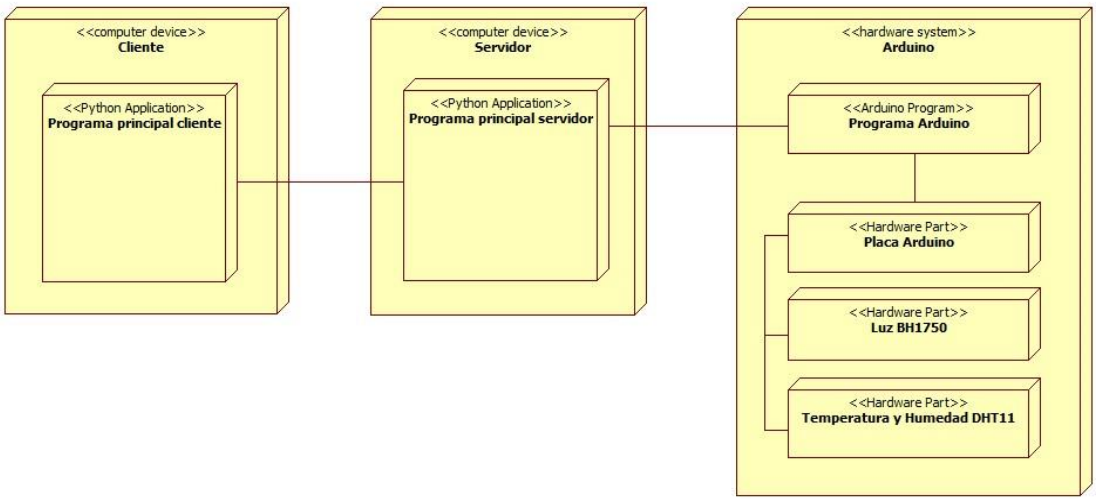


Figura 7.23. Se muestra el diagrama de despliegue básico de nuestro módulo orientador al completo.

Podemos observar dichos detalles en la figura 7.23, que muestra un diagrama de despliegue sencillo que comentaremos a continuación. En este diagrama se dispone de un equipo aislado que ejecuta el programa principal cliente y en otro plano tenemos el equipo que ejecuta el programa principal servidor. El servidor es el único encargado de comunicarse con el hardware diseñado, en definitiva, con nuestro Arduino.

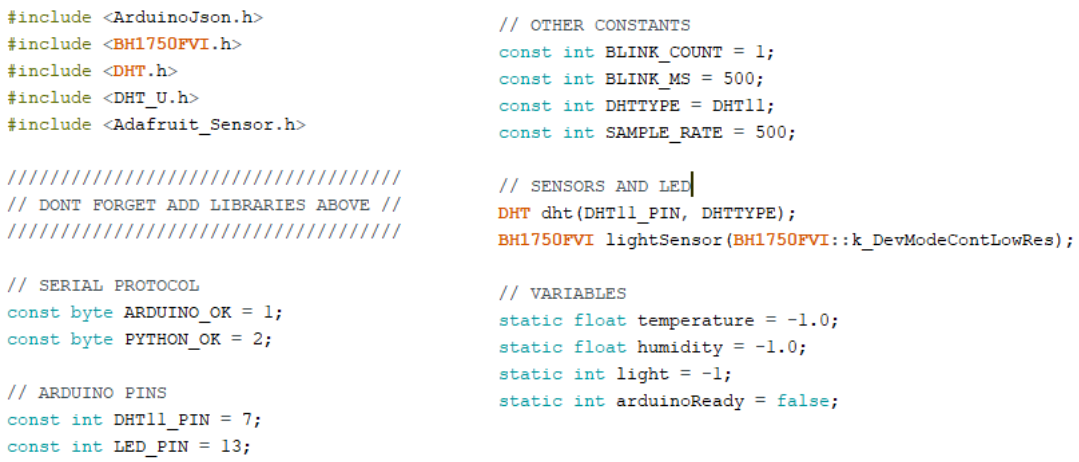
Se puede observar que, tanto para el programa principal cliente como para el servidor, el lenguaje utilizado es Python, por lo que estamos dando una pincelada del apartado que vendrá después de este, el de implementación.

Si bien el diagrama refleja a un alto nivel la arquitectura básica de nuestro módulo meteorológico, no hay que descartar un caso particular: *el equipo cliente y el equipo servidor son el mismo equipo*. Hemos plasmado equipo cliente y equipo servidor como equipos separados por el mero hecho de que nuestro sistema lo admite, pero evidentemente podemos ejecutar el servidor y el cliente principales en el mismo ordenador bajo *Localhost* (de hecho, se ha trabajado mayoritariamente de esta forma para agilizar el desarrollo).

No es una de nuestras tareas, pero cuando este módulo se integre dentro del cerebro principal del sistema IPS, los valores obtenidos (lectura de variables meteorológicas) se almacenarían en una base de datos.

7.2.2.3. Implementación

En el aspecto de implementación, entraremos a comentar principalmente el resultado software final, describiendo y explicando la estructura de clases definidas (diagrama de clases), algunos detalles estructurales empleados (patrones de diseño…) y aspectos explícitos del código que puedan resultar relevantes.



```
#include <ArduinoJson.h>
#include <BH1750FVI.h>
#include <DHT.h>
#include <DHT_U.h>
#include <Adafruit_Sensor.h>

////////////////////////////////////
// DONT FORGET ADD LIBRARIES ABOVE //
////////////////////////////////////

// SERIAL PROTOCOL
const byte ARDUINO_OK = 1;
const byte PYTHON_OK = 2;

// ARDUINO PINS
const int DHT11_PIN = 7;
const int LED_PIN = 13;

// OTHER CONSTANTS
const int BLINK_COUNT = 1;
const int BLINK_MS = 500;
const int DHTTYPE = DHT11;
const int SAMPLE_RATE = 500;

// SENSORS AND LED
DHT dht(DHT11_PIN, DHTTYPE);
BH1750FVI lightSensor(BH1750FVI::k_DevModeContLowRes);

// VARIABLES
static float temperature = -1.0;
static float humidity = -1.0;
static int light = -1;
static int arduinoReady = false;
```

Figura 7.24. Se muestra el diagrama de despliegue básico de nuestro módulo orientador al completo.

En este caso, como la parte hardware cuenta con su propia lógica de implementación (Arduino), se comentará ligeramente su estructura y su funcionamiento básico antes de comenzar con los programas principales (cliente-servidor). Para ello, se usarán capturas de código acompañado de diversas explicaciones.

Programa base en Arduino IDE

En el sencillo programa elaborado para Arduino, contamos con tres secciones de código fundamentales: establecimiento e inicialización de algunas variables y constantes, función `setup` y función `loop`. Estas dos funciones hacen uso a su vez de funciones auxiliares.

Comenzando con la primera de las secciones a comentar, podemos observar todo sobre ella en la figura 7.24. En primer lugar, vemos las *inclusiones* de librerías empleadas para el uso del formato JSON y el uso de los sensores de nuestro diseño. Hay que recalcar que, a pesar de estas líneas de código, es necesario importar las librerías desde el entorno de programación de Arduino.

Aparte de estas librerías, tenemos constantes de protocolo (donde especificamos los bytes que Arduino entenderá en el proceso de comunicación con el programa servidor), constantes de PIN (donde se especifica por ejemplo el PIN por defecto del LED nativo de Arduino y el usado por el sensor DHT11), variables que engloban la entidad de los sensores en nuestro programa (serán las que alberguen la inicialización de los sensores y las usadas para realizar lecturas) y diferentes variables que ajustan el comportamiento del programa.

Centrándonos en las funciones `setup` y `loop` que visualizamos en la figura 7.25, tenemos que la primera de ellas establece la tasa de transmisión de datos con el comando `Serial.begin(9600)` y espera hasta que el puerto serie esté conectado. Después, principalmente se limita a inicializar los sensores que se utilizarán y que conforman nuestro diseño físico. Tras eso, el estado actual de Arduino queda definido como *listo (true)*. Sin este requisito, Arduino no atenderá ninguna petición entrante.

```
void setup() {
  Serial.begin(9600);

  // Wait until serial port connected
  while(!Serial) { delay(SAMPLE_RATE); }

  // Set Pin 13 (LED) as an Output
  pinMode(LED_PIN, OUTPUT);

  // Initialize LightSensor and DHTSensor instance
  lightSensor.begin();
  dht.begin();

  // Arduino is now ready
  arduinoReady = true;
}
```

```
void loop() {
  // Time between checks
  delay(SAMPLE_RATE);

  // All loop tasks
  checkSerialInput();
}
```

Figura 7.25. Se expone la función principal de inicialización `setup` y la función `loop` características de cualquier programa Arduino.

```
/**
 * PROTOCOL COMMUNICATIONS
 */

void serialFlush(){
  arduinoReady = false;
  while(Serial.available() > 0) {
    char t = Serial.read();
  }
}

void sendReadyMessageToPython() {
  if (arduinoReady) {
    Serial.write(ARDUINO_OK);
  }
}
```

```
void checkSerialInput() {
  // No data received by Serial Port
  if (Serial.available() <= 0) {
    // ledBlink(BLINK_COUNT, BLINK_MS); // Blink slow
    sendReadyMessageToPython();
  }

  // Data received by Serial Port
  else {
    // ledBlink(BLINK_COUNT*2, BLINK_MS/5); // Blink faster
    if (Serial.read() == PYTHON_OK) {
      // ledBlink(BLINK_COUNT*2, BLINK_MS/10); // Blink fastest as pos
      serialFlush();
      sendDataToPython();
    }
  }
}
```

Figura 7.26. Conjunto de funciones auxiliares destinadas al protocolo de comunicación entre la placa Arduino con el programa principal servidor.

La segunda función se limita a comprobar el estado del puerto Serie para detectar solicitudes y actuar en consecuencia mandando los métodos. Todo esto ahora mismo está encapsulado en la función `checkSerialInput()`.

En la figura 7.26 podemos observar el comportamiento protocolario. Si no se están recibiendo datos por el puerto Serie, se manda la señal nombrada bajo ARDUINO_OK al programa principal servidor. Como podemos notar, esto jamás se llevará a cabo si la placa Arduino no está *lista*.

Si por el contrario se están recibiendo datos de algún tipo por el puerto Serie, se compara si la señal recibida es equivalente a la señal nombrada bajo PYTHON_OK. En caso de que no coincida, simplemente se pasa a otra iteración de la función `loop` (repitiendo este proceso). Sin embargo, si coincide, se limpia cualquier dato entrante (considerado ruido en el puerto de comunicación) y se empieza el proceso de lectura y respuesta de datos hacia el programa principal servidor, ya que quiere decir que el

programa servidor detectó nuestra disponibilidad y además ha realizado una solicitud de datos de los sensores. En este momento, Arduino deja de estar *listo*, pues atiende la actual solicitud.

Obviando detalles de implementación, el protocolo de comunicación puede comprenderse globalmente de la siguiente manera. En primer lugar, el Arduino se inicializa y marca su estado como *listo*. Indica al programa principal servidor que está listo mandando la señal correspondiente y queda a la espera de que el programa servidor vea el *semáforo en verde*. Cuando esto sucede y el programa servidor solicita los datos, le envía una señal al programa Arduino. El programa Arduino queda en estado ocupado mientras la resuelve y envía los datos. Finalmente, el programa servidor recibe los datos y se lo hace saber a Arduino, que brinda la posibilidad de realizar nuevas lecturas.

Este proceso final se ejecuta mediante las funciones de la figura 7.27, que en general se explica por si sola. Una vez ha realizado la lectura de los sensores y se encapsula y envía la respuesta, se espera por la respuesta de confirmación del programa principal servidor (la respuesta llegó). En ese momento, nuestro Arduino vuelve a estar listo para recibir más peticiones entrantes.

A lo largo de este código existen fragmentos comentados, la mayoría de ellos encapsulan el cambio de estado del LED nativo de Arduino en función del estado actual. Las funciones auxiliares para este proceso se plasman en la figura 7.28, aunque no son relevantes debido a que finalmente se ha decidido omitir su uso. La causa es que al ser un LED interno muchas veces presenta estados propios al leer/escribir del puerto Serie.

```
/**
 * READ AND RESPONSE RELATED FUNCTIONS
 */

void sendDataToPython() {
  readDHT11();
  readBH1750();
  response();
}

void readDHT11() {
  temperature = dht.readTemperature();
  humidity = dht.readHumidity();

  if (isnan(humidity) || isnan(temperature)) {
    // Failed to read from DHT sensor!
    temperature = -999.0;
    humidity = -999.0;
  }
}

void readBH1750() {
  light = lightSensor.GetLightIntensity();

  if (isnan(light)) {
    // Failed to read from BH1750 sensor!
    light = -999;
  }
}
```

```
void response() {
  // Create JSON object (document)
  const size_t capacity = JSON_OBJECT_SIZE(3);
  DynamicJsonDocument doc(capacity);

  // Set variables
  doc["temperature"] = temperature;
  doc["humidity"] = humidity;
  doc["light"] = light;

  // Print JSON in string format to be received in Python
  Serial.println(serializeJson(doc, Serial));

  //Wait until Python have received all data correctly...
  while (Serial.read() != PYTHON_OK) {}

  //...then arduino is ready to send more data
  arduinoReady = true;
}
```

Figura 7.27. Grupo de funciones encargadas de leer valores de los sensores y de formalizar la respuesta enviada al programa principal servidor.

```
/**
 * LED RELATED FUNCTIONS
 */

void ledOn() {
  digitalWrite(LED_PIN, HIGH);
}

void ledOff() {
  digitalWrite(LED_PIN, LOW);
}

void ledBlink(int count, int velocity) {
  for (int i = 0; i < count; i++) {
    ledOn();
    delay(velocity);
    ledOff();
    delay(velocity);
  }
}
```

Figura 7.28. Funciones auxiliares no demasiado relevantes que controlan el estado de los LED.

Programa principal (cliente-servidor)

Finalmente, le llegó el turno al programa principal elaborado en código Python. Esta característica no es un misterio, pues se pudo adelantar en apartados previos. Aunque Python es quizás un lenguaje orientado a la creación de *scripts*, no hay razón para renegar de él cuando se nos presenta la oportunidad de un desarrollo software. Realmente Python presenta muchas ventajas como el gran conjunto de librerías compatibles y la flexibilidad que ofrece, sin contar con que es un lenguaje usado en algún que otro aspecto del sistema IPS y a su vez bastante popular en la actualidad.

Realizada esta aclaración, pasamos a introducir el diagrama de clases que se trae entre manos el desarrollo completo de este módulo meteorológico. Se puede apreciar este diagrama en la figura 7.29, el cual iremos explicando parte por parte en los sucesivos párrafos. Aunque puede parecer difícil de asimilar, podemos agrupar las clases en diferentes categorías:

- *Clases de servicio*. Agrupamos bajo esta categoría a las clases `CommandClient`, `InteractiveClient`, `Server` y `RequestManager`. Las tres primeras son los archivos ejecutables de nuestra aplicación que ponen de manifiesto el servicio elaborado, mientras que la última de ellas es una clase encargada de encapsular el manejo de peticiones por parte del programa servidor.

  Hay que aclarar que las tres primeras realmente no son clases, sino *scripts* ejecutables que cuentan con funciones auxiliares. Sin embargo, a fin de no enturbiar el concepto estructural del diagrama, se han establecido como clases (de hecho, podrían serlo, pero a nivel programático carece de sentido).

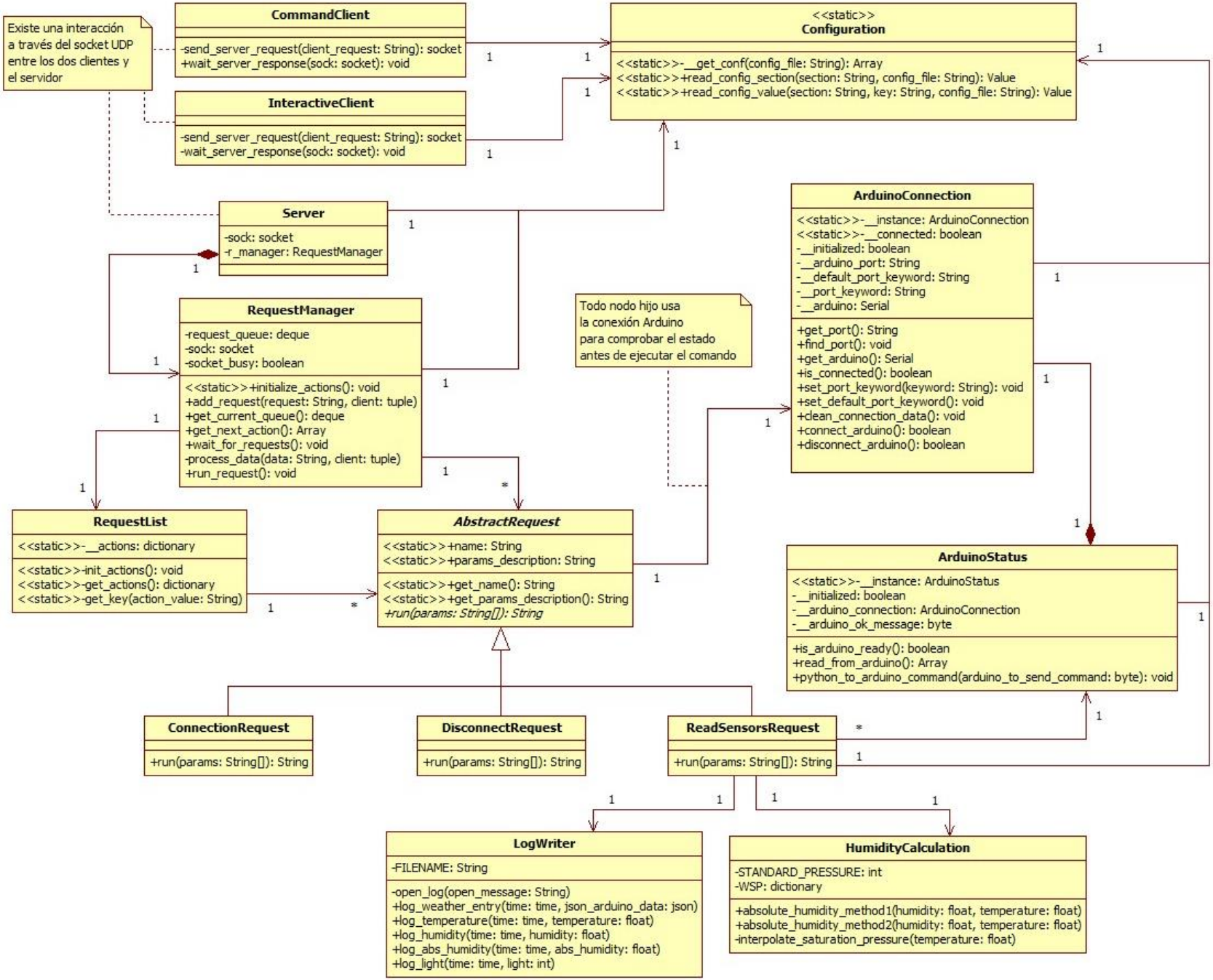


Figura 7.29. Diagrama de clases del programa principal del módulo de la estación meteorológica.

La diferencia básica entre el primer tipo de cliente y el segundo, es que el primero se ejecuta junto a unos parámetros (por ejemplo, ejecutando el comando de conexión: `CommandClient.py connect_command`) proporcionando un resultado y dando por concluida la ejecución, mientras que el segundo funciona de forma interactiva, es decir, se ejecuta sin parámetros y admite la interacción del usuario a modo de *Shell interactiva* (una vez ejecutado, el usuario es capaz de introducir los comandos, como por ejemplo el comando de conexión previamente visto).

Respecto al servidor, simplemente queda a la escucha de peticiones clientes entrantes y las va almacenando en una cola. Se añaden peticiones y se ejecutan de forma concurrente, ejecutando eso si un máximo de una sola petición en cada momento.

- *Clases de comandos*. Incluimos en esta lista a las clases `RequestList`, `AbstractRequest` y al resto de implementaciones de la clase `AbstractRequest`. Son clases que evidentemente tienen que ver con el comportamiento (aspecto funcional) de la aplicación.

  La primera de ellas, `RequestList`, simplemente inicializa y agrupa en una lista accesible al usuario la lista de comandos disponibles, incluyendo nombre del comando, invocación del comando y parámetros disponibles.

  La segunda clase es una clase abstracta que establece que todos los comandos deben tener una descripción accesible (nombre y parámetros) y un método *run* abstracto que cada comando se encargará de implementar para ejecutar correctamente la acción asignada. En principio, la ejecución de un comando devuelve una respuesta en formato *String* (de contenido arbitrario o regularizado en un JSON convertido en *String*), que será la que reciba el cliente.

- Clases *hardware*. En este grupo entran `ArduinoConnection` y `ArduinoStatus`. En definitiva, recogen toda la lógica encargada de la conexión y del estado de Arduino. Principalmente destacan la lectura del estado de Arduino (listo o no) y el envío de datos hacia este. Lógicamente, estas clases son fundamentales para lograr ejecutar todos los comandos propiamente.

- *Clases auxiliares.* Bajo esta categoría entran `LogWriter` y `HumidityCalculation`. Son clases que recogen una lógica encapsulada necesaria para el correcto funcionamiento de algún comando (o incluso otra parte de la aplicación), en este caso el comando encargado de realizar la lectura de los sensores.

La clase `LogWriter` agrupa las claves para almacenar los valores leídos en un fichero Log, mientras que la clase `HumidityCalculation` se encarga de ofrecer un tratamiento adicional para las lecturas de humedad, esto es, la posibilidad de obtener la humedad absoluta del ambiente a partir de la relativa y otros datos como la temperatura.

Además de estas dos clases, podríamos incluir en esta categoría la clase `Configuration`, ya que es la que permite leer valores de configuración que regulan la ejecución de todo el programa principal. Existen un archivo de configuración para los clientes y otro para el servidor, que es el que presenta una mayor carga de ajustes, evidentemente. Todos los ajustes se pueden ver mejor en el anexo con título *API*.

Con todo lo anterior queda más o menos claro el trabajo de cada clase en particular y, en consecuencia, un concepto general del funcionamiento de este módulo. De todos modos, entramos un poco más en detalle en cuanto al flujo general de ejecución, cubriendo así todos los aspectos relevantes que se pueden comentar en cuanto al desarrollo.

En resumen, nuestra clase/*script* servidor (*Server)* se ejecuta y queda a la espera de peticiones entrantes (un hilo escucha y añade las peticiones mientras otro las va ejecutando una a una). Ejecutamos también uno de los clientes, encargado de captar y enviar las peticiones (comandos) del usuario al servidor.

Es el servidor el que haciendo uso de diferentes valores de configuración y de la definición de órdenes (plasmada en `RequestList`), ejecuta el comando correspondiente. Ejecutar el comando al fin y al cabo consiste en ejecutar el método *run* definido para cada clase que hereda de `AbstractRequest`. Cada comando puede o no usar clases auxiliares, como en el caso de `ReadSensorsRequest`, que requiere usar `LogWriter` y `HumidityCalculation`. El uso de la clase `ArduinoStatus` por parte de este comando es fundamental para lograr comunicarse con Arduino y obtener el estado particular de los sensores (estado del ambiente).

Antes de abandonar este apartado y este módulo meteorológico, dejar una anotación final. En el desarrollo se han empleado algunos patrones de diseño como el *Singleton* (para la definición de algunas clases) y también el *Polimorfismo*, poniendo de manifiesto algunas prácticas relacionadas con el *clean code*. Se ha decidido reservar esto a un apartado dentro del Anexo con nombre *Patrones de diseño y clean code.*

7.3. Módulo B. Sistema Orientador

Dentro del posicionamiento en interiores, este módulo es igualmente importante que la estación meteorológica. Debido a que la orientación en la que se reciben las señales Bluetooth determina la naturaleza de la señal recibida, es lógico plantear esta la necesidad de este módulo en nuestro IPS.

Este módulo nace para cubrir las necesidades relacionadas con la orientación del receptor de nuestro IPS (control y monitorización) y se expondrán detalles sobre ello a lo largo de los diferentes apartados que siguen a continuación.

7.3.1. Hardware

En la parte hardware de este módulo orientador no existen demasiados aspectos de desarrollo a comentar. No obstante, se comentarán algunos aspectos que se han tenido

en cuenta a la hora de familiarizarnos con él y al realizar el montaje oportuno.

A grandes rasgos, el sistema que se tiene entre manos es un dispositivo capaz de rotar un eje incorporado. Lo que estará protagonizando el extremo de este eje rotatorio (con evidentes adaptaciones en la faceta de encapsulamiento final en el IPS) será un dispositivo de recepción de señales Bluetooth.

Como probablemente se haya adelantado en otros apartados de la memoria, el protagonista del diseño del orientador es un motor Dynamixel, que es exactamente el mismo que se incorporó en la plataforma robótica del IPS.

Componentes

Debido a que no se requiere un análisis de requisitos en cuanto a componentes (como en el caso del módulo anterior), se desglosan los resultados finales del diseño. Concretando un poco más, contaríamos con los siguientes componentes:

- Motor Dynamixel. Concretamente el modelo empleado ha sido el XM430-W350 (81), detalle relevante de cara a las especificaciones. Aunque no nos detendremos demasiado en este aspecto, en cuanto a especificaciones debemos destacar la precisión de este motor, que es de 0.088 grados. Esta, junto a su buena relación calidad-precio y su fuerza, constituyen las principales causas de su adquisición y su uso en el proyecto IPS. Podemos ver una imagen de este modelo en la figura 7.30. El precio de este motor Dynamixel queda en 213.67€.
- U2D2 Power Hub Board (82). Nos permite suministrar de energía eléctrica al motor Dynamixel y, junto a la interfaz U2D2 (83), conectarlo al ordenador para poder manejarlo programáticamente. Podemos apreciar esta placa y el controlador U2D2 (dentro de su contexto de funcionamiento) en las figuras 7.31 y 7.32, respectivamente. Los precios de estos dos componentes quedan en 32.47€ y 44.09€, también de forma respectiva.

El conjunto de componentes presenta un precio total de 290.23€. Estos componentes vienen de forma individual, por lo que se ejecutó una labor de montaje siguiendo principalmente las instrucciones de la página oficial en el apartado *how to assemble* (82). Una vez logrado el montaje, queda algo similar al contenido de la figura 7.33. Los pasos para conectar debidamente y sin ningún tipo de error deberían ser los siguientes:

Figura 7.30. Imagen oficial del motor Dynamixel, modelo XM430-W350.

- Se conecta el cable USB-MicroUSB (ordenador e interfaz U2D2).
- Se conecta la fuente de alimentación.
- Se acciona el interruptor para activar el suministro eléctrico. Se podrá visualizar como se enciende una luz en el motor.

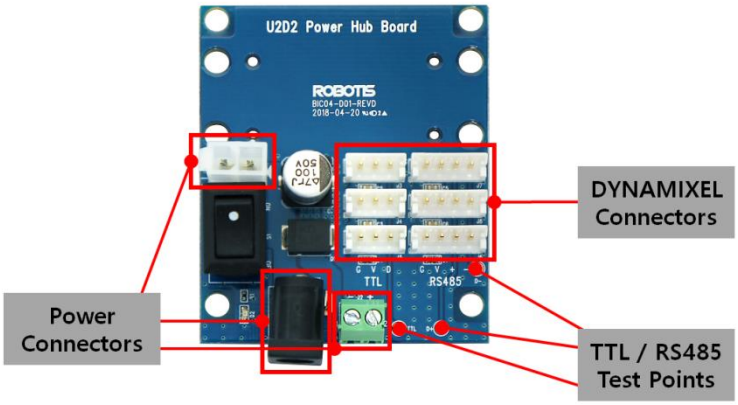


Figura 7.31. Aspecto y estructura básica del componente U2D2 Power Hub Board al que va conectado el controlador U2D2 y el motor Dynamixel.

Figura 7.32. Aspecto y contexto de funcionamiento de la interfaz U2D2.

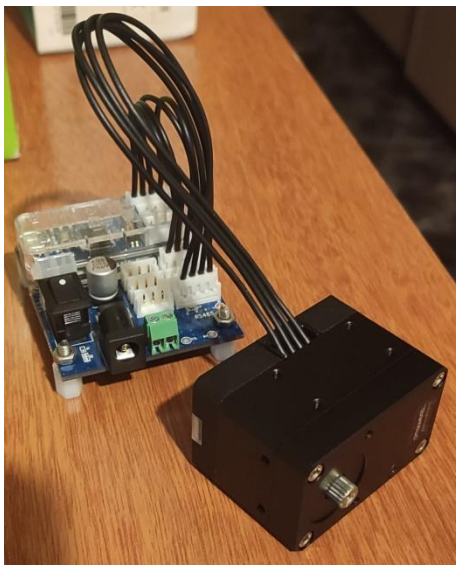


Figura 7.33. Diseño funcional del orientador

Antes de nuestro primer uso, es necesario configurar algunos aspectos del motor haciendo uso de su programa oficial, mostrado previamente en el apartado de *Herramientas utilizadas*. El principal ajuste es cambiar su ID para garantizar que estamos comunicándonos con este motor en concreto (se escogió el ID 9). Al margen de esto, nuestro hardware estará listo para usarse.

7.3.2. Software

De igual manera al módulo meteorológico, en este apartado software del módulo orientador se comentarán los aspectos funcionales, organizativos y de implementación propios del desarrollo.

Igualmente se presentarán dos vertientes relativas al producto software, por un lado, un apartado funcional que integre los requisitos de cada iteración y, por otro lado, un apartado más ligado a la organización y la implementación, es decir, al producto software desarrollado. En ambos casos, se aportará documentación auxiliar, que viene a ser la misma que la expuesta con anterioridad para los apartados equivalentes en el módulo meteorológico.

7.3.2.1. Funcional

Al igual que en el módulo meteorológico, se comentarán principalmente los requisitos contemplados en cada iteración y el tiempo de realización estimado con objetivos meramente ilustrativos.

Primera Iteración

Esta iteración es la base para obtener un primer producto entregable y verificar que el lenguaje seleccionado es apto para el desarrollo. En este caso, la parte de verificación del lenguaje se ve un poco más clara, ya que la especificación oficial del motor incluye a Python como un lenguaje compatible (existen librerías Dynamixel para este lenguaje) y se viene de una retroalimentación positiva del anterior módulo.

Los requisitos esenciales de esta primera iteración y que surgen a raíz del consenso con nuestro cliente (nosotros y el grupo de investigación del IUCTC encargado del proyecto) son los siguientes:

- Se debe implementar un sistema cliente-servidor que use el protocolo UDP como base de la comunicación para la correcta y futura integración del módulo en el sistema IPS final.
- El programa principal cliente debe ser capaz de mandar una orden de conexión, que el programa principal servidor ejecutará para conectarse al puerto serie (COM) en el que se encuentra conectado el motor Dynamixel.
- El programa principal cliente debe ser capaz de mandar una orden de desconexión, que el programa principal servidor ejecutará para desconectarse del Dynamixel.

- El programa principal cliente debe ser capaz de mandar una orden de rotación del motor, que el programa principal servidor ejecutará para rotar el motor a una posición concreta en grados (especificada en la petición cliente).
- El programa principal cliente debe ser capaz de mandar una orden de rotación del motor al grado cero del motor (origen), que el programa principal servidor ejecutará para rotar el motor a esa posición.

En resumen, el producto resultante de esta iteración debe formar un sistema cliente-servidor similar al expuesto en el otro módulo (de hecho, existen requisitos comunes en esta primera iteración). El cliente debe ser capaz de ejecutar comandos básicos como la conexión, desconexión o rotación del motor (parámetro en grados proporcionado por el propio cliente). Estos comandos deben ser ejecutados por el servidor haciendo uso de las herramientas de comunicación con el motor Dynamixel.

Esta primera iteración consumió aproximadamente cuatro semanas de trabajo. Esto se debe a que la primera iteración, aparte de los requisitos más *funcionales*, afronta la creación de la infraestructura básica, así como el primer contacto con el hardware, que nunca se había usado.

Segunda Iteración

Con el producto funcional requerido por la primera iteración expuesto a nuestro cliente, se extendió la lista de requisitos que quedaría como trabajo para esta segunda iteración. Estos nuevos requisitos son básicamente los expuestos a continuación:

- El programa principal servidor debe consultar y establecer todos los parámetros de configuración en un archivo accesible al usuario.
- El programa principal servidor debe ser capaz de rotar el motor a una posición especificada en la petición (de rotación) cliente donde el valor en grados contiene números decimales.
- El programa servidor debe ser capaz de encolar las peticiones entrantes y ejecutar las peticiones una a una. Ambos procesos deben de ser concurrentes.
- El programa principal cliente debe ser capaz de mandar una orden de consulta de posición del motor, que el programa principal servidor ejecutará para devolver la posición actual de este al cliente.
- El programa principal cliente debe ser capaz de mandar una orden de calibración automática del motor, que el programa principal servidor ejecutará para iniciar un proceso de calibración y establecer el nuevo origen relativo (cero grados).
- El programa principal cliente debe ser capaz de mandar una orden de establecimiento manual del valor de calibración del motor, que el programa principal servidor ejecutará para guardar el valor numérico pasado en la petición como el nuevo origen relativo (cero grados).

Es necesario realizar algunas aclaraciones sobre estos requisitos. El requisito de admitir rotaciones con números decimales es evidente por la alta precisión que nos aporta el

motor. Un caso de uso de este requisito es necesitar dividir los 360 grados de posicionamiento en más de 8 posiciones equidistantes, en cuyo caso necesitaríamos esta especificación decimal.

Adicionalmente a esto, el requisito que plantea la necesidad de tener una cola de peticiones se podría haber contemplado de otra manera, como cancelar la orden en curso para admitir otra. En cualquier caso, no se consideró que esta fuese la mejor solución para nuestro caso particular (decisión conjunta).

En cuanto a la necesidad de calibración, es bastante sencillo. Si el motor estuviera posicionado de tal forma que sus cero grados por defecto no coincidieran con nuestros cero grados relativos (en este caso, los cero grados de la plataforma robótica) es necesario redefinir estos *cero grados*.

Con estos requisitos satisfechos, la actual iteración del producto quedaría resumidamente como un sistema más configurable a nivel de usuario (archivo de configuración), más preciso (posición en grados más precisa con la incorporación de decimales) y con más funcionalidades como lo son la obtención de la posición actual en grados y la posibilidad de calibración del motor. Esta última funcionalidad incorpora un nuevo concepto al sistema orientador, que es la posición relativa.

El trabajo requerido en esta segunda iteración exigía algunos cambios importantes en el código y parte de la estructura de este, derivando en tres semanas de trabajo, aproximadamente.

Tercera iteración

Esta tercera iteración orienta sus objetivos a mejorar considerablemente algunos aspectos funcionales y de configuración, perfilando el funcionamiento y uso del orientador al escenario donde se incorporará, es decir, al IPS final. Los requisitos de está iteración fueron los siguientes:

- El programa principal cliente debe consultar y establecer todos los parámetros de configuración en un archivo accesible al usuario.
- El programa principal, tanto servidor como cliente, deben agrupar la dirección IP y puerto del servidor (valores comunes) en su archivo de configuración para permitir un fácil reajuste y admitir su ejecución desde dos equipos diferentes.
- El usuario debe poder ejecutar las órdenes desde un cliente alternativo al cliente interactivo, es decir, desde un cliente que se ejecuta desde la línea de comandos con la orden deseada como parámetro (finalizando su ejecución después de ello).
- El programa principal cliente debe ser capaz de mandar una orden de rotación aleatoria del motor, que el programa principal servidor ejecutará para rotar el motor a una posición aleatoria en grados el número de veces especificado por la petición cliente.
- El usuario debe poder decidir, mediante un valor en el archivo de configuración, si las rotaciones necesarias para ejecutar la orden se realizan recorriendo el

menor recorrido posible (óptimas) o dentro del rango de los 0 y los 360 grados relativos (no óptimas).

- El proceso de calibración automático ejecutado por el programa principal servidor no debe terminar a no ser que el usuario lo indique explícitamente.
- El programa principal cliente debe ser capaz de mandar una orden de calibración manual del motor, que el programa principal servidor ejecutará para iniciar un proceso de calibración más natural y libre donde se puede incrementar y decrementar la posición del motor a voluntad.
- Cualquier proceso de calibración llevado a cabo por el programa servidor a petición del cliente debe ejecutar el calibrado dentro del rango de los 0 y los 360 grados absolutos.

Aparte de los requisitos expuestos, se destinaron esfuerzos a resolver algunos errores arraigados a la estructura planteada en el código, no solo solventando estos errores sino mejorando el funcionamiento y eficiencia generales. Esto implicó que se añadieran conceptos como el de *Semáforo* para evitar que en mismo instante se intentase leer/escribir el motor varias veces.

Esta constituiría la última iteración para nuestro proyecto y tomaría aproximadamente cuatro semanas de trabajo. Aunque algunos de los requisitos son evidentes de visualizar, no lo fueron tanto al afrontarlos en la fase de programación.

Diagrama de casos de uso

En la figura 7.34 se muestra el diagrama de casos de uso para este módulo orientador. Aunque se esconden los detalles técnicos, representa de forma compacta lo que nuestro sistema proporciona al usuario.

Este diagrama recoge principalmente las órdenes que el cliente puede ejecutar, obviando, al igual que en el módulo meteorológico, los detalles que esconde el servidor y su comunicación con el hardware (motor). Los casos de uso que podemos observar en él son los siguientes:

- *Conectar el motor.* Este es un comando requerido para poder ejecutar cualquier operación sobre el motor Dynamixel.
- *Desconectar el motor.* Este es un comando simple para desconectar a nivel de programa el motor Dynamixel y que se usa principalmente cuando no se va a realizar ninguna operación más.
- *Solicitar Ayuda*. Es exactamente el mismo caso de uso que el *Solicitar Ayuda* visto en el módulo meteorológico, pues permite al usuario solicitar la lista de comandos disponibles.
- *Salir*. Es exactamente el mismo caso de uso que el *Solicitar Ayuda* visto en el módulo meteorológico, ya que permite cerrar el cliente y cerrar la entrada de peticiones (nuevos comandos) al servidor.

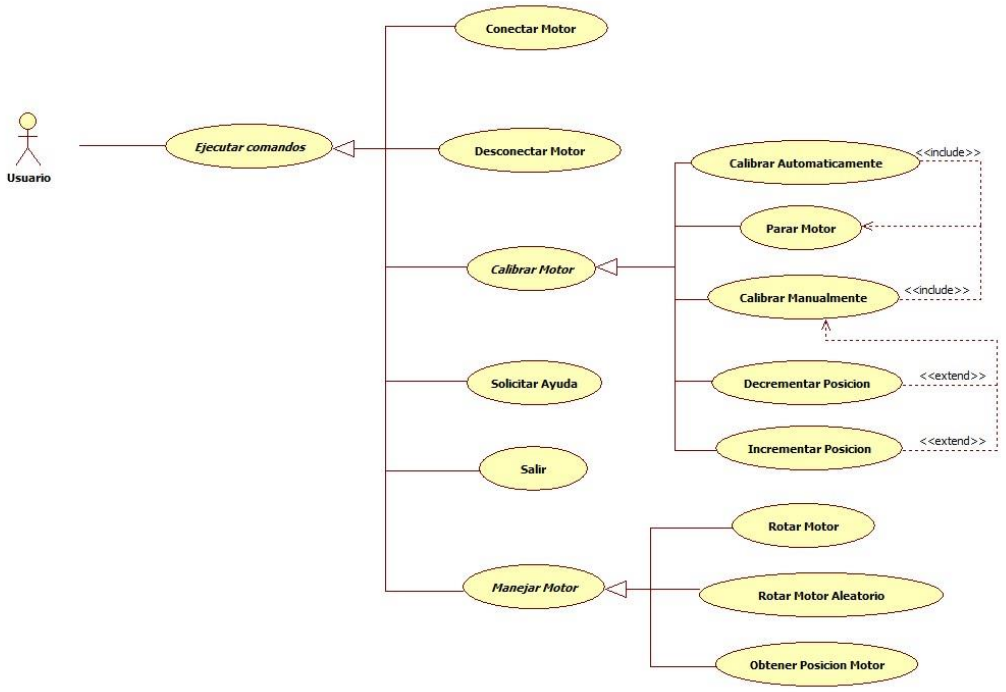


Figura 7.34. Diagramas de casos de uso del módulo orientador.

- *Calibrar el motor.* Este es un comando abstracto que consiste en establecer una posición origen (cero grados) relativa a petición del usuario. Este caso de uso es necesario cuando el origen del motor (sus cero grados absolutos) no coincide con el origen de la plataforma de nuestro IPS (los cero grados deseados, relativos). Este caso de uso abstracto engloba dos variantes:
    - *Calibrar Automáticamente.* En este calibrado automático el sistema va proponiendo valores aproximados para llegar a un valor final de calibración deseado por el usuario. Para ello, el usuario debe actuar parando el motor (*Parar motor*).
    - *Calibrar Manualmente*. En este calibrado manual el usuario maneja a su antojo (bajo ciertos límites) la posición del motor para establecer el valor final de calibración. Para ello, el usuario debe rotar el motor (*Incrementar Posición, Decrementar Posición*) y parar el motor cuando haya finalizado (*Parar Motor*).
- *Manejar el motor.* Este comando abstracto agrupa tres acciones diferentes que puede ejecutar el usuario:
    - *Rotar Motor*. Este comando hace girar el motor a una posición concreta en grados especificada por el usuario.
    - *Rotar Motor Aleatorio.* Este comando hace girar el motor a una posición aleatoria (un número de veces variable y especificado por el usuario).
    - *Obtener Posición Motor.* Este comando permite al usuario obtener la

posición actual del motor en diferentes formatos de salida, básicamente en grados y en tacómetros del motor.

El usuario puede ejecutar estas acciones mediante nuestro programa cliente o con un programa cliente personalizado que use la API del lado servidor. El cliente personalizado podría realizar acciones más concretas como ejecutar diferentes rotaciones de forma programada y secuencial (con sus debidas esperas).

En el caso particular del orientador, ese cliente personalizado debe tener muy en cuenta no solo los comandos y parámetros que puede enviar al servidor, sino el comportamiento y estado de este. A diferencia de la estación meteorológica donde la única acción básica era la lectura ambiental, en el orientador existen muchas alternativas, no todas ejecutables en todo momento. Un ejemplo rápido y sencillo es la opción de manejar el motor a voluntad, solo disponible cuando el motor está en proceso de calibración manual.

Dicho esto, hay que comentar que los detalles que se pierden en el diagrama de casos de uso expuesto se pueden complementar con la especificación de casos de uso presentada para ellos en el anexo titulado *Especificación de casos de uso*, al igual que ocurría con el otro módulo.

Diagrama de actividades

Para representar de forma dinámica las funcionalidades del orientador previamente descritas, se presenta un diagrama de actividades donde se muestra el flujo de acciones básicas a alto nivel. Este diagrama está disponible en la figura 7.35 y presenta similitudes respecto al diagrama de actividades del anterior módulo.

Básicamente el flujo consiste en ejecutar el cliente, conectar el motor si no lo está, ejecutar la calibración si no se ha realizado en usos anteriores del sistema (o si la arquitectura de este cambió) y finalmente, realizar todas las operaciones de manejo deseadas hasta desconectar el motor cuando se haya terminado de usar. Naturalmente, existen dos opciones de calibración (automática y manual) y tres opciones de manejo del motor (obtener posición, rotar y rotar a posición aleatoria), expuestas todas en la definición de casos de uso previa.

En este caso también se ignoran tareas vinculadas a la parte del servidor (preparación física y ejecución del programa servidor) y se omiten las acciones de *solicitar ayuda* y de *salir* con la intención de simplificar el flujo principal de acciones por parte del usuario. Las razones y causas de todo esto son exactamente las mismas que las planteadas en el módulo anterior, por lo que no se entra en detalles redundantes.

7.3.2.2. Organizativa

Llegó la hora de abordar el aspecto organizativo y estructural de este módulo orientador y para ello centraremos la atención en los componentes que lo forman. La información sobre esto mismo se puede apreciar en la figura 7.36, donde se expone un diagrama de despliegue sencillo que abordaremos en los siguientes párrafos.

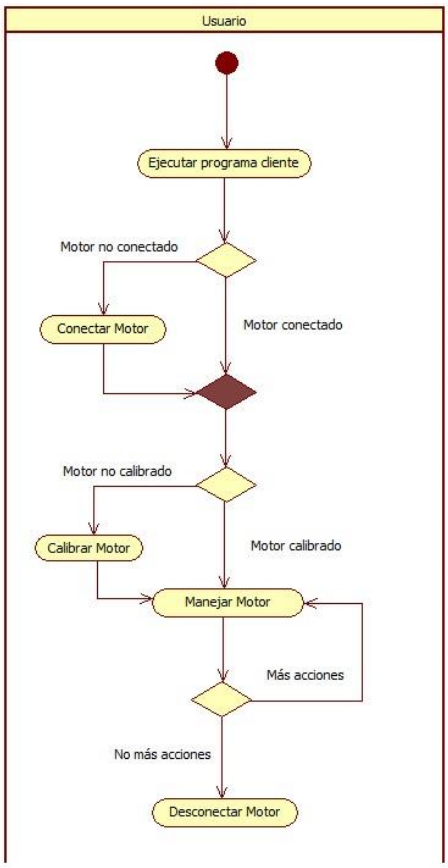


Figura 7.35. Diagrama de actividades con el flujo de acciones globales para el módulo orientador.

Disponemos de un equipo aislado que ejecuta el programa principal cliente y de otro equipo que ejecuta el programa principal servidor (encargado de comunicarse con el motor Dynamixel). Como ya se sabe y también se verá en el siguiente apartado de *Implementación*, ambos programas están elaborados en Python.

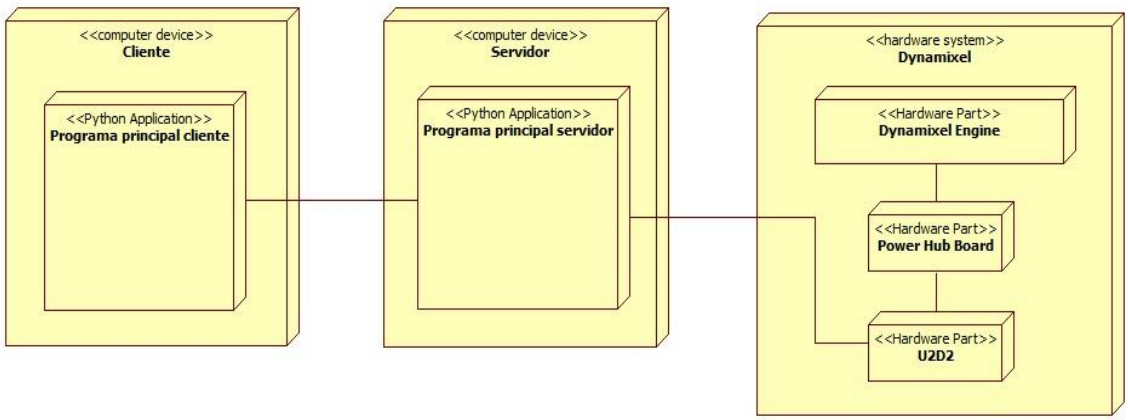


Figura 7.36. Diagrama de despliegue con los componentes que conforman el módulo orientador.

Un caso particular ya detectado en el módulo meteorológico es la posibilidad de que ambos programas se ejecuten en la misma máquina, escenario habitual durante el desarrollo. No entramos en más detalles sobre ello, ya que la situación es exactamente la misma que se planteó en su momento.

Otro aspecto en común con el módulo meteorológico es el concepto de la base de datos. Los datos de la orientación se guardarán en la base de datos *global* del IPS cuando este módulo se incorpore en el mismo.

7.3.2.3. Implementación

En cuanto a implementación se refiere, como se adelantó en ocasiones anteriores, se comentará el resultado software final. Principalmente nos centraremos en la estructura de clases (atendiendo al correspondiente diagrama de clases) y en detalles que se consideran relevantes, relacionados por ejemplo con el código resultante. En este caso, solo debemos detenernos en el programa principal (cliente-servidor), ya que la parte hardware no corre a nuestro cargo, es decir, no se requiere la incorporación de un programa propio y desarrollado como lo requirió la placa Arduino.

Programa principal (cliente-servidor)

El programa principal se elaboró en Python, como a estas alturas de la memoria ya es evidente. Esto pudo ser posible gracias a que Dynamixel presenta librerías de control de sus motores (en concreto nuestro modelo XM430-W350) para este lenguaje. Sinceramente, esto juega a nuestro favor, puesto que se han podido elaborar ambos módulos en el mismo lenguaje de programación, favoreciendo la coherencia entre el desarrollo de ambos módulos y, por tanto, de este mismo documento.

Sin más demora, pasamos a introducir el diagrama de clases perteneciente al módulo orientador. Este diagrama lo podemos ver con más detenimiento en la figura 7.37, aunque se irá desglosando parte por parte para comprender su estructura y el ámbito de actuación de cada una de las clases. Si se comprendió el diagrama de clases del módulo meteorológico, este será algo más fácil de digerir, puesto que comparten similitudes estructurales. A continuación, agruparemos las clases en diferentes conjuntos en función de su misión o contexto de actuación. Esos grupos o categorías son los siguientes (idénticos al módulo meteorológico):

- *Clases de servicio*. Agrupamos bajo esta categoría a las clases `CommandClient`, `InteractiveClient`, `Server` y `RequestManager`. Las tres primeras son los archivos ejecutables de nuestra aplicación que ponen de manifiesto el servicio elaborado, mientras que la última de ellas es una clase encargada de encapsular el manejo de peticiones por parte del programa servidor.

  Como esta parte ya se ha comentado, no se quiere entrar en lo redundante. Se pueden resumir en un servidor que atiende, encola y ejecuta peticiones (usando una clase auxiliar) y dos variantes de clientes encargados de enviar las peticiones (comandos) al servidor.

- *Clases de comandos*. Incluimos en esta lista a las clases `RequestList`, `AbstractRequest` y al resto de implementaciones de la clase `AbstractRequest`. Todas estas clases engloban el comportamiento del sistema y se observa que también aguardan similitudes con el módulo meteorológico.

  En este caso, lo que comparten es la estructura y la organización estructural de la lista de comandos. Evidentemente la lista de comandos en este módulo orientador no solo es diferente, sino que es más larga (poniendo de manifiesto la mayor complejidad o exigencia de este módulo).

  Resumiendo, `RequestList` se encarga de inicializar y presentar la lista de comandos disponibles (nombre, invocación y parámetros), mientras que `AbstractRequest` presenta la plantilla que cada comando debe cumplir en su implementación.

  Cada comando debe presentar un nombre, unos parámetros y una acción propios (método *run*). La acción normalmente devuelve una respuesta que a su vez es la que se le devuelve al cliente fruto de la ejecución de ese comando (satisfactoria o no).

- *Clases hardware.* En esta categoría entran las clases protagonistas del bajo nivel del servidor, en definitiva, las que gestionan y comunican todo lo relativo al estado y las acciones hardware. En este caso, esas clases son `EngineConnection`, `EngineAction` y `EngineStatus`.

  En este caso se presenta la necesidad de tener no dos, sino tres clases encargadas de la gestión completa del hardware (a diferencia del módulo meteorológico). Aunque presentan nombres esclarecedores, añadir que se encargan de la conexión con el motor, de ejecutar acciones en el mismo (lectura o escritura) y de manejar/consultar el estado de este (a nivel físico o de aplicación). Gran parte de estas clases son utilizadas por la mayoría de las implementaciones de comandos y por `RequestManager` (directamente o no).

- *Clases auxiliares.* En esta categoría se agrupan clases que recogen cierta lógica requerida para el correcto funcionamiento de algún comando (u otra parte de la aplicación). La clase principal de esta categoría es `MathsTools`.

  Como se puede intuir de su nombre y de su composición, recoge la parte matemática encargada de albergar los cálculos para la conversión lineal entre grados y tacómetros, así como los cálculos necesarios para rotar de forma óptima desde un origen a un destino específico (recorriendo el menor número de grados posible).

  Aparte de esta clase, incluimos también la clase `Configuration`, que permite la lectura y escritura de valores de configuración claves para el funcionamiento apropiado del sistema. En este caso, la configuración está segmentada en dos, por un lado, tendríamos la configuración del cliente y, por otro, la configuración del servidor (este último presenta quizás una mayor importancia, relativamente). Los ajustes de configuración se muestran en el anexo *API*.

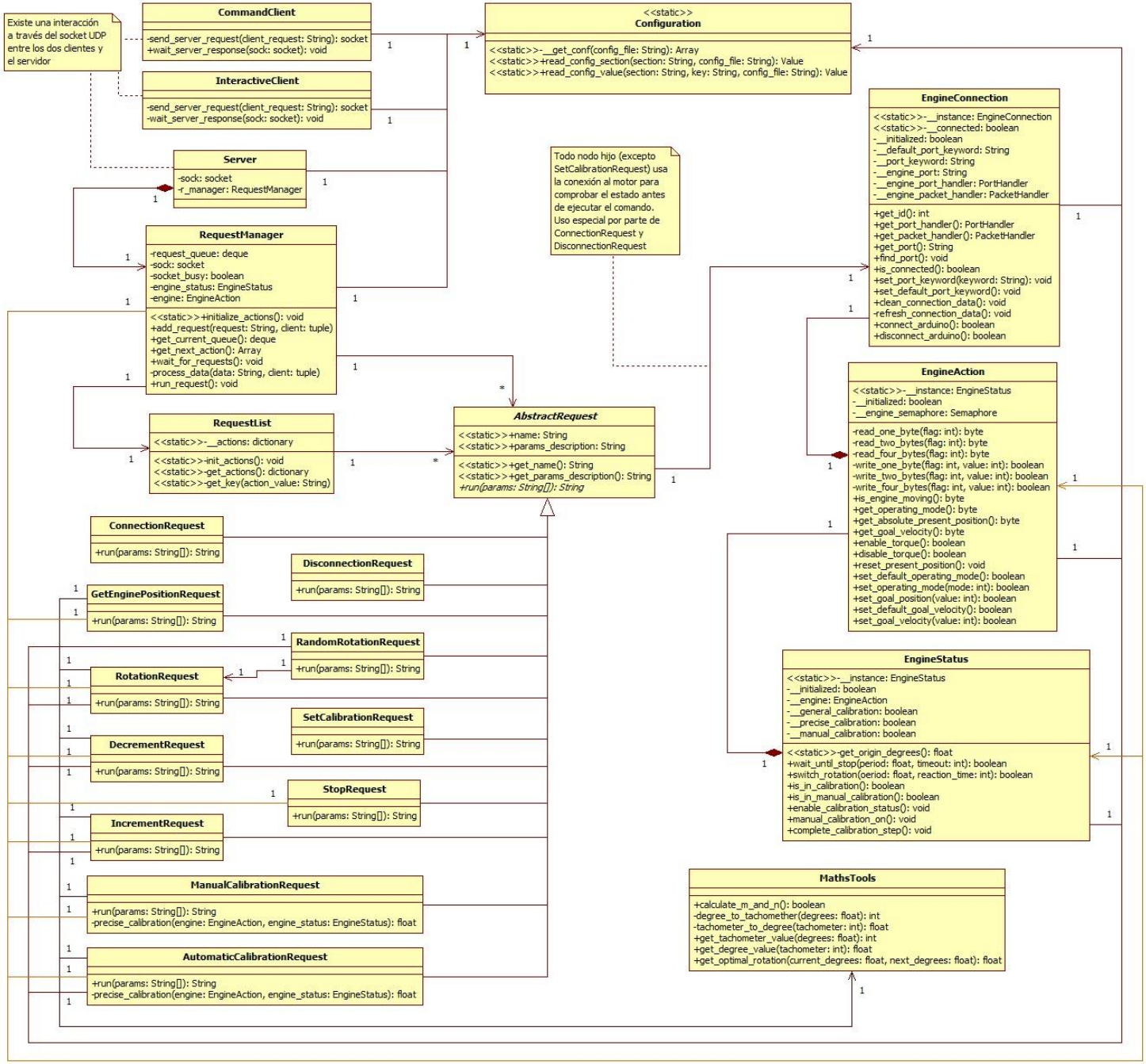


Figura 7.37. Diagrama de clases del desarrollo en Python del módulo orientador.

Con el diagrama de clases comentado, quedan expuestas las clases protagonistas del software desarrollado y un concepto general del funcionamiento.

En la misma línea del desarrollo del otro módulo, comentaremos resumidamente una visión más dinámica de esta jerarquía de clases. Con la clase/*script* servidor (*Server*) en funcionamiento, todas las peticiones entrantes del cliente que se decida ejecutar se van encolando y ejecutando (se emplean dos hilos para ello).

Las peticiones que llegan se contrastan con la lista de peticiones disponibles (`RequestList`) y si la petición encaja correctamente se ejecuta el comando en cuestión (si fuera posible, en función del estado del sistema). Esto es así porque recordemos que, si por casualidad llega una petición que se asocia al comando de la clase `StopRequest`, esta solo se ejecutará si el sistema está en modo calibración, de lo contrario, realmente constituiría un comando incorrecto dentro del contexto actual.

Si se admite y se ejecuta el comando en cuestión, recordemos que hará uso de las *clases hardware* para lograr su cometido. Si el comando en cuestión presenta dependencias con alguna de las clases auxiliares, hará uso de ellas para lograr su cometido.

Una anotación final dentro de este módulo orientador para *ponerle la guinda al pastel* es que en el desarrollo se emplearon los mismos patrones de diseño y las mismas técnicas relacionadas con el *clean code* que se emplearon en el módulo meteorológico. Este contenido estará visible por tanto en el mismo Anexo que indicamos en su momento, el anexo *Patrones de diseño y clean code*.

8. Conclusiones y trabajos futuros

En este capítulo de la memoria, se agruparán las conclusiones obtenidas tras la realización de este TFM y se expondrá una perspectiva futura relacionada con el trabajo realizado.

8.1. Conclusiones

Podemos afirmar que, con todo lo trabajado y desarrollado a lo largo de este TFM, se han logrado los objetivos esenciales expuestos al comienzo de este documento, es decir, a grandes rasgos, el desarrollo funcional de los dos módulos solicitados: la estación meteorológica y el módulo orientador. Aparte de eso, se ha logrado satisfacer el resto de los objetivos secundarios expuestos.

Gracias a nuestro desarrollo, el proyecto de un sistema IPS más completo está al alcance de nuestra mano. Al considerar aspectos tan relevantes como la orientación o el estado ambiental, sin duda se da un gran paso en el ámbito de los IPS. Incluso si resulta que los resultados procesados llegan a no ser tan relevantes como se esperaba, habremos dado un gran paso demostrando que muchos de los estudios y planteamientos encontrados no estaban en la certeza absoluta, estrechando favorablemente el conjunto de soluciones o variables posibles.

Es cierto que muchos de los resultados y frutos de nuestro esfuerzo se verán plasmados y trasladados a la realidad en ciertos trabajos futuros que se comentarán en otro apartado, después de estas conclusiones. Estos trabajos futuros de alguna manera darán forma y protagonismo al trabajo desarrollado dentro del proyecto IPS. Con estas palabras no se pretende robar mérito al resto de módulos (no relacionados con este TFM), pues todos y cada uno de los módulos tienen su papel fundamental en el IPS y son igual de importantes.

Dejando de un lado el resultado que ofrecemos con este TFM, es conveniente resaltar igualmente aspectos más subjetivos, como la experiencia y conocimientos envueltos en la realización de este TFM.

En primer lugar, gracias al planteamiento de este TFM, se ha tenido la oportunidad de abordar no solo el contexto, sino un proyecto real relacionado con el posicionamiento en interiores. Sin duda, a nivel personal y profesional me parece una línea de investigación interesante, ya que al igual que el posicionamiento en exteriores, el avance tecnológico en el posicionamiento en interiores ofrece una gran cantidad de servicios susceptibles de ser explotados, por ejemplo, la hora de visitar ciertos lugares (en términos de orientación y eficiencia). Ya sea por ocio (visita de un museo) o por necesidad (centro de salud), con un IPS orientarnos fácilmente en interiores es una realidad cercana que indudablemente aporta una gran agilidad e independencia al visitante.

Por otra parte, y para concluir este apartado, este TFM nos ha servido para reforzar algunos conocimientos adquiridos no solo en el Máster, sino en el propio Grado en Ingeniería Informática. Desde la investigación y el análisis de conceptos totalmente

nuevos, hasta el desarrollo de una aplicación (elaborada en Python, en este caso) que toca conceptos relacionados con la comunicación (cliente-servidor, protocolo UDP…) y con el mundo del software que todos conocemos.

8.2. Trabajos futuros

En lo que respecta a los módulos desarrollados, aunque es evidente que se encuentran en un estado final (a nivel funcional), no significa que no estén expuestos a mejoras y adaptaciones futuras en cualquiera de los sentidos. Es por ello por lo que a lo largo de este breve apartado comentaremos los aspectos relevantes en cuanto a ampliación de nuestros módulos se refiere. A continuación, presentamos esos detalles:

- Encapsulamiento físico. El resultado de nuestros módulos envuelve principalmente el desarrollo software y un desarrollo hardware mínimo para su funcionamiento. Naturalmente el punto de mejora más claro viene dado por este concepto de encapsulamiento. En el futuro, tanto la estación meteorológica como el sistema orientador se encapsularán y se ubicarán de manera óptima en la plataforma final del sistema IPS.

  Por exponer un ejemplo, la estación meteorológica podría consistir en una superficie donde irían adheridos tanto la placa Arduino (envuelta en un recipiente específico para ocultar y proteger la placa) como los diferentes sensores. Por otro lado, el orientador podría consistir en el motor Dynamixel atornillado a alguna parte estratégica de la plataforma al cual se le incorporaría una extensión con el módulo receptor de señales Bluetooth.

  Todo esto no es algo trivial, es algo que hay que estudiar y analizar para determinar la mejor manera de incorporar físicamente estos módulos al IPS.

- Evaluación de comportamiento de los sensores de la estación. En el futuro, cuando se obtengan resultados empíricos avanzados y/o surjan nuevos conocimientos en el ámbito que rodea a este proyecto, se pueden plantear mecanismos de mejora que busquen contemplar otro tipo de variables meteorológicas o que busquen sustituir los sensores planteados para nuestro módulo meteorológico (por sensores de mejor precisión, mejor relación calidad-precio en el futuro…).

  En esta labor futura, una tarea muy interesante sería una comparación exhaustiva de los resultados ofrecidos por un conjunto variado de sensores, lo cual no ha sido posible en este TFM (ya sea por tiempo, recursos…).

- Solución para determinados errores y mejora continua. Seríamos poco realistas si pensamos que un producto software no está expuesto a un mantenimiento o una mejora continua. Existen varias circunstancias que inducen a mantener el código como lo son: actualizaciones de los lenguajes o de las librerías, aparición de errores no contemplados en el desarrollo, cambio en las necesidades… Es por ello, que en el futuro y siempre que sea posible, el desarrollo de estos módulos dará soporte al proyecto IPS global.

Bibliografía

1. **RAE.** Nomadismo. [En línea] https://dle.rae.es/nomadismo?m=form.
2. —. Nómada. [En línea] https://dle.rae.es/n%C3%B3mada?m=form.
3. —. Sedentarismo. [En línea] https://dle.rae.es/sedentarismo?m=form.
4. —. Sedentario. [En línea] https://dle.rae.es/sedentario?m=form.
5. **Wikipedia.** Astronomía. [En línea] https://es.wikipedia.org/wiki/Astronom%C3%ADa.
6. **educa-ciencia.** Orientación mediante la astronomía. [En línea] https://www.educa-ciencia.com/astronomia-orientacion.htm.
7. **Google.** Orientación por indicios naturales. [En línea] https://sites.google.com/site/actividadesenlanaturaleza/orientacion-por-indicios-naturales.
8. **Pospelova, Vera.** Posicionamiento.
9. **Wikipedia.** TRANSIT. [En línea] https://es.wikipedia.org/wiki/Transit_(sat%C3%A9lite).
10. —. Navstar. [En línea] https://es.wikipedia.org/wiki/Navstar.
11. —. GLONASS. [En línea] https://es.wikipedia.org/wiki/GLONASS.
12. —. Beidou. [En línea] https://es.wikipedia.org/wiki/Beidou.
13. —. Galileo. [En línea] https://es.wikipedia.org/wiki/Galileo_(navegaci%C3%B3n_por_sat%C3%A9lite).
14. **Unigis.** Posicionamiento Interiores. [En línea] https://www.unigis.es/posicionamiento-indoor/ .
15. **Abalit.** IPS Beacons. [En línea] https://www.abalit.org/blog/post/sistema-posicionamiento-interiores-beacon/es.
16. **Wikipedia.** RSSI. [En línea] https://es.wikipedia.org/wiki/Indicador_de_fuerza_de_la_se%C3%B1al_recibida.
17. **Science Direct.** RSS. [En línea] https://www.sciencedirect.com/topics/computer-science/received-signal-strength.
18. **Researchgate.** Study on an Indoor Positioning System for Harsh Environments Based on Wi-Fi and Bluetooth Low Energy. [En línea] https://www.researchgate.net/publication/317376668_Study_on_an_Indoor_Positioning_System_for_Harsh_Environments_Based_on_Wi-Fi_and_Bluetooth_Low_Energy.
19. —. Beacon-Related Parameters of Bluetooth Low Energy: Development of a Semi-Automatic System to Study Their Impact on Indoor Positioning Systems. [En línea] https://www.researchgate.net/publication/334435471_Beacon-Related_Param_of_Bluetooth_Low_Energy_Development_of_a_Semi-Automatic_System_to_Study_Their_Impact_on_Indoor_Positioning_Systems.
20. **AccedaCRIS.** A Survey on Bluetooth Low Energy Indoor Positioning Systems. [En línea] https://accedacris.ulpgc.es/handle/10553/70818.
21. **Wikipedia.** Tecnología Wifi. [En línea] https://es.wikipedia.org/wiki/Wifi.
22. —. Bluetooth. [En línea] https://es.wikipedia.org/wiki/Bluetooth.
23. —. BLE. [En línea] https://es.wikipedia.org/wiki/Bluetooth_de_baja_energ%C3%ADa.
24. —. Ultrawideband. [En línea] https://es.wikipedia.org/wiki/Ultrawideband.
25. —. Triangulación. [En línea] https://en.wikipedia.org/wiki/Triangulation_(surveying) .
26. —. Trilateración. [En línea] https://es.wikipedia.org/wiki/Trilateraci%C3%B3n.

27. —. K-Neighbours. [En línea]
https://es.wikipedia.org/wiki/K_vecinos_m%C3%A1s_pr%C3%B3ximos.
28. **Datascience.** Interferencia Bayesiana. [En línea]
https://datascience.com.co/c%C3%B3mo-funciona-la-inferencia-bayesiana-dc4ad29d4697 (alternativa: Wikipedia).
29. **APD.** Machine Learning. [En línea] https://www.apd.es/que-es-machine-learning/.
30. **Fritzing.** Web Fritzing. [En línea] https://fritzing.org/home/.
31. **Arduino.** Web Arduino. [En línea] https://www.arduino.cc/.
32. **Wikipedia.** Arduino IDE. [En línea] https://en.wikipedia.org/wiki/Arduino_IDE.
33. **JetBrains.** PyCharm. [En línea] https://www.jetbrains.com/pycharm/.
34. —. Web JetBrains. [En línea] https://www.jetbrains.com/.
35. **Git.** Web Git. [En línea] https://git-scm.com/.
36. **SourceTree.** Web SourceTree. [En línea] https://www.sourcetreeapp.com/.
37. **Bitbucket.** Web Bitbucket. [En línea] https://bitbucket.org/product/.
38. **Microsoft.** Office 365. [En línea] https://www.microsoft.com/es-es/microsoft-365.
39. **Sourceforge.** FreeMind. [En línea]
http://freemind.sourceforge.net/wiki/index.php/Main_Page.
40. **Robotis.** R+ Manager 2.0. [En línea]
https://emanual.robotis.com/docs/en/software/rplus2/manager/.
41. **BOE.** LOPD. [En línea] https://www.boe.es/boe/dias/1999/12/14/pdfs/A43088-43099.pdf.
42. —. RGPD. [En línea] https://www.boe.es/doue/2016/119/L00001-00088.pdf.
43. **ULPGC.** Competencias Máster. [En línea]
https://www2.ulpgc.es/archivos/plan_estudios/5038_50/MasterenIngenieriaInformatica2014_Parte2.pdf.
44. —. Competencias Grado. [En línea]
https://www2.ulpgc.es/archivos/plan_estudios/4008_40/ObjetivosyCompetenciasdelGII.pdf.
45. **Wikipedia.** Desarrollo en Cascada. [En línea]
https://es.wikipedia.org/wiki/Desarrollo_en_cascada.
46. —. Modelo de Prototipos. [En línea]
https://es.wikipedia.org/wiki/Modelo_de_prototipos .
47. —. Scrum. [En línea]
https://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software).
48. —. Frecuencia 2.4 GHz. [En línea]
https://en.wikipedia.org/wiki/2.4_GHz_radio_use.
49. **Goldtouch.** How to Stop Bluetooth Interference From Messing With Other Devices. *www.goldtouch.com.* [En línea] 06 de noviembre de 2014.
https://www.goldtouch.com/stop-bluetooth-interference-messing-devices/.
50. **Harwood, Michael and Dulaney, Emmett.** *CompTIA Network+ N10-005 Exam Cram (4th Edition).* 2012. https://flylib.com/books/en/1.152.1.33/1/. v078974905X.
51. **Granby, John.** Chron. *What Are Some Signal Interference Issues for the Bluetooth Technology?* [En línea] https://smallbusiness.chron.com/signal-interference-issues-bluetooth-technology-58273.html.
52. **Liu, Jie.** Quora. *What environmental factors affect bluetooth signal strength?* [Foro]. 12 de agosto de 2014. https://www.quora.com/What-environmental-factors-affect-bluetooth-signal-strength.

53. **Harwood, Mike.** *CompTIA Network+ N10-004 Exam Cram (3rd Edition).* s.l. : Pearson IT Certification, 2009. http://www.pearsonitcertification.com/articles/article.aspx?p=1329709&seqNum=3.
54. **Johnson, Raymond L.** The Weather Station. *Does weather affect your wifi?* [Blog]. 11 de Marzo de 2018. https://the-weather-station.com/does-weather-affect-your-wifi/.
55. **McClaughry, Thomas.** Quora. *Does cold weather affect Bluetooth connectivity?* [Foro]. 2019. https://www.quora.com/Does-cold-weather-affect-Bluetooth-connectivity.
56. **Wikipedia.** Onda. [En línea] https://es.wikipedia.org/wiki/Onda.
57. —. Espectro electromagnético. [En línea] https://es.wikipedia.org/wiki/Espectro_electromagn%C3%A9tico.
58. —. Radiación Electromagnética. [En línea] https://es.wikipedia.org/wiki/Radiaci%C3%B3n_electromagn%C3%A9tica.
59. **20minutos.** Luz y Sonido. [Blog]. 29 de agosto de 2008. https://blogs.20minutos.es/ciencia/2008/08/29/luz-y-sonido/.
60. **Khanacademy.** *La luz: ondas electromagnéticas, espectro electromagnético y fotones.* https://es.khanacademy.org/science/physics/light-waves/introduction-to-light-waves/a/light-and-the-electromagnetic-spectrum.
61. **Electronics-notes.** *Antennas propagation ionospheric sun HF radio propagation.* [Web]. https://www.electronics-notes.com/articles/antennas-propagation/ionospheric/sun-hf-radio-propagation.php.
62. **Fondriest.** Solar Radiation & Photosynthetically Active Radiation. [En línea] https://www.fondriest.com/environmental-measurements/parameters/weather/photosynthetically-active-radiation/#PAR1.
63. **Mungra, Nisarg.** Quora. *What is the frequency of sunlight? .* [Foro]. 22 de diciembre de 2018. https://www.quora.com/What-is-the-frequency-of-sunlight.
64. **Tyler.** Wirebiters. *Why do fluorescent lights mess up my WiFi signal?* 22 de enero de 2014. https://www.wirebiters.com/fluorescent-lights-mess-wifi-signal/.
65. **Luomala, Jari and Hakala, Ismo.** Annals-csis . *Effects of Temperature and Humidity on RadioSignal Strength in Outdoor Wireless SensorNetworks.* https://annals-csis.org/proceedings/2015/pliks/241.pdf.
66. **Wendt, Zach.** *Presentamos las mejores diez placas de desarrollo de 2018 .* 28 de marzo de 2018. https://www.arrow.com/es-mx/research-and-events/videos/the-top-10-development-platforms-dev-kits-2018.
67. **Lozano, Iván.** *Cuatro alternativas a Arduino: BeagleBone, Raspberry Pi, Nanode y Waspmote.* 09 de enero de 2013. https://blogthinkbig.com/4-alternativas-arduino-beaglebone-raspberrypi-nanode-waspmote.
68. **Arduino.** Foro Arduino. [En línea] https://forum.arduino.cc/.
69. —. Tienda Arduino. [En línea] https://store.arduino.cc/arduino-uno-rev3.
70. **Wikipedia.** Microcontrolador. [En línea] https://es.wikipedia.org/wiki/Microcontrolador.
71. —. Microprocesador. [En línea] https://es.wikipedia.org/wiki/Microprocesador.
72. **Raspberry Pi.** Foro Raspberry. [En línea] https://www.fororaspberry.es/.
73. **Prácticas con Arduino.** Detector de luz. [En línea] http://www.practicasconarduino.com/manualrapido/detector_de_luz.html.
74. **Naylampmechatronics.** Tutorial módulo sensor de luz BH1750 . [En línea] https://naylampmechatronics.com/blog/44_Tutorial-m%C3%B3dulo-sensor-de-luz-

BH1750.html.
75. **Cetronic .** Sensor de luz TSL2561 High Adafruit para Arduino. [En línea] https://www.cetronic.es/sqlcommerce/disenos/plantilla1/seccion/producto/DetalleProducto.jsp?idIdioma=&idTienda=93&codProducto=999334022&cPath=1343.
76. **Sensirion.** Digital Humidity Sensor SHT7x (RH/T) (End of Life). [En línea] https://www.sensirion.com/en/environmental-sensors/humidity-sensors/pintype-digital-humidity-sensors/.
77. **Programar Fác¡l.** Sensor DHT11. [En línea] https://programarfacil.com/blog/arduino-blog/sensor-dht11-temperatura-humedad-arduino/.
78. **Programar Fácil .** Sensor LM35 . [En línea] https://programarfacil.com/tutoriales/fragmentos/leer-el-sensor-de-temperatura-lm35-en-arduino/.
79. **Bluedot .** BlueDot BME280+TSL2591 Advanced Weather Station . [En línea] https://www.bluedot.space/sensor-boards/bme280-tsl2591/.
80. **Wikipedia.** Pull-up resistor. [En línea] https://en.wikipedia.org/wiki/Pull-up_resistor.
81. **Robotis.** XM430-W350. [En línea] https://emanual.robotis.com/docs/en/dxl/x/xm430-w350/ .
82. —. U2D2 Power Hub. [En línea] https://emanual.robotis.com/docs/en/parts/interface/u2d2_power_hub/ .
83. —. U2D2 Interface. [En línea] https://emanual.robotis.com/docs/en/parts/interface/u2d2/ .
84. **Refactoring guru.** Singleton. [En línea] https://refactoring.guru/es/design-patterns/singleton.
85. **Wikipedia.** Polimorfismo. [En línea] https://es.wikipedia.org/wiki/Polimorfismo_(inform%C3%A1tica).
86. **Refactoring guru.** Replace conditional with polymorphism. [En línea] https://refactoring.guru/es/replace-conditional-with-polymorphism.

Anexo 1. API

En este anexo se comentarán para cada uno de los dos módulos, la API de comandos en la parte del servidor. Estos comandos son los que se pueden ejecutar por parte del cliente. Hay que recalcar que los comandos son configurables mediante el archivo de configuración del servidor, por lo que aquí definiremos la lista de comando partiendo de la configuración por defecto establecida por nosotros mismos.

API. Común en ambos módulos.

La lista de comandos presente en ambos módulos (aunque pueden personalizarse de forma independiente) es la siguiente:

- *exit* → Al enviar una solicitud con este comando el programa cliente finaliza y la parte del servidor que escucha peticiones entrantes también lo hace. Ejecutar este comando solo cuando sea seguro que no se vayan a enviar más comandos.
- *help* → Al enviar una solicitud con este comando al servidor el cliente recibe una respuesta en formato texto con el listado de comandos aceptados por el servidor (atendiendo a la configuración actual). Si se ha personalizado la lista de comandos, con recordar el valor de este comando sería suficiente para un fácil acceso al resto.
- *conn [cadena de búsqueda]* → Este comando permite establecer la conexión entre el servidor y la parte hardware del módulo. Admite un parámetro que se usaría para localizar el dispositivo conectado (si no se suministra, se toma el valor por defecto del archivo de configuración). Esto es indispensable para ejecutar el conjunto de órdenes que siguen a continuación. Se devuelve una cadena de texto con el resultado de ejecución (exitoso o no).
- *disc* → Este comando permite finalizar la conexión entre el servidor y la parte hardware del módulo. Esto es útil cuando ya no se van a ejecutar más comandos donde participe la parte hardware o si se experimenta algún tipo de problema con el comando *conn*. Se devuelve una cadena de texto con el resultado de ejecución (exitoso o no).

API. Módulo de la Estación Meteorológica

La lista de comandos presente para la estación meteorológica es la siguiente:

- *read* → Este comando es el más importante desde una perspectiva funcional, ya que permite obtener todos los valores de los sensores de la parte hardware, es decir, el estado ambiental obtenido de todos esos sensores. Este comando devuelve una cadena de texto adaptada para una fácil lectura, pero detrás se esconde una estructura en formato JSON con los siguientes datos:
    - *temperature* → Valor actual de la temperatura ambiental.
    - *humidity* → Valor actual de la humedad relativa (porcentual).

- *abs_humidity* → Valor actual de la humedad absoluta (gramos/m^3).
- *light* → Valor actual de la luz ambiental (lúmenes).
- *datetime* → Valor de la fecha y hora en que se realizó la lectura.

API. Módulo del Sistema Orientador

La lista de comandos presente para el sistema orientador presenta algunas similitudes (cuatro comandos) respecto al módulo de la estación meteorológica y es la siguiente:

- *pos* → Este comando permite obtener la posición actual del motor. Actualmente se devuelve una cadena de texto con los valores de posición para la posición absoluta del motor y para la posición relativa, tanto en grados como en tacómetros del motor. La posición relativa es la que parte de un origen personalizado, es decir, que los cero grados absolutos del motor no tienen por qué coincidir con nuestros cero grados relativos.
- *rand [número de rotaciones]* → Este comando permite rotar el motor a una posición aleatoria el número de veces deseado (indicado por el parámetro). Por defecto, se rota aleatoriamente solo una vez.
- *rotate [grados]* → Este comando permite rotar el motor a la posición deseada (en grados). Esta rotación toma en cuenta nuestro origen relativo en grados, que puede o no ser el mismo que el origen absoluto (los cero grados del motor).
- *set [grados]* → Este comando establece de forma explícita el origen relativo en grados de nuestro sistema orientador. En definitiva, realiza una escritura en el archivo de configuración almacenando ese valor.
- *ac* → Este comando ejecuta el proceso de calibración automática. Durante este proceso, se requiere la interacción del usuario (comandos *stop*).
- *mc* → Este comando ejecuta el proceso de calibración automática. Durante este proceso, se requiere la interacción del usuario (comandos *stop, up, down*).

A parte de estos comandos, existen tres adicionales que se atenderán exclusivamente cuando el sistema orientador este en proceso de calibración (etapas concretas):

- *stop* → Se encarga de parar el motor en una fase concreta de calibración.
- *up* → En el calibrado manual, se encarga de incrementar la posición actual del motor (los grados dependen del valor en los archivos de configuración).
- *down* → En el calibrado manual, se encarga de decrementar la posición actual del motor (los grados dependen del valor en los archivos de configuración).

Módulo Meteorológico. Configuración cliente.

- SETTINGS
  - SERVER_IP: Este valor especifica la dirección IP del servidor al que irán

dirigidas las peticiones/comandos. Debe tener el formato de una dirección IP válida y debe ser la misma que la establecida en el servidor.

- SERVER_IP: Este valor especifica el puerto del servidor que acompaña al anterior parámetro de configuración. Debe ser un numero de puerto válido, normalmente se emplean cinco dígitos. Debe ser el mismo que el establecido en el servidor.

- CLIENT_COMMANDS
    - EXIT_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de salida. Debe ser el mismo que el establecido en el lado servidor.
    - HELP_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de ayuda. Debe ser el mismo que el establecido en el lado servidor.

Módulo Meteorológico. Configuración servidor.

- SETTINGS
    - SERVER_IP: Este valor especifica la dirección IP en la que está accesible el servidor. Debe tener el formato de una dirección IP válida.
    - SERVER_IP: Este valor especifica el puerto de escucha del servidor y va ligado al anterior parámetro de configuración. Debe ser un número de puerto válido, normalmente se emplean cinco dígitos.
    - PORT_KEYWORD: Cadena de texto empleada por defecto para localizar el puerto COM al que está conectado nuestro Arduino.
    - LOG_ENABLED: Valor numérico para establecer el log activado (1) o desactivado (0).

- COMMANDS
    - EXIT_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de salida.
    - HELP_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de ayuda.
    - CONNECT_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de conexión.
    - DISCONNECT_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de desconexión.
    - READ_SENSORS_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de lectura de sensores.

- VALUES

    - ARDUINO_OK_MESSAGE: Valor numérico que indica el valor empleado para determinar que Arduino está listo.

    - PYTHON_OK_MESSAGE: Valor numérico que indica el valor empleado por Python para comunicarse con Arduino (solicitud de lectura y confirmación de recepción de datos). Si hubiera más comandos, existirían un valor diferente por cada uno de ellos.

    - TIME_BETWEEN_COMMANDS: Tiempo en segundos que el servidor espera entre la ejecución de un comando y otro por parte del cliente.

Módulo Orientador. Configuración cliente.

- SETTINGS

    - SERVER_IP: Este valor especifica la dirección IP del servidor al que irán dirigidas las peticiones/comandos. Debe tener el formato de una dirección IP válida y debe ser la misma que la establecida en el servidor.

    - SERVER_IP: Este valor especifica el puerto del servidor que acompaña al anterior parámetro de configuración. Debe ser un numero de puerto válido, normalmente se emplean cinco dígitos. Debe ser el mismo que el establecido en el servidor.

- KEYS

    - STOP_KEY: Identificador numérico de la tecla usada para ejecutar el comando de parada del motor (fase de calibración). Valor 13 por defecto (tecla *enter*).

    - UP_KEY: Identificador numérico de la tecla usada para ejecutar el comando de incremento de posición del motor (fase de calibración). Valor 43 por defecto (tecla *+*).

    - DOWN_KEY: Identificador numérico de la tecla usada para ejecutar el comando de decremento de posición del motor (fase de calibración). Valor 45 por defecto (tecla -).

- CLIENT_COMMANDS

    - EXIT_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de salida. Debe ser el mismo que el establecido en el lado servidor.

    - HELP_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de ayuda. Debe ser el mismo que el establecido en el lado servidor.

    - AUTOMATIC_CALIBRATE_COMMAND: Cadena de texto con el nombre de

comando establecido para el comando de calibración automática. Debe ser el mismo que el establecido en el lado servidor.

- MANUAL_CALIBRATE_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de calibración manual. Debe ser el mismo que el establecido en el lado servidor.

- CALIBRATE_STOP_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de parada (fase de calibración). Debe ser el mismo que el establecido en el lado servidor.

- CALIBRATE_UP_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de incrementar posición del motor (fase de calibración). Debe ser el mismo que el establecido en el lado servidor.

- CALIBRATE_DOWN_COMMAND: Cadena de texto con el nombre de comando establecido para el comando decrementar posición del motor (fase de calibración). Debe ser el mismo que el establecido en el lado servidor.

Módulo Orientador. Configuración servidor.

- SETTINGS

    - DXL_ID: Representa el valor identificador (numérico) de nuestro motor orientador. Por defecto, se establece como 9 y no se debe cambiar a no ser que se cambie primero en el motor.

    - BAUDRATE = Representa el *baudrate* del motor. Por defecto está a 57600 y no se recomienda cambiarlo.

    - PROTOCOL_VERSION = Valor especifico de configuración del motor. El protocolo por defecto es el 2.0 y no se recomienda cambiarlo.

    - PORT_KEYWORD: Cadena de texto empleada por defecto para localizar el puerto COM al que está conectado nuestro Arduino.

    - SERVER_IP: Este valor especifica la dirección IP en la que está accesible el servidor. Debe tener el formato de una dirección IP válida.

    - SERVER_IP: Este valor especifica el puerto de escucha del servidor y va ligado al anterior parámetro de configuración. Debe ser un número de puerto válido, normalmente se emplean cinco dígitos.

- COMMANDS

    - EXIT_COMMAND: Cadena de texto con el nombre de comando establecido para el comando de salida.

    - HELP_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de ayuda.

- CONNECT_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de conexión al motor.

- DISCONNECT_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de desconexión del motor.

- GET_DEGREES_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de obtención de la posición actual del motor.

- RANDOM_ROTATE_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de rotación aleatoria del motor.

- ROTATE_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de rotación del motor.

- SET_CALIBRATION_VALUE_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de establecimiento de un origen relativo en grados.

- AUTOMATIC_CALIBRATE_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de calibración automática.

- MANUAL_CALIBRATE_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de calibración manual.

- CALIBRATE_STOP_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de parar el motor en fase de calibración.

- CALIBRATE_UP_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de incremento de posición del motor en fase de calibración.

- CALIBRATE_DOWN_COMMAND = Cadena de texto con el nombre de comando establecido para el comando de decremento de posición del motor en fase de calibración.

- OPERATING MODES

    - CURRENT_CM: Por especificación oficial, asociado al valor 0. Este modo controla la corriente (torque) y parece el mejor para gobernar una pinza.

    - VELOCITY_CM = Por especificación oficial, asociado al valor 1. Este modo controla la velocidad y parece el mejor para robots con ruedas. Interesante para nuestro modulo orientador (en algunos aspectos).

    - POSITION_CM = Por especificación oficial, asociado al valor 3. Este modo controla posición para sistemas que roten entre 0 y 360 grados. Interesante para nuestro modulo orientador.

- EXTENDED_POSITION_CM = Por especificación oficial, asociado al valor 4. Este modo controla posición para sistemas que roten fuera del rango de los 0 y 360 grados. Interesante para nuestro modulo orientador.

- CURRENT_BASED_POSITION_CM = Por especificación oficial, asociado al valor 5. Este modo controla la posición y corriente (torque). Parece el mejor para robots articulados o pinzas que requieran control de posición y corriente.

- VOLTAGE_CM = Por especificación oficial, asociado al valor 16. Controla directamente el PWM (Voltaje) del motor. Parece ser el más *elemental.*

- ADDRESSES

    - ADDR_OPERATING_MODE: Por especificación oficial, la dirección de memoria para establecer el modo de operación es la 11.

    - ADDR_TORQUE_ENABLE: Por especificación oficial, la dirección de memoria para establecer el modo de operación es la 6.

    - ADDR_GOAL_POSITION: Por especificación oficial, la dirección de memoria para establecer la posición objetivo (dentro de los modos de operación POSITION_CM y EXTENDER_POSITION_CM) es la 116.

    - ADDR_PRESENT_POSITION: Por especificación oficial, la dirección de memoria donde se almacena la posición actual del motor es la 132 (solo lectura).

    - ADDR_GOAL_VELOCITY: la dirección de memoria para establecer la velocidad objetivo (dentro del modo de operación VELOCITY_CM) es la 104.

    - ADDR_PRESENT_VELOCITY: Por especificación oficial, la dirección de memoria donde se almacena la velocidad actual del motor es la 128 (solo lectura).

    - ADDR_MOVING: Por especificación oficial, la dirección de memoria donde se almacena el valor que determina si el motor se está o no moviendo es la 122.

- VALUES

    - TORQUE_ENABLE: Valor establecido a 1 por defecto y que habilita el *torque* del motor. No cambiar ya que viene predefinido por las especificaciones del motor.

    - TORQUE_DISABLE: Valor establecido a 0 por defecto y que deshabilita el *torque* del motor. No cambiar ya que viene predefinido por las especificaciones del motor.

- IN_MOTION: Valor que indica que el motor está actualmente moviéndose. Este valor es 1 y no se recomienda cambiarlo.

- NOT_IN_MOTION: Valor que indica que el motor está actualmente estático. Este valor es 0 y no se recomienda cambiarlo.

- DEFAULT_VELOCITY: Velocidad por defecto a la que se mueve el motor en el modo de operación VELOCITY_CM. Debe ser un valor numérico razonable de 15 en adelante para un funcionamiento idóneo. Se recomienda un valor entre 20 y 40.

- TIME_BETWEEN_RANDOM: Tiempo en segundos que el sistema espera entre rotaciones aleatorias. Aunque se establece a 5 segundos por defecto, puede alterarse sin consecuencias.

- TIME_BETWEEN_COMMANDS: Tiempo en segundos que el servidor espera entre la ejecución de un comando y otro por parte del cliente.

- OPTIMAL_ROTATION: Valor que indica si el motor rotará siempre en el sentido óptimo (valor 1) o no (valor 0). En caso de ser 0, el motor se limitará a rotar entre los 0 y los 360 grados (relativos).

- ORIGIN_DEGREES: Valor de configuración que almacena el origen relativo del orientador (acorde con nuestra plataforma robótica), es decir, los cero grados relativos. Si se establece a 0 el origen relativo coincidirá con el origen absoluto del motor.

- CALIBRATION_TIME_LOOP: Tiempo de espera entre las rotaciones de la calibración automática (segunda fase de calibración).

- CALIBRATION_PRECISION: Número de grados que el motor rota en las fases de calibración (segunda fase de calibración). Establecido a 15.0 para apreciar a nivel humano el comportamiento, lo óptimo es disminuir el valor para aumentar la precisión de la calibración.

- CALIBRATION_RANGE: Numero de grados que marca el intervalo en el que el motor rota en la calibración automática (segunda fase de calibración). Por defecto establecido a 30.0 por las mismas razones que el parámetro anterior. Lo óptimo sería ajustarlo a un valor más pequeño para mejorar la precisión.

Anexo 2. Manual de instalación

A pesar de que nuestros módulos no albergan una complejidad en cuanto a la instalación en sí misma, sí que se tienen que indicar algunas recomendaciones para poder usarlo de forma correcta y sin errores. Es por eso por lo que en este anexo indicaremos de forma breve el procedimiento que seguir.

En primer lugar, empezaremos por la estación meteorológica, concretamente la parte relacionada con Arduino. Considerando que el hardware está montado de acuerdo con las especificaciones plasmadas en el apartado de diseño y de montaje (en caso contrario, se recomienda consultar esta sección), los pasos que seguir serían los siguientes:

- Conectar el hardware Arduino al equipo.
- Abrir el IDE de Arduino.
- Abrimos nuestro programa Arduino (*WeatherVarRead.ino*) y se asegura que la placa seleccionada es *Arduino UNO* y que el puerto es el correcto. Esta configuración se puede modificar desde la pestaña H*erramientas* del IDE.
- Finalmente, se sube el programa a la placa Arduino haciendo uso del botón habitual para ello.
- Con la placa Arduino ejecutando nuestro programa, tocaría configurar la parte desarrollada en código Python (servidor y cliente).

En segundo lugar, hablaremos sobre el segundo módulo, concretamente del motor. Haciendo uso del programa oficial R+ Manager 2.0 se selecciona el modelo de motor y el único ajuste que hay que especificar es el ID del motor. Este debe indicarse también en los archivos de configuración de la aplicación Python (servidor).

Finalmente, como ambos proyectos comparten similitudes en la parte de código Python, se agruparán las anotaciones de configuración en un mismo *saco*. Para la ejecución de servidor y cliente, se recomienda encarecidamente hacer uso del IDE usado para el desarrollo (JetBrains PyCharm), a fin de evitar posibles incidencias. Las tareas indispensables en este punto son las siguientes:

- Añadir todas las dependencias específicamente usadas en el proyecto (fusionando ambos módulos). Estas dependencias son principalmente: *pyserial y dynamixel-sdk.* Otras dependencias con las que ha contado nuestro proyecto (tengan relación o no con el mismo) son: *pip, setuptools, pyparsing, numpy y kaitaistruct.*
- Añadir a las variables de entorno del sistema (concretamente al *Path*) la ruta completa del proyecto. Esto es para evitar errores en la detección de clases del proyecto (y dependencias), entre otras cosas.

Finalmente, solo falta ejecutar el servidor y el cliente deseado para seguir el flujo de funcionamiento de cada módulo correctamente.

Anexo 3. Especificación de casos de uso

CASO DE USO	1	Conectar Arduino		
Descripción		El usuario conecta el programa servidor y la placa Arduino para poder ejecutar acciones.		
Actores		Usuario		
Precondiciones		El Arduino debe estar conectado al equipo servidor y con su programa en ejecución. El programa servidor debe estar en ejecución.		
Flujo normal		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente con el comando conectar (defecto: *conn*) como parámetro asociado.	
		2	El programa servidor recibe el comando y lo encola.	
		3	El programa servidor ejecuta el comando conectar.	
		4	El programa servidor finaliza la ejecución.	
		5	El programa servidor envía un mensaje al usuario indicando que la conexión fue exitosa.	
Postcondiciones		El programa servidor sigue en ejecución y en disposición de aceptar más comando.		
Variaciones		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente con el comando conectar (defecto: *conn*) y con una cadena de reconocimiento como parámetros asociados.	
		1	El usuario ejecuta el programa cliente interactivo y envía el comando de conectar (defecto: *conn*).	
		1	El usuario ejecuta el programa cliente interactivo y envía el comando de conectar (defecto: *conn*) junto a una cadena de reconocimiento como parámetros asociado.	
Extensiones		**Paso**	**Condición**	**Caso de Uso**
Excepciones		5	El programa servidor no se conecta y manda un mensaje de error al usuario.	
Observaciones				

Figura A3.1. Especificación del caso de uso 1: Conectar Arduino.

CASO DE USO	2	Desconectar Arduino		
Descripción		El usuario desconecta el programa servidor y la placa Arduino al no requerir más acciones.		
Actores		Usuario		
Precondiciones		El programa servidor ha pasado satisfactoriamente por el proceso de conexión de Arduino (CU:1).		
Flujo normal		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente con el comando desconectar (defecto: *disc*) como parámetro asociado.	
		2	El programa servidor recibe el comando y lo encola.	
		3	El programa servidor ejecuta el comando desconectar.	
		4	El programa servidor finaliza la ejecución.	
		5	El programa servidor envía un mensaje al usuario indicando que la desconexión fue exitosa.	
Postcondiciones		El programa servidor cierra su conexión con Arduino y no está disponible para aceptar más peticiones entrantes.		
Variaciones		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente interactivo y envía el comando de desconectar (defecto: *disc*).	
Extensiones		**Paso**	**Condición**	**Caso de Uso**
Excepciones		5	El programa servidor no se desconecta y manda un mensaje de error al usuario.	
Observaciones				

Figura A3.2. Especificación del caso de uso 2: Desconectar Arduino.

CASO DE USO	3	Obtener Meteorología		
Descripción		El usuario obtiene los datos meteorológicos de todos los sensores de Arduino.		
Actores		Usuario		
Precondiciones		El programa servidor ha pasado satisfactoriamente por el proceso de conexión de Arduino (CU:1).		
Flujo normal		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente con el comando obtener meteorología (defecto: *read*) como parámetro asociado.	
		2	El programa servidor recibe el comando y lo encola.	
		3	El programa servidor ejecuta el comando desconectar.	
		4	El programa servidor finaliza la ejecución.	
		5	El programa servidor envía un mensaje al usuario con los valores obtenidos de todos los sensores.	
Postcondiciones		El programa servidor sigue en ejecución y en disposición de aceptar más comandos.		
Variaciones		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente interactivo y envía el comando de obtener meteorología (defecto: *read*).	
Extensiones		**Paso**	**Condición**	**Caso de Uso**
Excepciones		5	El programa servidor envía un mensaje de error al usuario o con valores anómalos de los sensores.	
Observaciones				

Figura A3.3. Especificación del caso de uso 3: Obtener Meteorología.

CASO DE USO	4	Solicitar Ayuda		
Descripción		El usuario solicita ayuda al sistema para obtener la lista de comandos disponibles.		
Actores		Usuario		
Precondiciones		El programa servidor está en ejecución		
Flujo normal		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente con el comando de ayuda (defecto: *help*) como parámetro asociado.	
		2	El programa servidor recibe el comando y lo encola.	
		3	El programa servidor ejecuta el comando de ayuda.	
		5	El programa servidor envía un mensaje al usuario con la lista de comandos disponible.	
Postcondiciones		-		
Variaciones		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente sin ningún comando y el sistema entiende que el comando solicitado es el de ayuda (defecto: *help*).	
		1	El usuario ejecuta el programa cliente interactivo y envía el comando de ayuda (defecto: help).	
Extensiones		**Paso**		**Caso de Uso**
Excepciones				
Observaciones				

Figura A3.4. Especificación del caso de uso 4: Solicitar Ayuda.

CASO DE USO	5	Salir		
Descripción		El usuario ha finalizado de utilizar la herramienta y cierra el sistema completo (cliente activo y servidor).		
Actores		Usuario		
Precondiciones		El programa servidor está en ejecución.		
Flujo normal		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente con el comando de salida (defecto: *exit*) como parámetro asociado.	
		2	El programa cliente activo envía el comando ejecutado y finaliza.	
		3	El programa servidor recibe el comando y lo ejecuta inmediatamente.	
		4	El programa servidor finaliza la ejecución.	
		5	El programa servidor cierra la cola de entrada de peticiones y resuelve posibles comandos en cola.	
Postcondiciones		El programa cliente finaliza y el programa servidor ya no está habilitado para añadir más comandos a la cola.		
Variaciones		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente interactivo y envía el comando de salida (defecto: *exit*).	
Extensiones		**Paso**	**Condición**	**Caso de Uso**
Excepciones				
Observaciones				

Figura A3.5. Especificación del caso de uso 5: Salir.

CASO DE USO	6	Conectar Motor		
Descripción		El usuario conecta el programa servidor y el motor Dynamixel para poder ejecutar acciones.		
Actores		Usuario		
Precondiciones		El motor Dynamixel debe estar conectado al equipo servidor y el programa servidor debe estar en ejecución.		
Flujo normal		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente con el comando conectar (defecto: *conn*) como parámetro asociado.	
		2	El programa servidor recibe el comando y lo encola.	
		3	El programa servidor ejecuta el comando conectar.	
		4	El programa servidor finaliza la ejecución.	
		5	El programa servidor envía un mensaje al usuario indicando que la conexión fue exitosa.	
Postcondiciones		El programa servidor sigue en ejecución y en disposición de aceptar más comando.		
Variaciones		**Paso**	**Acción**	
		1	El usuario ejecuta el programa cliente con el comando conectar (defecto: *conn*) y con una cadena de reconocimiento como parámetros asociados.	
		1	El usuario ejecuta el programa cliente interactivo y envía el comando de conectar (defecto: *conn*).	
		1	El usuario ejecuta el programa cliente interactivo y envía el comando de conectar (defecto: *conn*) junto a una cadena de reconocimiento como parámetros asociado.	
Extensiones		**Paso**	**Condición**	**Caso de Uso**
Excepciones		5	El programa servidor no se conecta y manda un mensaje de error al usuario.	
Observaciones				

Figura A3.6. Especificación del caso de uso 6: Conectar Motor.

CASO DE USO	7	Desconectar Motor		
Descripción	El usuario desconecta el programa servidor y el motor Dynamixel al no requerir más acciones.			
Actores	Usuario			
Precondiciones	El programa servidor ha pasado satisfactoriamente por el proceso de conexión del motor (CU:1).			
Flujo normal	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente con el comando desconectar (defecto: *disc*) como parámetro asociado.		
	2	El programa servidor recibe el comando y lo encola.		
	3	El programa servidor ejecuta el comando desconectar.		
	4	El programa servidor finaliza la ejecución.		
	5	El programa servidor envía un mensaje al usuario indicando que la desconexión fue exitosa.		
Postcondiciones	El programa servidor cierra su conexión con el motor y no está disponible para aceptar más peticiones entrantes.			
Variaciones	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente interactivo y envía el comando de desconectar (defecto: *disc*).		
Extensiones	**Paso**	**Condición**	**Caso de Uso**	
Excepciones	5	El programa servidor no se desconecta y manda un mensaje de error al usuario.		
Observaciones				

Figura A3.7. Especificación del caso de uso 7: Desconectar Motor.

CASO DE USO	8	Calibrar Automáticamente	
Descripción	El usuario calibra el motor Dynamixel de forma automática para establecer un nuevo origen relativo en grados.		
Actores	Usuario		
Precondiciones	El programa servidor ha pasado satisfactoriamente por el proceso de conexión del motor (CU:1).		
Flujo normal	**Paso**	**Acción**	
	1	El usuario ejecuta el programa cliente con el comando calibrar automáticamente (defecto: *ac*) como parámetro.	
	2	El programa servidor recibe el comando y lo encola.	
	3	El programa servidor ejecuta el comando calibrar automáticamente.	
	4	El programa servidor gira el motor a cero grados (absolutos).	
	5	El programa servidor gira el motor a una velocidad constante entre 0 y 360 grados.	
	6	El usuario ejecuta el comando parar motor (defecto: *stop*) (CU: 10).	
	7	El programa servidor gira el motor periódicamente entre un rango de grados dependiente del anterior punto de parada.	
	8	El usuario ejecuta el comando parar motor (defecto: *stop*) (CU: 10).	
	9	El programa servidor almacena el nuevo origen relativo en grados y envía un mensaje al usuario.	
Postcondiciones	El programa servidor no está disponible para aceptar más peticiones entrantes que no se relacionen con la calibración.		
Variaciones	**Paso**	**Acción**	
	1	El usuario ejecuta el programa cliente interactivo y envía el comando de calibrar automáticamente (defecto: *ac*).	
Extensiones	**Paso**	**Condición**	**Caso de Uso**
Excepciones	9	El programa servidor presenta un error de escritura del nuevo valor y se lo comunica al usuario.	
Observaciones			

Figura A3.8. Especificación del caso de uso 8: Calibrar Automáticamente.

CASO DE USO	9	Calibrar Manualmente	
Descripción	El usuario calibra el motor Dynamixel de forma manual para establecer un nuevo origen relativo en grados.		
Actores	Usuario		
Precondiciones	El programa servidor ha pasado satisfactoriamente por el proceso de conexión del motor (CU:1).		
Flujo normal	**Paso**	**Acción**	
	1	El usuario ejecuta el programa cliente con el comando calibrar manualmente (defecto: *mc*) como parámetro.	
	2	El programa servidor recibe el comando y lo encola.	
	3	El programa servidor ejecuta el comando calibrar manualmente.	
	4	El programa servidor gira el motor a cero grados (absolutos).	
	5	El programa servidor gira el motor a una velocidad constante entre 0 y 360 grados.	
	6	El usuario ejecuta el comando parar motor (defecto: *stop*) para la primera fase de calibración (CU: 10).	
	7	El programa servidor queda a la espera de una acción de calibración del usuario.	
	8	El usuario ejecuta el comando parar motor (defecto: *stop*) para la segunda fase de calibración (CU: 10).	
	9	El programa servidor almacena el nuevo origen relativo en grados y envía un mensaje al usuario.	
Postcondiciones	El programa servidor no está disponible para aceptar más peticiones entrantes que no se relacionen con la calibración.		
Variaciones	**Paso**	**Acción**	
	1	El usuario ejecuta el programa cliente interactivo y envía el comando de calibrar manualmente (defecto: *mc*).	
Extensiones	**Paso**	**Condición**	**Caso de Uso**
	7	**El motor se encuentra en una posición inferior a la deseada**	**11**
	7	**El motor se encuentra en una posición superior a la deseada**	**12**
Excepciones	8	El programa servidor presenta un error de escritura del nuevo valor y se lo comunica al usuario.	
Observaciones			

Figura A3.9. Especificación del caso de uso 9: Calibrar Manualmente.

CASO DE USO	10	Parar Motor		
Descripción	El usuario para el motor durante el proceso de calibración pasando a la siguiente fase de este proceso.			
Actores	Usuario			
Precondiciones	El programa servidor ha pasado satisfactoriamente por el proceso de conexión del motor (CU:1) y además ha iniciado un proceso de calibración (CU: 6 o CU: 7).			
Flujo normal	**Paso**	**Acción**		
	1	El usuario acciona la tecla asociada al comando de parada del motor (defecto: *stop*).		
	2	El programa servidor recibe el comando.		
	3	El programa servidor ejecuta el comando y para el motor si está en movimiento.		
	4	El programa servidor estaba en la primera fase de calibración y pasa a la siguiente fase.		
Postcondiciones	El programa servidor pasa a la segunda fase de calibración o finaliza el proceso de calibración (según la variante 4).			
Variaciones	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente (interactivo o por comandos) y envía el comando de parada de motor de forma explícita (defecto: stop).		
	4	El programa servidor estaba en la segunda fase de calibración y finaliza el proceso de calibración.		
Extensiones	**Paso**	**Condición**	**Caso de Uso**	
Excepciones				
Observaciones				

Figura A3.10. Especificación del caso de uso 10: Parar Motor.

CASO DE USO	11	Incrementar Motor		
Descripción	El usuario incrementa la posición del motor durante la segunda fase del proceso de calibración manual.			
Actores	Usuario			
Precondiciones	El programa servidor ha pasado satisfactoriamente por el proceso de conexión del motor (CU:1) y se encuentra en la segunda fase del proceso de calibración manual.			
Flujo normal	**Paso**	**Acción**		
	1	El usuario acciona la tecla asociada al comando de incrementar la posición del motor (defecto: *up*).		
	2	El programa servidor recibe el comando.		
	3	El programa servidor ejecuta el comando e incrementa la posición del motor los grados definidos en la configuración.		
Postcondiciones	El programa servidor se mantiene en la segunda fase de calibración manual.			
Variaciones	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente (interactivo o por comandos) y envía el comando de incrementar la posición del motor de forma explícita (defecto: *up*).		
Extensiones	**Paso**	**Condición**	**Caso de Uso**	
Excepciones				
Observaciones				

Figura A3.11. Especificación del caso de uso 11: Incrementar Motor.

CASO DE USO	12	Decrementar Motor		
Descripción	El usuario decrementa la posición del motor durante la segunda fase del proceso de calibración manual.			
Actores	Usuario			
Precondiciones	El programa servidor ha pasado satisfactoriamente por el proceso de conexión del motor (CU:1) y se encuentra en la segunda fase del proceso de calibración manual.			
Flujo normal	**Paso**	**Acción**		
	1	El usuario acciona la tecla asociada al comando de decrementar la posición del motor (defecto: *down*).		
	2	El programa servidor recibe el comando.		
	3	El programa servidor ejecuta el comando y decrementa la posición del motor los grados definidos en la configuración.		
Postcondiciones	El programa servidor se mantiene en la segunda fase de calibración manual.			
Variaciones	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente (interactivo o por comandos) y envía el comando de incrementar la posición del motor de forma explícita (defecto: *up*).		
Extensiones	**Paso**	**Condición**	**Caso de Uso**	
Excepciones				
Observaciones				

Figura A3.12. Especificación del caso de uso 12: Decrementar Motor.

CASO DE USO	13	Rotar Motor		
Descripción	El usuario rota el motor a la posición especificada en grados (posición relativa).			
Actores	Usuario			
Precondiciones	El programa servidor ha pasado satisfactoriamente por el proceso de conexión del motor (CU:1) y el motor no se encuentra en proceso de calibración.			
Flujo normal	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente con el comando de rotación del motor (defecto: *rotate*) y una posición en grados como parámetros asociados.		
	2	El programa servidor recibe el comando y lo encola.		
	3	El programa servidor ejecuta el comando de rotación del motor.		
	4	El programa servidor finaliza la ejecución.		
	5	El programa servidor envía un mensaje al usuario indicando que la rotación fue realizada con éxito.		
Postcondiciones	El programa servidor se mantiene conectado y dispuesto a aceptar más comandos.			
Variaciones	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente con el comando de rotación del motor (defecto: *rotate*) como parámetro asociado, sin una posición en grados (el sistema lo entiende como *ir al origen*).		
	1	El usuario ejecuta el programa cliente interactivo y envía el comando de rotación del motor (defecto: *rotate*) junto a los grados deseados.		
	1	El usuario ejecuta el programa cliente interactivo y envía el comando de rotación del motor (defecto: *rotate*) sin a los grados deseados (el sistema lo entiende como *ir al origen*).		
Extensiones	**Paso**	**Condición**	**Caso de Uso**	
Excepciones				
Observaciones				

Figura A3.13. Especificación del caso de uso 13: Rotar Motor.

CASO DE USO	14	Rotar Motor Aleatorio		
Descripción	El usuario rota el motor a una o varias posiciones aleatorias en grados (posición relativa).			
Actores	Usuario			
Precondiciones	El programa servidor ha pasado satisfactoriamente por el proceso de conexión del motor (CU:1) y el motor no se encuentra en proceso de calibración.			
Flujo normal	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente con el comando de rotación aleatoria del motor (defecto: *rand*) y el número de rotaciones deseadas como parámetros asociados.		
	2	El programa servidor recibe el comando y lo encola.		
	3	El programa servidor ejecuta el comando de rotación aleatoria del motor.		
	4	El programa servidor finaliza la ejecución y espera el tiempo predefinido. Se vuelve al paso 3 hasta completar el número de rotaciones solicitadas.		
	5	El programa servidor envía un mensaje al usuario indicando todas las rotaciones realizadas.		
Postcondiciones	El programa servidor se mantiene conectado y dispuesto a aceptar más comandos.			
Variaciones	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente con el comando de rotación aleatoria del motor (defecto: *rand*) como parámetro asociado, sin el número de rotaciones deseadas (el sistema entiende que se desea una única rotación).		
	1	El usuario ejecuta el programa cliente interactivo y envía el comando de rotación aleatoria del motor (defecto: *rand*) junto al número de rotaciones deseadas.		
	1	El usuario ejecuta el programa cliente interactivo y envía el comando de rotación aleatoria del motor (defecto: *rand*), sin el número de rotaciones deseadas (el sistema entiende que se desea una única rotación).		
Extensiones	**Paso**	**Condición**	**Caso de Uso**	
Excepciones				
Observaciones				

Figura A3.14. Especificación del caso de uso 14: Rotar Motor Aleatorio.

CASO DE USO	15	Obtener Posición Motor		
Descripción	El usuario solicita al sistema (programa servidor) la posición actual del motor.			
Actores	Usuario			
Precondiciones	El programa servidor ha pasado satisfactoriamente por el proceso de conexión del motor (CU:1) y el motor no se encuentra en proceso de calibración.			
Flujo normal	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente con el comando de obtención de posición del motor (defecto: *pos*) como parámetro asociado.		
	2	El programa servidor recibe el comando y lo encola.		
	3	El programa servidor ejecuta el comando de obtención de posición del motor.		
	4	El programa servidor finaliza la ejecución.		
	5	El programa servidor envía un mensaje al usuario con la posición actual (relativa y absoluta) del motor.		
Postcondiciones	El programa servidor se mantiene conectado y dispuesto a aceptar más comandos.			
Variaciones	**Paso**	**Acción**		
	1	El usuario ejecuta el programa cliente interactivo y envía el comando de obtención de posición del motor (defecto: *pos*).		
Extensiones	**Paso**	**Condición**	**Caso de Uso**	
Excepciones				
Observaciones				

Figura A3.15. Especificación del caso de uso 15: Obtener Posición Motor.

Anexo 4. Patrones de diseño y *clean code*

A lo largo de los dos módulos se han empleado principalmente dos conceptos bastantes conocidos. Tenemos por un lado el uso del famoso patrón de diseño *Singleton* para algunas clases principales y por otro lado el uso del concepto de *Polimorfismo* para mejorar la comprensión y al mismo tiempo la extensión del código.
Se ha reservado este apartado para un anexo concreto debido al factor común que acarrea al estar presente en ambos módulos. En parte, esto es así porque se ha decidido abordar ambos módulos con algo de paralelismo, es decir, con ciertas regularidades estructurales y de comportamiento a nivel de código. La motivación de esta decisión es la coherencia que se busca dentro de nuestro marco de trabajo y dentro del proyecto global del sistema IPS, fomentando de esta manera la comprensión de ambos módulos en conjunto y por separado.

Singleton

El patrón de diseño *Singleton* (84) consiste en obtener como máximo una instancia de una misma clase. Esto conlleva a que, si una clase no ha sido instanciada, se obtendrá una instancia de la misma y, además, se almacenará esa instancia. Cuando se vuelva a solicitar una instancia, no se creará otra, sino que se devolverá esa misma.

Esto puede tener muchos objetivos, pero el principal es lograr un acceso global a una clase. Aunque se ha implementado esto en varias de las clases a lo largo del desarrollo, el origen principal de esta necesidad nos viene impuestos por el uso del hardware. Ejemplos de ello lo tenemos en la conexión con el hardware o en el uso de este (ya sea para comunicarnos y obtener datos o para ejecutar determinadas órdenes). En estos casos, nos interesa tener una única instancia de ello.

```
# Singleton instance of this class
__instance = None
__connected = False

def __new__(cls, *args, **kwargs):
    # When call EngineStatus() we assign to "self" the new object or...
    if cls.__instance is None:
        cls.__instance = super(EngineConnection, cls).__new__(cls)
        cls.__instance.__initialized = False
    # the existing one (Singleton)
    return cls.__instance

def __init__(self):
    # Only set initial values when instantiated first time
    if self.__initialized:
        return
    self.__initialized = True
    self.__default_port_keyword = Configuration.read_config_value("SETTINGS", "PORT_KEYWORD", "Configuration.conf")
    self.__port_keyword = self.__default_port_keyword
    self.__engine_port = 'None'  # No connection yet
    self.__engine_port_handler = None  # No connection yet
    self.__engine_packet_handler = None  # No connection yet
```

Figura A4.1. Se muestra el código Python para una de las clases que usa el patrón *Singleton*.

Explicado el concepto y el motivo de su uso, toca comentar que Python puede no parecer un lenguaje donde se implementen este tipo de conceptos de diseño de alto nivel. Sin embargo, se ha logrado una adaptación del patrón *Singleton* para este lenguaje, que mostramos en un ejemplo en la figura A4.1, donde se muestra la clase `ArduinoConnection`. Podíamos haber escogido cualquier otra, pero esto es indiferente.

En esta figura, podemos ver la variable de clase `__instance`. Por otra parte, podemos observar dos métodos especiales de Python `__new__` (usado en la creación de nuevas instancias) e `__init__` (empleado en la inicialización de instancias). Lo que se experimenta al tratar de obtener una instancia de esta clase es la valoración de una condición principal: si existe una instancia almacenada en la variable `__instance` omitimos cualquier proceso adicional y devolvemos esa instancia, en caso contrario, obtenemos una instancia de esta clase para luego devolverla.

Independientemente de si se crea o no una instancia nueva en el método `__new__`, se pasa obligatoriamente por el método `__init__` al tratar de obtener una instancia de esta clase. Esto quiere decir, que de alguna manera tenemos que controlar el aspecto de inicialización, ya que, aunque garanticemos devolver una única instancia de esta clase (método `__new__`), si la inicializamos cada vez que la devolvemos estaríamos eliminando cualquier estado de nuestra instancia global y única.

Es por ello por lo que, en la creación de la instancia, asignamos un valor *boolean* a esa instancia para identificar si ha sido o no inicializada. Evidentemente su valor inicial es *falso*, pero una vez pasa por el método `__init__` se conmuta a *verdadero* y se realiza de manera única la inicialización de la instancia. Durante el resto de las llamadas posteriores, este proceso se omitirá al valorar el estado de inicialización como *verdadero*.

Polimorfismo. Clean Code.

El polimorfismo (85), cuyo significado literal puede entenderse como *muchas formas*, representa en programación orientada a objetos el hecho de representar varios objetos diferentes bajo un mismo concepto, de forma que, a través de ese concepto, podemos acceder y llamar a métodos que implementan todas las clases hijas.

Puede parecer algo lioso, pero consideremos esto con un ejemplo sencillo. Tenemos dos clases de ejemplo, M*anzana* y *Pera*. Ambas presentan la propiedad *color* y encapsulan la acción *comer*. Normalmente, accederíamos a sus propiedades y métodos a través de un objeto de su propio tipo (*Manzana* o *Pera*, respectivamente).

Si nos damos cuenta, ambas clases podrían abstraerse en el concepto de *Fruta*. Imaginemos una clase *Fruta* con las mismas propiedades que presentan tanto *Manzana* como *Pera*: *color* y acción de *comer*. Si tuviéramos asignada una instancia de cualquier tipo de *Fruta* (*Manzana* o *Pera*) bajo la clase *Fruta*, podríamos acceder a su *color* o a la acción de *comer*, independientemente de qué tipo de objeto albergue (siempre que sea una *Fruta*). Este es el uso que se ha plasmado en nuestro desarrollo para el tema de las peticiones/comandos de usuario.

En ambos módulos, se ha implementado una clase abstracta nombrada

`AbstractRequest`. Esta clase presenta un nombre de petición/comando, una descripción de los parámetros admitidos y un método *run*. Independientemente de las diferencias entre comandos (nombres, parámetros y acciones diferentes), todos ellos presentarán estas características, la principal de ellas, ejecutar una acción diferente y personalizada (método *run*).

Sea cual sea la clase seleccionada (petición/comando), podremos ejecutar su método *run* y obtener resultados diferentes en cada caso. En símil al ejemplo anterior, sea una fruta roja o verde, podremos obtener su color.

La motivación de emplear esta estrategia ha sido mejorar la comprensión del código, reemplazando una larga lista de condicionales por una estructura de clases más coherente y ampliable donde podemos crear las clases que se requieran de forma que no vaya creciendo el mismo bloque de código de forma innecesaria (código espagueti) (86). Dicho de otro modo, con esto, reemplazamos un enorme *Switch* que compara peticiones del usuario para ejecutar el código asociado a la acción solicitada, por la selección de un objeto abstracto (`AbstractCommand`) que simplemente tenemos que ejecutar. Todo lo comentado lo podemos ver a un nivel más práctico en las figuras A4.2 y A4.3.

```
class RequestsList:
    __actions = {}

    @classmethod
    def init_actions(cls):
        commands = Configuration.read_config_section("COMMANDS", "Configuration.conf")
        cls.__actions[commands['CONNECT_COMMAND']] = ConnectionRequest()
        cls.__actions[commands['DISCONNECT_COMMAND']] = DisconnectionRequest()
        cls.__actions[commands['GET_DEGREES_COMMAND']] = GetEnginePositionRequest()
        cls.__actions[commands['RANDOM_ROTATE_COMMAND']] = RandomRotationRequest()
        cls.__actions[commands['ROTATE_COMMAND']] = RotationRequest()
        cls.__actions[commands['SET_CALIBRATION_VALUE_COMMAND']] = SetCalibrationRequest()
        cls.__actions[commands['AUTOMATIC_CALIBRATE_COMMAND']] = AutomaticCalibrationRequest()
        cls.__actions[commands['MANUAL_CALIBRATE_COMMAND']] = ManualCalibrationRequest()
```

Figura A4.2. Se muestra la definición de peticiones o comandos disponibles. Cada entrada tiene la palabra clave y la clase encargada de ejecutar dicha acción.

```
            selected_key = None
            selected_action = None
            for key, action in RequestsList.get_actions():
                if command == str(key):
                    print("Request: {} => {} {}\n".format(action.get_name(), command, parameters))
                    selected_key = key
                    selected_action = action
                    break

            # Socket busy sending response
            self.socket_busy = True

            # key found
            if selected_key is not None:
                sent = self.sock.sendto(selected_action.run(parameters).encode(), client)
                print('Return {} Byte-data to Client {}'
                      .format(sent, client))
```

Figura A4.3. Parte del código donde se selecciona y se ejecuta (*run*) el comando que coincide.