Contenido

[Índice de ilustraciones 3](#_Toc155292119)

[Índice de tablas 3](#_Toc155292120)

[Introducción 4](#_Toc155292121)

[Necesidad de los MiniRovers 4](#_Toc155292122)

[KVIs 4](#_Toc155292123)

[Desarrollo del Proyecto 5](#_Toc155292124)

[Escenario 5](#_Toc155292125)

[Funcionalidades 6](#_Toc155292126)

[Comunicación y tramas de datos 7](#_Toc155292127)

[MiniRovers 9](#_Toc155292128)

[Hardware 9](#_Toc155292129)

[Comunicaciones 10](#_Toc155292130)

[Máquina de estados 13](#_Toc155292131)

[Tramas y envío de datos 14](#_Toc155292132)

[Central 15](#_Toc155292133)

[Render 3D 31](#_Toc155292134)

[MQTT 32](#_Toc155292135)

[Chatbot de Telegram 32](#_Toc155292136)

[Main 33](#_Toc155292137)

[Start 34](#_Toc155292138)

[Coordinates 35](#_Toc155292139)

[Set Waypoint 36](#_Toc155292140)

[Download 37](#_Toc155292141)

[Emergency 38](#_Toc155292142)

[Node-RED 39](#_Toc155292143)

[Map 39](#_Toc155292144)

[Commands 40](#_Toc155292145)

[Sensors 43](#_Toc155292146)

[Emergency 44](#_Toc155292147)

[Conclusiones 47](#_Toc155292148)

# Índice de ilustraciones

[Fig. 1. Diagrama del Proyecto. 5](#_Toc155292086)

[Fig. 2. Código de comunicaciones 12](#_Toc155292087)

[Fig. 3. Código de Comunicaciones 12](#_Toc155292088)

[Fig. 4. Máquina de estados 13](https://uma365-my.sharepoint.com/personal/alba_correal_uma_es/Documents/Memoria.docx#_Toc155292089)

[Fig. 5. Diagrama de flujo. 16](#_Toc155292090)

[Fig. 6. Código de la función 28](#_Toc155292091)

[Fig. 7. Código de la función 31](#_Toc155292092)

[Fig. 8 Flujo 33](#_Toc155292093)

[Fig. . Flujo 34](#_Toc155292094)

[Fig. . Mensaje de Telegram 34](#_Toc155292095)

[Fig. . Mensaje de Telegram 35](https://uma365-my.sharepoint.com/personal/alba_correal_uma_es/Documents/Memoria.docx#_Toc155292096)

[Fig. . Flujo 35](#_Toc155292097)

[Fig. . Flujo 36](#_Toc155292098)

[Fig. . Mensaje de Telegram 36](#_Toc155292099)

[Fig. . Flujo 37](#_Toc155292100)

[Fig. . Flujo 37](https://uma365-my.sharepoint.com/personal/alba_correal_uma_es/Documents/Memoria.docx#_Toc155292101)

[Fig. . Flujo 38](#_Toc155292102)

[Fig. . Mensaje de Telegram 38](#_Toc155292103)

[Fig. . Flujo 38](#_Toc155292104)

[Fig. . Dashboard, Mapas 39](#_Toc155292105)

[Fig. . Dashboard, detalle de Mapa 39](https://uma365-my.sharepoint.com/personal/alba_correal_uma_es/Documents/Memoria.docx#_Toc155292106)

[Fig. . Flujo 40](#_Toc155292107)

[Fig. . Flujo 40](#_Toc155292108)

[Fig. . Dashboard. Comandos 40](https://uma365-my.sharepoint.com/personal/alba_correal_uma_es/Documents/Memoria.docx#_Toc155292109)

[Fig. . Flujo 41](#_Toc155292110)

[Fig. . Flujo 42](#_Toc155292111)

[Fig. . Flujo 42](#_Toc155292112)

[Fig. . Dashboard, sensores 43](https://uma365-my.sharepoint.com/personal/alba_correal_uma_es/Documents/Memoria.docx#_Toc155292113)

[Fig. . Flujo 43](#_Toc155292114)

[Fig. . Dashboard, emergencias 44](https://uma365-my.sharepoint.com/personal/alba_correal_uma_es/Documents/Memoria.docx#_Toc155292115)

[Fig. . Dashboard. Notificación de emergencia 44](https://uma365-my.sharepoint.com/personal/alba_correal_uma_es/Documents/Memoria.docx#_Toc155292116)

[Fig. . Flujo 45](#_Toc155292117)

[Fig. . Flujo 46](#_Toc155292118)

# Índice de tablas

[Tabla 1. Asociación de las funciones implementadas 7](#_Toc155291578)

[Tabla 2. Tramas de comunicación 8](#_Toc155291579)

[Tabla 3. Constructor 15](#_Toc155291580)

# Introducción

## Necesidad de los MiniRovers

La exploración espacial ha sido siempre una de las aspiraciones más desafiantes de la humanidad. Muchos son los problemas por abordar en este sector, que confía inversiones millonarias y decenas de años de investigación a una sola máquina de carga elevada, con una aparamenta compleja y, en ocasiones, capacidad de comunicación incierta. Es por ello que cada uno de los pasos de este proceso supone un reto para la tecnología contemporánea, siendo el sector espacial, junto con el militar, los que disponen de la maquinaria tecnológicamente más avanzada de la industria.

Dada la complejidad de estos mecanismos robóticos, y su importancia para la ciencia, conceptos como la modularidad o la autonomía del control de procesos cada vez son más populares, cobrando sentido la existencia de los MiniRovers: robots más pequeños, con tareas más específicas, pero interconectados entre sí.

Sin embargo, una definición correcta de estas matrices robóticas es una tarea muy compleja, y arroja varias preguntas: ¿Cuál debe ser la jerarquía de los MiniRovers? ¿Cómo definir qué datos se enviarán? ¿Cómo interactuarán los científicos con estas redes?

Como Prueba de Concepto, ofreceremos una posible definición de este conjunto, profundizando en algunas de las características clave que se podrían implementar.

## KVIs

Los elementos principales que dan valor al proyecto y que se quieren demostrar son:

* Redundancia: en el sector espacial, los fallos de comunicación o de hardware son muy frecuentes. Transmitir los datos por más de un protocolo simultáneamente es una necesidad casi imprescindible.
* Modularidad: reducir el peso y costo de los robots, así como facilitar su independencia es uno de los principales valores del proyecto
* Edge-computing: la descentralización del procesamiento, cuando se tratan de procesos tan complejos como la navegación o el análisis de datos de un sensor muy específico, hace necesario dotar a los rovers de autonomía de procesamiento.
* Optimización de recursos: No obstante, si la carga computacional es demasiado grande, debe tener la posibilidad de derivar ese dato a una capa superior, garantizando que el hardware siempre sea ad-hoc y no exceda las necesidades del dispositivo a controlar.

# Desarrollo del Proyecto

Hemos desarrollado un caso compacto donde la simplicidad se alinea con la diversidad, con el fin de abordar una amplia cantidad de situaciones reales. Para una profunda comprensión de la casuística, proporcionaremos una descripción detallada del escenario.

## Escenario

Hay 4 agentes principales en nuestro escenario:

* Rover Sensor: recolecta datos del entorno.
* Rover actuador: extrae muestras del suelo en exploración.
* Ordenador central: secuencia tareas, sirve como punto de recarga y zona de extracción de muestras. También puede analizar datos complejos.
* Red satelital: recibe datos de sensores y de la computadora central, los almacena en la nube y proporciona a los científicos una interfaz de usuario para interactuar tanto con los rovers como con los propios datos.

En un contexto como este, la interconexión es clave. Todos los elementos están conectados entre sí para diferentes propósitos y con protocolos que se adaptan a sus necesidades. Adicionalmente, cada subsistema está provisto de un hardware de procesamiento optimizado para las funciones a realizar (Fig.1).

Diagrama, Esquemático

Descripción generada automáticamente

Fig. 1. Diagrama del Proyecto.

## Funcionalidades

#### Emergency data sharing

En caso de emergencia, los rovers se comunicarían de manera directa por ESP-NOW, un protocolo rápido, de medio alcance y nativo de estas placas. Para demostrarlo, hemos supuesto que, si uno de los dos detecta un terremoto, se lo comunica al otro mediante ESP-NOW para que ambos cancelen su tarea actual y entren en estado de emergencia, comunicándose así con la central vía LoRa, la cual actualiza sus tareas y les ordena dirigirse a la central inmediatamente.

#### Remote control

Para enviar y recibir datos de navegación, los rovers se comunican con la central por LoRa, más apto para distancias largas. Con este propósito, la central dispone de un secuenciador de tareas, gracias al cual puede enviar a un rover determinado una ubicación de destino, de manera que, cuando esté al alcance (comprobando que sus coordenadas GPS corresponden con las de la ubicación enviada), se ejecute el proceso autónomo del robot. La central, a su vez, recibe datos de la ubicación y estado de los rovers, permitiéndole actuar en caso de timeout y monitorizar el proceso.

#### Triggers

Hay casos específicos donde es necesario un control adicional a la navegación. Por ello, los rovers disponen de varios desencadenantes de emergencia:

* Low Battery: la batería del rover es baja. Se actualiza el estado del mismo a “Emergency” y se cancela la tarea actual, a la espera de que la central actualice la secuencia y lo envíe de vuelta al punto de carga y extracción.
* Sample Overflow: se da cuando el rover actuador ha llenado todos los recipientes de extracción de muestras disponibles. Esta emergencia se aborda exactamente igual que la anterior.

Para fomentar el concepto de interconexión, existe un flujo adicional que permite que, pasado un umbral determinado de los sensores del Rover Sensor, se genere un desencadenante (como todos, enviado por LoRa) que permita al secuenciador de tareas crear una tarea que se envíe al Rover Actuador para extraer una muestra de suelo en esa ubicación, pues se considera una ubicación interesante desde el punto de vista científico.

#### Specific data collection

En ocasiones, la carga computacional de algunos datos hace complejo su envío directo a la nube o interfaz de usuario. La extracción de muestras, por ejemplo, necesita de un análisis químico y espectroscópico que arroje luz sobre la composición y naturaleza del suelo. Actualmente, robots como el Perseverance no disponen de la tecnología necesaria para hacer todos los análisis in-situ, obligando a que sus muestras se analicen una vez el rover retorne a la Tierra. Si dispusiéramos de una central con la habilidad de procesar esos datos, no sería necesario devolverlo a nuestro planeta, pues los datos post-procesados se publicarían en la red y podrían visualizarse de manera remota.

Para ejemplificarlo, hemos generado un Render 3D del planeta en exploración, donde se pueden ubicar en tiempo real los Rovers. Naturalmente, esa información no puede ser procesada directamente por la nube, sino que debe ser la central quien analice y genere el Render.

#### Data aggregation

El estado de los rovers y sus mediciones no es la única información de relevancia científica. La secuenciación de tareas o los Renders como el descrito anteriormente también tienen un valor a analizar. Es por eso por lo que la central envía a la nube por MQTT la tabla donde recoge la secuencia de tareas y los datos adicionales que genera, como dicho Render.

#### Real-time monitoring

Simultáneamente, toda la información sobre los sensores, estado de los robots, emergencias encontradas, etcétera, se envía por MQTT a la base de datos en la nube. Estos datos se pueden visualizar en una interfaz creada con Node-RED.

#### User prompts & Firmware updates

Aunque la autonomía de este sistema es muy alta, es necesario que los científicos dispongan de las herramientas necesarias para que los rovers estén a su servicio. Por ello, nuestra nube de Node-RED también controla un chatbot a través del cual los científicos pueden enviar consignas, leer las coordenadas o exportar datos. Se prevé la implementación de actualizaciones remotas de firmware, pero no se ha desarrollado en el proyecto actual.

Curiosamente, todos los protocolos descritos previamente pueden utilizarse en el espacio. La comunicación WiFi en estos entornos es cada vez más habitual y, por lo que respecta a este proyecto, tan solo faltaría disponer de una antena y un conmutador de protocolo CCSDS para poder transmitir la información de nuestro sistema a la Tierra, cubriendo así el ciclo completo.

Como síntesis de lo explicado anteriormente, se muestra una tabla que relaciona la funcionalidad con el caso implementado.

|  |  |  |
| --- | --- | --- |
| Sender à Receiver | Function | Showcase |
| Rover ßà Rover | Emergency Data | Earthquake (accelerometer) |
| Rover àCentral | Triggers | * Low Battery * Sample Overflow * Interesting waypoint |
| Remote Control | * Coordinates * Status |
| Specific Data | * Latitude * Longitude   (Central creates a 3D render) |
| Central àRover | Remote Control | * Desired Coordinates * Task information |
| Rover àSatellite | Real-time monitoring | * Sensor Data * Rover status |
| Satellite àRover | Firmware updates | * FOTA   (not implemented) |
| Central àSatellite | Data aggregation | * Rover Tasks data * 3D render output |
| Satellite àCentral | User prompts | Scientists’ waypoints |

Tabla 1. Asociación de las funciones implementadas

## Comunicación y tramas de datos

Para conseguir implementar todas las funciones, nos hemos basado en cinco tramas de datos de una arquitectura similar, que se trasmitirán simultáneamente por los distintos canales de comunicación dispuestos arriba, y que serán escuchados por el receptor pertinente. De este modo,

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Usage | Sender 🡪 Receiver | Fields & Content |
| II15/Rovers[[1]](#footnote-2) |  | SensorRover  ActuatorRover | "Sender":"SensorRover” / “ActuatorRover ",  "Timestamp”: integer,  "Latitude”: double  "Longitude”: double  "Status": "Task finished. Available”/”Task ongoing. Not available” / “Emergency state" |
| Remote control & Triggers | Central |
| Real-time monitoring: Map | Node-RED |
| II15/Sensors |  | SensorRover | "Sender": "SensorRover",  "Timestamp": integer,  "Latitude": double  "Longitude”: double  "Altitude": float  "Temperature": float  "Humidity": float  "Pressure": float  "Radiation": float  "Status": “ “/”Interesting location. Humidity Pressure Radiation” |
| Triggers: interesting waypoint | Central |
| Real-time monitoring: sensor data | Node-RED |
| II15/Emergency |  | SensorRover  ActuatorRover | "Sender":"SensorRover” / “ActuatorRover ",  "Timestamp” : integer,  "Latitude” : double  "Longitude" : double  "Status": "Earthquake” / “Low battery” / “Sample overflow" “Timeout” |
| Real-time monitoring: Emergency notification | Node-RED |
| Emergency data sharing | SensorRover  ActuatorRover |
| Command[[2]](#footnote-3) |  | Central | "Sender":"SensorRover”/”ActuatorRover”/”Auto”/”Central”/”User",  "RoverId":"SensorRover/ActuatorRover",  "Timestamp”: integer,  "Latitude”: double  "Longitude”: double  "Priority": 1,  "Timeout": integer |
| Remote control: waypoint sending | SensorRover  ActuatorRover |
| II15/Commands |  | Central | "Sender":"SensorRover”/”ActuatorRover”/”Auto”/”Central”/”User",  "RoverId":"SensorRover/ActuatorRover",  "Timestamp”: integer,  "Latitude”: double  "Longitude”: double  "Priority": 1 / 2 / 3…  "Timeout": integer |
| Commands table | Node-RED |
| II15/Waypoint |  | Node-RED | "Sender":"SensorRover”/”ActuatorRover”/”Auto”/”Central”/”User",  "RoverId":"SensorRover/ActuatorRover",  "Timestamp”: integer,  "Latitude”: double  "Longitude”: double  "Priority": 1,  "Timeout": integer |
| User prompts: scientists’ waypoints | Central |

Tabla 2. Tramas de comunicación

En síntesis, Rovers transmite el estado de los rovers y sus coordenadas, Sensors el valor de los sensores y si se encuentra alguna ubicación de interés, Emergency la ubicación y motivo de la emergencia, Commands las nuevas ubicaciones de destino y el creador de estas, y Table exporta la tabla de comandos completa. De manera adicional, se envía el archivo html del render por MQTT de la Central a Node-RED.

## MiniRovers

Nuestra prueba consiste en dos MiniRovers similares en morfología, pero de distinta funcionalidad.

### Hardware

Los Rovers cuentan con sensores, actuadores, módulos de comunicación y un multiplexor. Todos ellos son controlados por una ESP32, que es el núcleo del rover.

#### Sensores

En el caso concreto del sensor, se han empleado varios módulos[[3]](#footnote-4) para medir distintos valores ambientales que permitan definir umbrales de interés[[4]](#footnote-5). Dichos módulos son:

* DHT11: sensor de temperatura y humedad que utiliza un sensor capacitivo de humedad y un termistor para medir el aire que circula .
* BMP180: sensor de presión que utiliza un transductor piezorresistivo para convertir la presión en una señal eléctrica. Esta señal es procesada por un circuito de acondicionamiento para obtener la lectura de presión. El umbral definido es de 1033 hPa.
* MPU-6050: sensor inercial que utiliza un conjunto de microelectromecanismos (MEMS) para medir la velocidad angular (giroscopio) y la aceleración lineal (acelerómetro). Estos MEMS están conectados a un circuito de acondicionamiento que traduce los movimientos físicos en señales eléctricas. El umbral definido es una aceleración superior a 12 m/s2.
* GUVA-S12SD: sensor de radiación ultravioleta que utiliza un fotodiodo sensible a la radiación UV. Cuando la radiación UV incide sobre el fotodiodo, se generan corrientes eléctricas proporcionales a la intensidad de la radiación. Estas corrientes son amplificadas y convertidas en señales digitales mediante un circuito de acondicionamiento.

#### Actuadores

Para mostrar una demo, se ha utilizado un montaje simple de dos motores con ruedas y el módulo L298N, un controlador de motor dual que utiliza puentes H para controlar la dirección y velocidad de los motores de corriente continua o paso a paso. Permite control bidireccional y regulación de la velocidad mediante señales PWM.

Para ejemplificar la extracción de muestras, se emplea un led que se encenderá cuando la extracción esté en curso.

#### Módulos de Comunicación

* Para obtener la ubicación del rover en coordenadas (latitud y longitud), se ha empleado el NEO-6M: módulo GPS que contiene un receptor GPS y un circuito de procesamiento. El receptor GPS se comunica con múltiples satélites para obtener señales de posicionamiento. Estas señales son procesadas por el circuito, que determina la ubicación geográfica en función de la información recibida de los satélites.
* La transmisión por LoRa ha sido posible gracias al módulo SX1278, un transceptor de largo alcance que recibe datos por LoRa y los transmite a la ESP gracias al protocolo SPI.
* Para evitar la saturación del puerto I2C de la placa, se ha empleado un multiplexor I2C que conmute el sensor de lectura.

### Comunicaciones

Como se muestra en el diagrama, nuestro sistema emplea diversos protocolos simultáneos, cada uno con un fin específico. Esto ha supuesto un reto técnico, pues la ESP32 sólo dispone de una antena WiFi (por la cual hay que enviar parte de la información por dos protocolos diferentes) y los tiempos de latencia son críticos (ya que el rover debe estar escuchando casi en tiempo real a muchos emisores). A continuación, se describen los protocolos usados, así como los elementos más destacables del código.

#### ESP-NOW

La comunicación básica con ESP-NOW emplea una MAC, que se puede obtener fácilmente con funciones de la librería. Dicha MAC, introducida en Esp\_now\_peer\_info\_t, devuelve todos los datos necesarios para una correcta comunicación entre dos placas ESP32.

Se han definido dos funciones de callback, una para el envío y otra para la recepción de datos, que facilitan la labor de identificar fallos de transmisión de datos. En la callback de recepción, se ha introducido una función de memcpy(), que almacena los datos recibidos en una variable de la placa.

A continuación, basta con inicializar el módulo y, en el bucle principal del programa, enviar los datos pertinentes con Esp\_now\_send(). Nótese que, para el envío en ESP-NOW, es necesario indicar la longitud del mensaje.

#### WiFi

El protocolo WiFi, realmente, se emplea para usar MQTT. Dado que este formato de comunicación se ha usado con asiduidad a lo largo de la asignatura, no se explicará en detalle el código asociado.

No obstante, sí cabe destacar que, en WiFi, existen varias configuraciones de conexión:

* Station: permite que el dispositivo se conecte a una red WiFi existente como cliente.
* Access Point: permite que el dispositivo cree su propia red WiFi a la cual otros dispositivos se pueden conectar.
* APSTA: fusiona ambas funcionalidades simultáneamente.

Dada la naturaleza concurrente del proyecto, para hacer funcionar MQTT y ESP-NOW simultáneamente era necesario emplear el modo APSTA.

##### HANDLE NETWORK

El punto más interesante de este binomio de protocolos es que, como bien se ha dicho previamente, sólo se dispone de una antena WiFi para transmitir dos protocolos simultáneos. Eso implica que, si la conexión del protocolo WiFi fallase, automáticamente se perdería la conexión en ESP-NOW. Para evitarlo, se ha definido un flujo para habilitar ESP-NOW sin WiFi, y sin tener que reiniciar la placa.

  void handle\_network(void) {

      if (WiFi.status() != WL\_CONNECTED) {

          if (millis() - CONNECT\_TIMER >= RETRY\_TIMEOUT) {

              CONNECT\_TIMER = millis();

              init\_wifi();

          } if (WiFi.status() != WL\_CONNECTED) {

              enable\_esp\_now();

          }

      } if (!mqtt\_client.connected()) {

          if (millis() - CONNECT\_TIMER >= RETRY\_TIMEOUT) {

              CONNECT\_TIMER = millis();

              init\_mqtt();

                  enable\_esp\_now();

              } else {

                  mqtt\_client.loop();

              }

          }

      }

  }

Fig. 2. Código de comunicaciones

enable\_esp\_now() modifica el modo de configuración a Station, para habilitar solo el protocolo en cuestión.

#### LORA

LoRa (Long Range) es un protocolo de red de área extensa de baja potencia diseñado para la comunicación a larga distancia entre dispositivos. A diferencia del Wi-Fi, LoRa opera en bandas de frecuencia no licenciadas, minimizando la interferencia en aplicaciones IoT. Es por ello, que se ha usado como comunicación principal entre la central y los rovers. Además, es muy sencillo de programar en ESP32. El código empleado se muestra a continuación.

* Init\_lora() define pines y begin
* Lora\_recv\_callback(lora.parsePacket())
  + Mensaje no vacío
  + Address correcto

  void send\_msg\_lora(String msg) {

      LoRa.beginPacket();

      LoRa.write(RECEIVER\_ADDR);

      LoRa.write(LOCAL\_ADDR);

      LoRa.write(MSG\_COUNT);

      LoRa.write(msg.length());

      LoRa.print(msg);

      LoRa.endPacket();

      MSG\_COUNT++;

  }

Fig. 3. Código de Comunicaciones

### Máquina de estados

Diagrama, Esquemático

Descripción generada automáticamenteEl proceso seguido por el robot cuenta con una máquina de estados de 6 etapas, con 3 desencadenantes.

Fig. 4. Máquina de estados

En el flujo principal, el robot comienza en el estado “FINISHED” y, si recibe un mensaje de la central (detectado cuando contiene el campo “Priority”), el desencadenante EXECUTE toma el valor “true”, permitiendo el avance de la máquina a “MOVING”, donde se desplaza al punto solicitado. Una vez haya finalizado la navegación (DONE activado), pasa a tomar la medida o extraer la muestra en “MEASURING”, que, al acabar, vuelve a “FINISHED”.

El flujo secundario se da si sucede alguno de los casos de emergencia o desencadenante asociado (“Low Battery”, “Sample Overflow” y “Earthquake”). Así, si en alguno de los casos se da un caso de emergencia, el rover que detecta el fallo lo envía a la central, que actualiza las tareas y lo envía de vuelta a origen. Una vez llegado, e independientemente de la naturaleza de la emergencia, el rover carga y, en el caso del ActuatorRover, descarga sus muestras. Una vez acabado todo el proceso, está listo para salir de nuevo.

Existen dos desencadenantes adicionales para el control:

* SEND: usado para enviar emergencias por mqtt solo cada 5 segundos
* FROM\_PARTNER: usado para evitar que el estado de emergencia entre en bucle y que los rovers se envíen indefinidamente que hay un terremoto. Esto se debe a que el tipo de emergencia en cuestión se comunica, además de a la central vía LoRa, al resto de los rovers vía ESP-NOW. Dado este caso, los rovers están programados para ignorar cualquier emergencia que no sea del tipo “Earthquake”. Si un rover entra en emergencia a causa de movimiento sísmico, provocará que el resto de rovers lo haga también, pero el resto no transmitirá vía ESP-NOW al resto de rovers dicha emergencia, a menos que sea detectada por ellos mismos.

Para simular la casuística descrita, a falta de recorrer la distancia suficiente con los rovers para generar una consigna, y a falta también de tener un sensor de batería, se ha empleado una función que usa el botón de la ESP32 para generar los flags, button\_handler, que en caso de click, envía un “DONE”, en caso de click largo, supone un “Low Battery”, y en caso de doble click, genera un “EXECUTE”.

### Tramas y envío de datos

#### II15/Rovers

Los campos asociados a este topic se mandan por LoRa y MQTT cada 5 segundos.

* Sender será SensorRover o ActuatorRover según quién transmita la información a Central.
* Timestamp tomará el valor de sincronización inicial, más los milisegundos pasados desde entonces, obtenidos con la función millis().
* Latitude y Longitude corresponderán a los datos del GPS.
* Status corresponderá con “Task finished…” si está en el estado FINISHED, “Task ongoing…” si está en MOVING o MEASURING y “Emergency State” si está en EMERGENCY, ya sea por “Low Battery”, “Sample Overflow” o “Earthquake”.

#### II15/Sensors

Los campos asociados a este topic se mandan por LoRa y MQTT cada vez que se llega a la ubicación de destino.

* Sender siempre será SensorRover.
* Timestamp se define de la misma forma que en el topic anterior.
* Latitude, Longitude, Altitude, Temperature, Humidity, Pressure y Radiation corresponderán a los datos de los sensores.
* “Status” se envía vacío si la ubicación no es de interés científico, y con “Interesting location.” más el nombre del campo del que se haya obtenido un valor interesante en caso de que algún valor supere un umbral. Es compatible con valores interesantes de distintos sensores.

#### II15/Emergency

Los campos asociados a este topic se enviarán por LoRa o MQTT cada vez que se dé un estado de emergencia.

* Sender será SensorRover o ActuatorRover según quién transmita la información a Central.
* Timestamp se define igual que en los topics anteriores.
* Latitude y Longitude también.
* Status tomará el valor del tipo de emergencia encontrado (“Low Battery”, “Sample Overflow” o “Earthquake”).

## Central

En primer lugar, es necesario inicializar mqtt, mongo y el puerto serie, así como declarar los diccionarios, que son:

* Roversinfo: información de los rovers recibida del puerto serie
* newroverinfo: flag
* Status: permite identificar si la información recibida es nueva o no
* Sensorcmd: si se trata de un mensaje donde el sensor mande un comando al rover
* Ack: necesario para evitar que se sobreescriba el dato si llega en menos de 5 segundos respecto al anterior

A continuación, se declaran las clases de comunicación y sus respectivas funciones

* HandleMqtt: con todas las funciones de conectividad mqtt
* HandleSerial: con todas las funciones de comunicación serial. Para poder leer por puerto serie, hay que almacenar los datos byte a byte hasta que se encuentra un “\r”. Estos datos se almacenan en una variable y, según la llegada, se activa o no ack para evitar que se sobreescriban los datos.

No obstante, la clase más relevante de todas es la SequencePlanner, que contiene todas las funciones implementadas con relevancia estructural.

#### Constructor \_\_init\_\_()

El constructor declara todos los elementos significativos de la clase. Concretamente, estos son:

|  |  |  |  |
| --- | --- | --- | --- |
| Object type | Name | Fields | Usage |
| List | Sequencer | TaskId, priority, latitude, longitude,roverId, sender, timeout, timestamp, distance | Task manipulation and sequencing |
| roversId | roverId | ID of current rovers |
| Explored | roverId, latitude, longitude, last\_time\_explored,timestamp) | All set of current explored waypoints |
| Locations | roverId,current latitude, current longitude | Current location of each rover |
| Variable | Lat\_max  Long\_max  Lat\_home  Long\_home |  |  |
| Direction | +1 / -1 | Set sign of next auto command coordinate direction. (+ means North & East) |
| Usercmd | Sender, RoverId, latitude, longitude, timestamp | Existing user command for a rover |
| New\_usercmd | True/false | Flag |
| Sensorcmd | "Sender, "Timestamp": integer, "Latitude", "Longitude, "Altitude", "Temperature", "Humidity", "Pressure", "Radiation", Status | Existing sensor command for a rover |
| New\_sensorcmd | True/false | Flag |
| Autocmds | TaskId, Priority, latitude, longitude, RoverID,Sender, Timeout,Tiemstamp,Distance | Existing automatic command for a rover |
| New\_autocmds | True/false | Flag |
| Pending | True/false | Pending tasks to send flag |
| Task\_id | integer | Counter for new task generation |
| SerialModule | HandleSerial object | Serial communication |
| Mqtt | HandleMqtt object | MQTT communication |

Tabla 3. Constructor

Además de esto, en el constructor se declararán las funciones de la clase, pero estas se detallarán en apartados posteriores.

#### Funciones

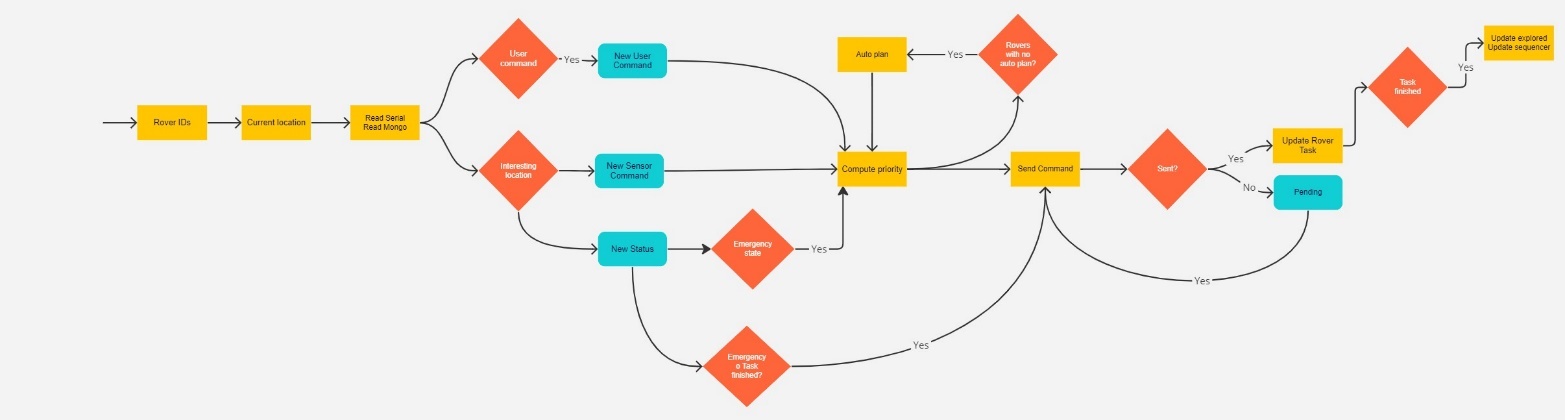
Para una mejor comprensión de las funciones, es preciso recurrir a un diagrama de flujo, que permita discernir aquellas principales de las secundarias, y cómo interactúan entre sí. 

Fig. 5. Diagrama de flujo.

En primer lugar, se obtienen los identificadores de los rovers con la función Updaterovers(), que actualiza la lista desde mysql. También se obtiene la ubicación actual gracias a process\_location(), que obtiene la ubicación real de los rovers con un query a MongoDB y la vuelca en la lista Location.

Después, con get\_data(), se lee el puerto serial, así como el comando de usuario de la colección. Internamente, se ejecuta la función de compute\_data(), donde se dan tres casos:

1. Nuevo comando de usuario: obtenido previamente de Telegram por el topic de MQTT II15/Waypoint, se activa el flag correspondiente a este caso.
2. Nuevo comando del sensor: se extrae del campo Status si este contiene “Interesting Waypoint”, en cuyo caso se activa el flag pertinente.
3. Nuevo valor del estado: si no es ninguno de los anteriores, se asume que no hay un comando y que se trata simplemente de una actualización.

En todos los casos, se guarda el valor en sus variables correspondientes.

Para calcular la prioridad y asignarla al secuenciador de tareas, se utiliza la función compute\_priority(). Dentro de esta función, se recogen varios casos:

1. Estado de emergencia: nótese que, cuando llega un nuevo estado de los rovers, éste contiene un campo “status” donde también se registran las emergencias. En caso de tenerla, se sitúa la tarea al comienzo de la lista, y se actualiza la prioridad
2. Nuevo comando de usuario: si en la función anterior se ha activado el flag de comando de usuario (new\_usercmd), se inserta detrás del último comando de emergencia y, en relación al resto de comandos de usuario, de acuerdo con la distancia respecto a la tarea actual (si la hubiera) o las coordenadas actuales (si estuviera libre)
3. Nuevo comando del sensor: si el flag de “new\_sensorcmd” está activo, se sitúa este comando delante de los generados automáticamente, pero en orden de distancia respecto al punto final de la tarea actual (si la hubiera) o las coordenadas actuales (si estuviera libre). Para determinar qué robot enviar a esa ubicación, se ejecuta la función compute\_roverid(), pues esto depende de qué dato ha superado el umbral.
4. # ------------------------------------------- COMPUTE PRIORITY ----------------------------------------------------
5. def compute\_priority]
6. """
7. This method computes tasks priority of all Rovers
8. PRIORITY ORDER: EMERGENCY,USER COMMAND, AUTOMATIC COMMAND
9. """
10. # ======================================= ROVERS PRIORITY COMPUTATION ==========================================
11. # --------------------------------------------------------------------------------------------------------------
12. # ID | Priority |  Latitude  | Length  |  Rover ID    |  timeout  |  Sender   |   Timestamp |    Distance
13. # --------------------------------------------------------------------------------------------------------------
14. # 5  |    1     |    25.2    |   0.2   |   Sensor     |  125 sec  |   User    |   166598    |       34
15. # --------------------------------------------------------------------------------------------------------------
16. # 2  |    2     |    25.2    |   0.2   |   Sensor     |  125 sec  |   User    |   177509    |       67
17. # --------------------------------------------------------------------------------------------------------------
18. # 0  |    3     |    25.2    |   0.2   |   Sensor     |  125 sec  |   Auto    |   189067    |       23
19. # --------------------------------------------------------------------------------------------------------------
20. # 3  |    1     |    25.2    |   0.2   |   Actuator   |  125 sec  |   User    |   166598    |       123
21. # --------------------------------------------------------------------------------------------------------------
22. # 6  |    2     |    25.2    |   0.2   |   Actuator   |  125 sec  |   User    |   177509    |       234
23. # --------------------------------------------------------------------------------------------------------------
24. # 1  |    3     |    25.2    |   0.2   |   Actuator   |  125 sec  |   Auto    |   189067    |       56
25. # --------------------------------------------------------------------------------------------------------------
27. # --------------------------------------- PRIORITY 1: RETURN HOME ------------------------------------------------
29. # Check if new critical status is received
30. if (self.new\_status and self.status['Status']=='Emergency state'):
31. # Inform
32. print("Info: Rover " + str(self.status['Sender']) + " is in emergency. Computing task as most prioritary...\n")
33. # Current rover location
34. index = [i for i,x in enumerate(self.Locations) if (x[0] == self.status['Sender'])]
36. lat\_now = self.Locations[index[0]][1]
37. lng\_now = self.Locations[index[0]][2]
38. # Task ID
39. self.task\_id =  self.task\_id + 1
40. # Return home command
41. id        =  self.task\_id                                                              # (0) Task ID
42. priority  =  1                                                                         # (1) Most priority
43. latitude  =  self.home\_lat                                                             # (2) Latitude (Home)
44. longitude =  self.home\_long                                                            # (3) Longitude (Home)
45. roverid   =  self.status['Sender']                                                     # (4) Rover ID
46. sender    =  'Central'                                                                 # (5) Sender
47. timeout   =  self.compute\_timeout([[lat\_now, lng\_now],[latitude, longitude]])          # (6) Timeout
48. timestamp =  self.status['Timestamp']                                                  # (7) Timestamp
49. distance  =  0                                                                         # (8) Distance (not needed)
51. # Command structure
52. command = [id, priority, latitude, longitude, roverid, sender, timeout, timestamp, distance]
53. print("Info: Adding return home command to Sequencer: " + str(command)+'\n')
54. try:
55. # Find actual most prioritary command of rover in Sequencer
56. index = [i for i,x in enumerate(self.Sequencer) if (x[4] == roverid)][0]
57. self.Sequencer.insert(index, command)
59. except:
61. # Add at the end of Sequencer
62. self.Sequencer.append(command)
63. # Update priority
64. self.update\_priority(roverid, priority,delete=False)
65. # Inform
66. print("Info: Return home command added to Sequencer\n")
67. # -----------------------------------------------------------------------------------------------------------


71. # ------------------------------------------ PRIORITY 2: USER COMMAND ----------------------------------------
73. #Check if a new user command is received
74. if (self.new\_usercmd):
76. # Inform
77. print("Rover " + str(self.usercmd['RoverId']) + " received a user command. Computing priority of task...\n")
78. # Task ID
79. self.task\_id =  self.task\_id + 1
80. # User command
81. id         =  self.task\_id                                                                   # (0) Task ID
82. priority   =  None                                                                           # (1) Priority
83. latitude   =  self.usercmd['Latitude']                                                       # (2) Latitude
84. longitude  =  self.usercmd['Longitude']                                                      # (3) Longitude
85. roverid    =  self.usercmd['RoverId']                                                        # (4) Rover ID
86. sender     =  self.usercmd['Sender']                                                         # (5) Sender
87. timeout    =  None                                                                           # (6) Timeout (Computed later)
88. timestamp  =  self.usercmd['Timestamp']                                                      # (7) Timestamp
89. distance   =  0                                                                              # (8) Distance (Computed later)
90. # Computing priority based on distance to current rover task location
91. try:
92. # ID of current rover task
93. current\_id  = [d for d in self.RoversTask if (d[0] == self.usercmd['RoverId'])][0][1]
94. # Index of current task in Sequencer
95. index = [i for i,x in enumerate(self.Sequencer) if (x[0] == current\_id)][0]
96. # Compute distance to current task location
97. lat\_now    =  self.Sequencer[index][2]                                                   # Current task latitude
98. long\_now   =  self.Sequencer[index][3]                                                   # Current task longitude
99. except:
100. # Get last location
101. lat\_now   = [d[1] for d in self.Locations if (d[0] == self.usercmd['RoverId'])][0]       # Current latitude
102. long\_now  = [d[1] for d in self.Locations if (d[0] == self.usercmd['RoverId'])][0]       # Current longitude
103. # Compute timeout and distance
104. timeout    =  self.compute\_timeout([[lat\_now, long\_now],[latitude, longitude]])               # (6) Timeout
105. distance   =  self.compute\_distance([[latitude, longitude],[lat\_now, long\_now]])              # (8) Distance
106. # Get tasks in Sequencer for rover received as user commands
107. matches   = [d for d in self.Sequencer if (d[4] == roverid and d[5] == sender)]
108. # Get indexes in Sequencer for user commands
109. indexes   = [i for i,x in enumerate(self.Sequencer) if x in matches]
110. # Get current distances of user commands
111. distances = [d[8] for d in matches]
112. distances.append(distance)
114. # Sort sublist based on distance
115. distancesorted = sorted(distances)
117. # Get index of distance actual user command in sorted sublist
118. index = [i for i,x in enumerate(distancesorted) if (x == distance)][0]
120. # Compute index in Sequencer
121. if (indexes != []):
123. # Index in Sequencer
124. newindex = index + indexes[0]
126. # Compute priority
127. priority = self.Sequencer[indexes[0]][1] + index
129. else:
131. # Get more priotary tasks for the same rover
132. tasks = [d for d in self.Sequencer if (d[4] == roverid and d[5] == 'Central')]
134. # Check if there are more prioritary tasks for the same rover
135. if(tasks != []):
137. # Index in Sequencer (user command with most prio   )
138. newindex = indexes[-1] + 1
140. # Compute priority
141. priority = self.Sequencer[indexes[-1]][1] + 1
143. # No more prioritary tasks
144. else:
145. # Check if there are any command for rover in Sequencer
146. matches = [d for d in self.Sequencer if (d[4] == roverid)]
147. # Get indexes of the commands
148. indexes = [i for i,x in enumerate(self.Sequencer) if x in matches]
150. if (indexes != []):
151. # Index in Sequencer (command with most priority for rover)
152. newindex = indexes[0]
154. else:
156. # Index in Sequencer (add to the bottom of the Sequencer)
157. newindex = len(self.Sequencer)
159. # Compute priority
160. priority = 1
162. # Command structure
163. command = [id,priority, latitude, longitude, roverid, sender, timeout, timestamp,distance]
164. # Add  command
165. self.Sequencer.insert(newindex,command)
166. # Update priority
167. self.update\_priority(roverid,priority,delete=False)
168. # Reset usercmd
169. self.new\_usercmd = False
170. # -------------------------------------------------------------------------------------------------------------------
172. # ------------------------------------------ PRIORITY 3: SENSOR COMMAND (CMD) ---------------------------------------
174. # Check if new sensor data is received
175. if (self.new\_sensorcmd):
177. print("Rover " + str(self.sensorcmd['Sender']) + " found an interesting location. Computing priority of task and assigning it...\n")
178. # Task ID
179. self.task\_id =  self.task\_id + 1
181. # Sensor command
182. id         =  self.task\_id                                                                   # (0) Task ID
183. priority   =  3                                                                              # (0) Priority
184. latitude   =  self.sensorcmd['Latitude']                                                     # (1) Latitude
185. longitude  =  self.sensorcmd['Longitude']                                                    # (2) Longitude
186. roverid    =  self.compute\_roverid(self.sensorcmd['Status'])                                 # (3) Rover ID
187. sender     =  self.sensorcmd['Sender']                                                       # (4) Sender
188. timeout    =  None                                                                           # (5) Timeout (Compute later)
189. timestamp  =  self.sensorcmd['Timestamp']                                                    # (6) Timestamp
190. try:
191. # ID of current rover task
192. current\_id  = [d for d in self.RoversTask if (d[0] == self.sensorcmd['RoverId'])][0][1]
193. # Index of current task in Sequencer
194. index = [i for i,x in enumerate(self.Sequencer) if (x[0] == current\_id)][0]
195. # Compute distance to current task location
196. lat\_now    =  self.Sequencer[index][2]                                                     # Current task latitude
197. long\_now   =  self.Sequencer[index][3]                                                     # Current task longitude
199. except:
200. # Get last location
201. lat\_now   = [d[1] for d in self.Locations if (d[0] == roverid)][0]                          # Current latitude
202. long\_now  = [d[1] for d in self.Locations if (d[0] == roverid)][0]                          # Current longitude

205. # Compute timeout and distance
206. timeout    =  self.compute\_timeout([[lat\_now, long\_now],[latitude, longitude]])                  # (6) Timeout
207. distance   =  self.compute\_distance([[latitude, longitude],[lat\_now, long\_now]])                 # (8) Distance

210. # Get tasks and indexes in Sequencer for rover received as user commands
211. matches   = [d for d in self.Sequencer if (d[4] == roverid and d[5] == sender)]
212. # Get indexes in Sequencer for user commands
213. indexes   = [i for i,x in enumerate(self.Sequencer) if x in matches]
214. # Get distances
215. distances = [d[8] for d in matches] + distance
216. # Sort sublist based on distance
217. distancesorted = sorted(distances)
218. # Get index of distance actual user command in sorted sublist
219. index = [i for i,x in enumerate(distancesorted) if (x == distance)][0]

222. # Compute index in Sequencer
223. if (indexes != []):
224. # Index in Sequencer
225. newindex = index + indexes[0]
227. # Compute priority
228. priority = self.Sequencer[indexes[0]][1] + index
230. # No sensor commands in Sequencer for this rover
231. else:
233. matches = [d for d in self.Sequencer if (d[4] == roverid)]
234. indexes = [i for i,x in enumerate(self.Sequencer) if x in matches]
235. #Get automatic tasks for rover
236. task    = [d for d in self.Sequencer if (d[4] == roverid and d[5] == 'Auto')]
238. # Check if there are automatic commands
239. if(task != []):
240. # Get index of first automatic command for rover
241. index = [i for i,x in enumerate(self.Sequencer) if x in task][0]
243. # Index in Sequencer
244. newindex = index
246. # Compute priority
247. priority = self.Sequencer[index][1]
249. # No automatic commands yet
250. else:
251. # Check if rover has other tasks in Sequencer
252. matches = [d for d in self.Sequencer if (d[4] == roverid)]
253. indexes = [i for i,x in enumerate(self.Sequencer) if x in matches]
255. if (indexes != []):
256. # Index in Sequencer (added to the bottom of the commands for the rover)
257. newindex = indexes[-1] + 1
259. # Compute priority
260. priority = self.Sequencer[indexes[-1]][1]
262. # Rover has no tasks in Sequencer yet
263. else:
264. # Index in Sequencer
265. newindex = len(self.Sequencer)
267. # Compute priority
268. priority = 1
269. # Command structure
270. command = [id,priority, latitude, longitude, roverid, sender, timeout, timestamp,distance]
271. # Add  command
272. self.Sequencer.insert(newindex,command)
273. # Update priority
274. self.update\_priority(roverid,priority,delete=False)
275. # Reset sensor comand to false (alredy processed information)
276. self.new\_sensorcmd = False
277. # -------------------------------------------------------------------------------------------------------------------
278. # ------------------------------------------ PRIORITY 4: AUTOMATIC COMMAND ----------------------------------------
280. # Check if a new automatic command is generated (LEAST PRIORITY)
281. if (self.new\_autocmd):
283. print("Computing priority of automatics commands generated...\n")
284. # Compute priority of all automatics commands generated
285. for command in self.autocmds:
287. # Current rover location
288. index = [i for i,x in enumerate(self.Locations) if (x[0] == command[4])][0]
289. lat\_now = self.Locations[index][1]
290. lng\_now = self.Locations[index][2]
291. # Task ID
292. self.task\_id =  self.task\_id + 1
293. # New automatic command
294. id         =  self.task\_id                                                                # (0) Task ID
295. priority   =  None                                                                        # (1) Priority
296. latitude   =  command[2]                                                                  # (2) Latitude
297. longitude  =  command[3]                                                                  # (3) Longitude
298. roverid    =  command[4]                                                                  # (4) Rover ID
299. sender     =  "Auto"                                                                      # (5) Sender
300. timeout    =  self.compute\_timeout([[lat\_now, lng\_now],[latitude, longitude]])            # (6) Timeout (Compute later)
301. timestamp  =  time.time()                                                                 # (7) Timestamp
302. distance   =  0                                                                           # (8) Distance (not needed)


306. # Get tasks in Sequencer for rover
307. matches  = [d for d in self.Sequencer if (d[4] == command[4])]
309. # Compute priority
310. if (matches != []):
311. # Compute priority as the least prioritary task for rover in Sequencer
312. priority = [d[1] for d in matches][-1] + 1
314. # No tasks found
315. else:
317. # Compute priority as the most prioritary task for rover in Sequencer
318. priority = 1
320. # Command structure
321. command    =  [id, priority, latitude, longitude, roverid, sender, timeout, timestamp, distance]
323. try:
324. # Get index of last task in Sequencer for the rover
325. index = [d for d,x in enumerate(self.Sequencer) if (x[4] == command[4])][-1] + 1
327. # Insert command in Sequencer
328. self.Sequencer.insert(index,command)
330. except:
331. # Add command to the bottom of Sequencer
332. self.Sequencer.append(command)
334. # Reset autocmd
335. self.new\_autocmd = False
336. # Update rovers task
337. self.update\_rover\_task(rover=None,cmd = None)
338. # ------------------- SAVE SEQUENCER IN MYSQL DB ----------------------------
340. # Conexion to mysql database
341. cnx = pymysql.connect(host='localhost', user='root', password='centralrovers', database='CentralInfo')
342. # Cursor
343. cursor = cnx.cursor()
345. # Get Sequencer rows
346. data = [tuple(d) for d in self.Sequencer]
348. # Delete last Sequencer
349. drop\_query = "DROP TABLE IF EXISTS Sequencer;"
350. cursor.execute(drop\_query)
351. # Commit
352. cnx.commit()
353. # Create a new table
354. create\_query = "CREATE TABLE Sequencer (TaskId INT, Priority INT, Latitude FLOAT, Longitude FLOAT, RoverId VARCHAR(255), Sender VARCHAR(255), Timeout FLOAT, Timestamp FLOAT, Distance FLOAT);"
355. cursor.execute(create\_query)
356. # Insert Sequencer
357. query = "INSERT INTO Sequencer (TaskId, Priority, Latitude, Longitude, RoverId, Sender, Timeout, Timestamp, Distance) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s);"
358. cursor.executemany(query, data)
359. # Commit
360. cnx.commit()
361. # Close conection
362. cnx.close()
363. # -------------------------------- SEND SEQUENCER (MQTT) ------------------------------
364. # -------------- INITIAL MESSAGE -----------------
365. msg = {}
366. msg['Info'] = "Sending"
368. # Serialize json
369. msg = js.dumps(msg)
370. # Publish
371. self.Mqtt.publish(client,cmdtopic,msg)
373. # -------------- SEND SEQUENCER -------------------
374. # List for saving Sequencer in json
375. data = []
376. # Transform list to dictionary
377. for d in self.Sequencer:
378. seq = dict(zip(["Priority", "Latitude", "Longitude", "RoverId", "Sender", "Timeout", "Timestamp"], [d[1], d[2], d[3], d[4], d[5], d[6], d[7]]))
379. data.append(seq)
380. # Mesage structure for Sequencer
381. msg = {}
382. msg['documents'] = data
383. # Serialize json
384. json = js.dumps(msg)
386. # Publish
387. self.Mqtt.publish(client,cmdtopic,json)
388. # ----------------------- SERIAL MESSAGES (TABLE FORMAT) -----------------------------
389. print("\nSEQUENCER")
391. # Sequencer table
392. print('\n' + tabulate(self.Sequencer, headers=["ID", "Priority", "Lat", "Long", "RoverID", "Sender", "Timeout", "Timestamp", "Distance"])+'\n' )
393. # ------------------------------------------------------------------------------------

Fig. 6. Código de la función

A continuación, se ejecuta la función de auto\_plan, que genera tareas automáticas en caso de que no las haya ya. El valor de la coordenada automática se computa de manera aleatoria, manteniendo una coordenada con el mismo valor y otra oscila entre -1 y 1 respecto al valor anterior. Hay dos problemas principales a evitar en esta función:

* La latitud o longitud está fuera de las cotas
* El sitio ha sido explorado en los últimos 5 días.

En estos casos, se vuelve a generar la coordenada aleatoria.

Una vez acabado este proceso, se crea una lista con todos los valores automáticos que se han generado y se establece el new\_autocmd a true.

   # ------------------------------------------------ AUTO PLAN -----------------------------------------------------------

    def auto\_plan(self):

        """

        This method will:

        - Generate automatic plan based on coordinates

        - Check if automatic plan is inside the map

        """

        # ---------------------- UPDATES --------------------------

        self.update\_rover\_task(rover=None,cmd=None)

        # -------------- CHECK AUTO COMMANDS ------------------------

        # Get Rovers with no auto commands

        autocmds =  [d for d in self.Sequencer if (d[5] == 'Auto')]                                       # Find auto commands

        matches  =  [d[4] for d in autocmds]                                                              # Get RoverIds of the auto commands

        missing  =  [rover\_id for rover\_id in self.RoversId if rover\_id not in matches]                   # Find RoverIds with no auto commands

        # Check if there are Rovers with no auto commands

        if (len(missing)>0):

            # ---------------- GENERATE AUTO PLAN --------------------------

            print("ROVERS WITH NO AUTOMATIC PLAN CURRENTLY: " + str(missing)+'\n')

            print("... GENERATING AUTOMATIC COMMANDS...\n")

            latitudes  = []

            longitudes = []

            # Get coordinates of current tasks of Rovers with no auto commands

            for rover in missing:

                print("... CREATING AUTO PLAN FOR " + str(rover) + "...\n")

                print("...GETTING " + str(rover) + " COORDINATES...\n")

                # Check if rover has a current task assigned

                matches  = [d for d in self.RoversTask if (d[0] == rover)]

                # Check if rover has other tasks assigned

                if (len(matches)> 0 and all(x[1] is not None for x in matches)):

                    # Get current Task Id

                    current\_id =  matches[0][1]

                    print("Current task of rover " + str(rover) + ": " + str(current\_id))

                    index      =  [i for i,x in enumerate(self.RoversTask) if (x[1] == current\_id[0])][0]

                    # Coordinates

                    latitudes.append(self.RoversTask[index][2])                                   # Latitude of current task

                    longitudes.append(self.RoversTask[index][3])                                  # Longitude of current task

                else:

                    # Inform

                    print("Info: Rover " + str(rover) + " has no current task assigned\n")

                    print ("Info: Getting last location received from " + str(rover)+'\n')

                    # Get last location received from rover

                    index = [i for i,x in enumerate(self.Locations) if (x[0] == rover)][0]

                    latitudes.append(self.Locations[index][1])                                    # Latitude of current location

                    longitudes.append(self.Locations[index][2])                                   # Longitude of current location

            # Generate automatic plan based on coordinates. Close latitude and same longitude.

            auto\_latitudes  = [x + self.direction \* np.random.uniform(0, 1) for x in latitudes if x is not None]

            auto\_longitudes = longitudes

            # Check if automatic plan is inside the map

            outer\_latmax = [d for d,x in enumerate(auto\_latitudes) if  (x > self.max\_lat)]

            outer\_latmin = [d for d,x in enumerate(auto\_latitudes) if  (x < self.min\_lat)]

            outer\_lng    = [d for d,x in enumerate(auto\_longitudes) if (x > self.max\_long)]

            if (len(outer\_latmax) > 0):

                # Change direction

                self.direction = -1

                # Set latitude inside the map and compute different longitude

                auto\_longitudes [outer\_latmax] = [x + self.direction \* np.random.uniform(0, 1) for x in longitudes[outer\_latmax]]

                auto\_latitudes  [outer\_latmax] = self.max\_lat

            elif (len(outer\_latmin) > 0):

                # Change direction

                self.direction = 1

                # Set latitude inside the map and compute different longitude

                auto\_longitudes [outer\_latmin] = [x + self.direction \* np.random.uniform(0, 1) for x in longitudes[outer\_latmax]]

                auto\_latitudes  [outer\_latmax] = self.min\_lat

            if (len(outer\_lng)> 0):

                    # Set longitude inside the map

                    auto\_longitudes[outer\_lng] = self.min\_long

            # Check if automatic plan has been explored recently (last 5 days)

            for d in range(len(auto\_latitudes)):

                while ((auto\_latitudes[d], auto\_longitudes[d]) in self.Explored):

                    matches = [i for i in self.Explored if (i[2]< 5)]

                    # Generate different latitude

                    if (matches != []):

                        auto\_latitudes[d] = latitudes[d] + self.direction \* np.random.uniform(0, 1)

            # Create list of the auto commands to add to the Sequencer:

            # [Id,Priority,Latitude, Longitude, RoverId, Sender, Timeout, Timestamp, Distance]

            autocmds = []

            for i in range(len(missing)):

                autocmds.append([None,None, auto\_latitudes[i], auto\_longitudes[i], missing[i], 'Auto', None, None, None])

                # Inform

                message = "{'Latitude':"+str(auto\_latitudes[i])+", 'Longitude':"+str(auto\_longitudes[i])+"}\n"

                print ("NEW AUTOMATIC COMMAND FOR " + str(missing[i])+ ": "+ message + '\n')

            # Save new automatic commands

            self.autocmds   = autocmds

            # Activate new automatic commands

            self.new\_autocmd = True

        else:

            # No new automatic command

            self.new\_autocmd = False

            autocmds         = None

            # Inform

            print("...ALL ROVERS HAVE CURRENTLY AN AUTOMATIC COMMAND ALREDY...\n")

            print("...NO NEW AUTOMATIC COMMAND WILL BE GENERATED...\n")

Fig. 7. Código de la función

Tras este flujo lógico, se guarda el secuenciador en la base de datos, se envía por mqtt al topic II15/Commands (aunque, previamente, se envía un JSON con un campo “Info” de valor “sending” que permita a Node-RED el correcto procesado de la trama.

Con send\_cmd(), se transmite el comando por LoRa a los rovers si el estado es de emergencia o ha terminado una tarea. En caso de haber terminado una tarea, se actualiza la lista de sitios explorados con update\_explored y el secuenciador, con update\_sequencer. Si el rover no tiene tareas, se actualiza el valor de pending. Para el envío de la tarea, se invoca al actualizador de tareas update\_rover\_task, así como se actualizan las distancias[[5]](#footnote-6) y el valor de timeout.

Todas las funciones anteriores se van invocando secuencialmente en el bucle principal, tal y como se muestra en la imagen inicial.

Para concluir, cabe recalcar que es necesario crear dos threads, de central y serial, para evitar que se bloquee el script al ejecutar el bucle de lectura serial y el bucle de la lógica de la central.

### Render 3D

Este código utiliza Flask y Dash para crear una aplicación web interactiva que muestra un render de un planeta y tres opciones de visualización.

1. Sequencer: con las tareas actuales del secuenciador. Posee un tooltip con el robot asignado a esa ubicación
2. Current tasks: muestra las tareas actuales, con el tooltip correspondiente al rover
3. Explored map: genera los puntos de las tareas ya realizadas, así como el tooltip correspondiente.

En esencia, hay una sección de configuración inicial, donde se manejan la obtención y procesamiento de datos, mientras que el layout y los callbacks se encargan de la interfaz de usuario y la actualización dinámica del render. Además, se establece un túnel de acceso remoto para permitir la visualización desde ubicaciones externas.

#### Setup

Convierte los datos recibidos de latitud y longitud a coordenadas con getxy(), y adapta dichos valores a puntos de la superficie del planeta con la función closest().

A continuación, importa una plantilla de datos, realiza una solicitud POST y obtiene el JSON resultante.

Finalmente, da formato a los datos serializando los paralelos, meridianos y superficie, definiendo los colores y generando las trazas de datos, para agrupar todo en el objeto “data”.

#### Layout

Se genera el layout del render, y se convierte en una figura. Con la librería Flask de Python, se ejecuta en el servidor, se introducen los iconos y botones correspondientes a las posibilidades de visualización y, con toda esa trama, se crea el Dash.

Gracias a ngrok, establece el túnel de acceso remoto para poder invocarlo a través de un iframe de Node-RED, y, sobre esa página, se construye el layout de la app.

Callback

Callback es una función que tiene como output el render y como input los botones, así como un intervalo de tiempo, de manera que, si no se actualiza a través de ningún botón, lo hará cada 8 segundos.

Seguidamente, toma de la base de datos la información del secuenciador y calcula la traza y los puntos para cada una de las posibilidades de visualización.

Para concluir, crea el dibujo como plot y ejecuta el servidor.

### MQTT

La comunicación con MQTT está hecha para recibir mensajes de los topics correspondientes y guardarlos en la colección de Central. Este código por separado ha sido necesario porque se sobrecarga el script y se desconecta el bróker si las lecturas del mqtt se hacen en el Python de Central.

## Chatbot de Telegram

El Chatbot tiene tres funciones principales:

1. Consultar las coordenadas actuales de los rovers.
2. Enviar una consigna para que uno de los rovers se dirija a un punto determinado.
3. Descargar como .csv los datos de los sensores para analizarlos en el laboratorio.

Para desarrollarlo, se ha recurrido a Node-RED, implementando el flujo que se muestra a continuación: Diagrama

Descripción generada automáticamente

Fig. 8 Flujo

### Main

La función Main es responsable de que, si llega algún comando o coordenadas a través del chat, se procesen adecuadamente y, de lo contrario se envíe un mensaje indicando el correcto uso del bot.

Texto

Descripción generada automáticamente

Fig.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Fig. 9. Flujo

Algunos detalles relevantes de esta función se retomarán más adelante.

### Start

En el comando /start, se recoge la información básica del bot. El envío de este comando es en el formato habitual de Telegram.

Texto

Descripción generada automáticamente

Fig. 10. Mensaje de Telegram

### Coordinates

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente/coordinates permite extraer las coordenadas de cualquiera de los dos rovers, de entre los cuales se puede escoger como opción:

Fig. 11. Mensaje de Telegram

Al pinchar en alguna de las dos, automáticamente se publican las últimas coordenadas registradas.

Para implementar esta posibilidad de pinchar sobre el objeto, se ha utilizado un objeto JSON que devuelve un número en función del botón pulsado.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Fig. 12. Flujo

A su vez, dicho número se recibe en la callback\_query, donde se procesa la lógica asociada al mismo.

Texto, Carta

Descripción generada automáticamente

Fig. 13. Flujo

Esta información es la que se introduce en la consulta de MongoDB, y posteriormente se envía al chatbot.

### Set Waypoint

Del mismo modo, el comando /set\_waypoint también permite escoger el robot al cual se va a enviar a unas nuevas coordenadas, cuyo formato de lectura corresponde a [x,y]. Una vez recibida dicha coordenada y enviada al secuenciador de tareas, se confirma la transmisión correcta de la información.

Texto

Descripción generada automáticamente

Fig. 14. Mensaje de Telegram

En este caso, los valores recibidos a través del callback\_query son “0” y “1”, y el mensaje transmitido indica el formato adecuado de coordenadas.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Fig. 15. Flujo

Después, se lee lo enviado por el chat gracias a Receiver, que puede detectar la llegada de una coordenada. En caso de hacerlo, genera una variable nueva y envía las coordenadas para su conversión a JSON.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamenteFinalmente, se serializa y manda a través del chatbot.

Fig. 16. Flujo

### Download

Adicionalmente, /download invoca a la base de datos de MongoDB y exporta su colección en .csv, escribe un archivo y lo envía a través de Telegram. Para enviar archivos por este medio, basta con indicarlo en el tipo del mensaje.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Fig. 17. Flujo

### Emergency

Finalmente, en caso de emergencia, se envía un mensaje indicando la causa de esta, y advirtiendo al usuario del posible mal funcionamiento del servicio.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Fig. 18. Mensaje de Telegram

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Fig. 19. Flujo

## Node-RED

El Dashboard de Node-RED permite visualizar gran parte de los datos, realizar distintas consultas y comprobaciones en tiempo real. Sus distintas pestañas recogen las funciones principales, que corresponden también a las distinciones dentro del flujo de programación.

### Map

Visualización 2D de las ubicaciones exploradas por los rover, así como las medidas obtenidas en ellas, y render 3D de las coordenadas.

Pantalla de video juego

Descripción generada automáticamente con confianza media

Una caricatura de una persona

Descripción generada automáticamente con confianza bajaFig. 21. Dashboard, Mapas

Fig. 20. Dashboard, detalle de Mapa

Para identificar los rovers, se muestran balizas con iconos. Así, el rover sensor corresponde a la baliza roja con el icono de la galga, mientras que el rover actuador es la baliza azul con la probeta. Cada uno de ellos cuenta también con un tooltip donde se muestran los valores obtenidos por los sensores o la operación efectuada, respectivamente.

Diagrama

Descripción generada automáticamente

Fig.

Para calcular las coordenadas del mapa, se utiliza el topic /Rovers en el caso del actuador, y /Sensors en el caso del sensor. Ambos se adaptan y guardan en sus respectivas colecciones de mongodb, para después consultarse y adaptar a los campos del mapa.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Fig. 22. Flujo

Dado que /Rovers publica constantemente el estado de ambos rovers, es necesario acotar el filtro al actuador, en caso de que esté disponible, pues esto supone que ya habrá tomado una medida.

Para permitir una correcta visualización del mapa, es necesario que el mensaje se muestre con los campos a continuación:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente con confianza media

Fig. 23. Flujo

Hay un timestamp que se ejecuta al inicio para definir una coordenada fija, la de la central, que se muestra en negro y con un icono de una casa.

### Commands

Pantalla de computadora

Descripción generada automáticamenteMuestra la tabla de comandos. Se puede filtrar por fecha, rover y prioridad.

Fig. 24. Dashboard. Comandos

Escala de tiempo

Descripción generada automáticamente

Fig.

Los datos de esta tabla tienen la peculiaridad de insertarse todos juntos, y eliminarse cada vez que entran datos nuevos. Por ello, la forma de actualizarse es diferente a la habitual.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Fig. 25. Flujo

Previo a la actualización de la tabla, llega desde la central un mensaje de “Sending” en el campo “Info”, que elimina la colección. De lo contrario, se inserta todo el contenido en la colección.

Para mostrar u ocultar el grupo de filtros, se necesita implementar un “ui control”, de manera que, si el deslizador de “Apply Filters” está activado, se muestren, mientras que, si no lo está, se desactiven. El código asociado a esta función se muestra a continuación:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Fig. 26. Flujo

Para poder aplicar todos los filtros simultáneamente, es necesario declararlos como “Flow” y aplicar la siguiente función al query:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Fig. 27. Flujo

Así, se agregarán todos los campos que sean necesarios en el mismo “msg.payload”. Existe un detalle a tener en cuenta, y es el timestamp cíclico que actualiza el query, para evitar que cada vez que llegue un dato se borren los filtros, o que solo se apliquen en cada entrada de valores.

### Captura de pantalla con la imagen de una pantalla Descripción generada automáticamente con confianza mediaSensors

Fig. 28. Dashboard, sensores

La pestaña “Sensors” muestra el histórico, valor actual, máximo, promedio y mínimo de cada uno de los sensores del rover sensor. Para desarrollarlo, basta con tomar los datos del topic /Sensors y deserializarlos, aplicando las funciones pertinentes en cada caso.

Diagrama

Descripción generada automáticamente

Fig. 29. Flujo

### Emergency

Captura de pantalla de computadora

Descripción generada automáticamentePanel de emergencias, filtrable por fecha, rover y tipo de emergencia. Sigue un formato similar a “Commands” en cuanto a los filtros respecta.

Fig. 30. Dashboard, emergencias

Captura de pantalla con la imagen de una pantalla

Descripción generada automáticamente Existe una notificación push que avisa en caso de la llegada de una nueva emergencia, permitiendo dirigirse, o no, a la pestaña de emergencias, para así actualizar la búsqueda.

Fig. 31. Dashboard. Notificación de emergencia

Implementar dicha notificación requiere no solo del uso del nodo de mensajes, sino del ui control de nuevo, para cambiar la ventana en caso de mandar “View emergencies”.

Escala de tiempo

Descripción generada automáticamente

Fig. 32. Flujo

Para cambiar la ventana, el código necesario viene recogido aquí:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Fig. 33. Flujo

# Conclusiones

Hemos comprobado que, efectivamente, podemos definir una red de MiniRovers con la topología implementada en nuestro diseño. Hemos comprobado la viabilidad de la ESP32 como sistema de Edge-computing, llevándola casi al límite de su memoria y capacidad de procesamiento, y hemos implementado un lenguaje de alto nivel alternativo, Python, que permite simultanear el control de bases de datos con la comunicación.

No obstante, existen aspectos a mejorar del proyecto: si se hubiese dispuesto de una mayor inversión, se podría comprar hardware específico para aplicaciones espaciales, que permitiera comprobar el rendimiento de estos protocolos, o se podría haber hecho un desplazamiento real de los rovers en vez de simulado en el hardware, e incluso se podría haber comprobado la viabilidad de integrar el protocolo CCSDS en la topología. También habría sido conveniente sustituir la computadora actual por una Raspberry Pi 4B, pero los problemas de hardware asociados a la misma no hicieron posible un correcto desarrollo del proyecto es esa computadora. Adicionalmente, no se disponía de un hosting propio donde poder implementar APPs adicionales.

En síntesis, ha sido un proyecto enriquecedor que seguiremos nutriendo con el conocimiento adquirido en el proceso, y con el apoyo de aquellos que se quieran involucrar en esta trepidante tarea de la exploración espacial en miniatura.

Para más información, y un acceso al código completo, se puede visitar el repositorio de GitHub. <https://github.com/albacorreal/infind.git>

1. El “Status” de Rovers indica a la Central que hay un caso de emergencia, pero no se indica cuál, porque es indiferente para la navegación. Sí se indica en Emergency, que se envía a Node-RED, para notificar a los científicos [↑](#footnote-ref-2)
2. En el caso específico de commands, RoverId corresponde al rover al que se quiere mandar el comando, mientras que Sender es quien lo genera. [↑](#footnote-ref-3)
3. Nótese que ninguna de la sensórica usada es apta para el uso espacial. No obstante, la obtención de módulos funcionales en ambientes extraterrestres es costosa y requeriría de un testeo, procesado e inversión que excede el ámbito de la asignatura. [↑](#footnote-ref-4)
4. Los umbrales definidos para activar el desencadenante de “ubicación de interés” no tienen validez científica, sino meramente ilustrativa. [↑](#footnote-ref-5)
5. La distancia se ha calculado con la fórmula de Haversine, dadas la latitud y longitud del globo planetario [↑](#footnote-ref-6)