



Team Project

---

# IOT MINIROVERS

---

INFINID

# MiniRovers

## SUMMARY

**Abstract** - The aim of this report is to show the implementation of a Proof of Concept regarding the future of Space Exploration: modular rovers interconnected with different protocols. The scope of this PoC encloses the control, sequencing, intercommunication, data processing and UI of this system, showcasing that a reduction of payloads and an increase in autonomy is possible applying basic IoT concepts [1].

**Keywords** - IoT, NodeRed, MiniRovers, Modularity, Python

*Correal. O. Alba, Castro L. Laura, Mengual P. Víctor G., Ávila O. Jorge*

## 01

### The necessity of MiniRovers

## 02

### Implementation of the Network

## 03

### Results and further tasks

## The necessity of MiniRovers.

Space Exploration has always been one of the most challenging aspirations of humankind. Given the complexity of these robotic mechanisms, the usage of smaller robots with specific tasks interconnected with one another is becoming increasingly popular [2].

However, a correct definition of this robotic arrays is a very complex task, and several questions arise: What should be the hierarchy of the MiniRovers? How can we define which data shall be sent? How will scientists interact with these networks?

As a Proof of Concept, we will provide a possible definition of this array, delving into some of the key features that shall be implemented.

## Implementation of the Network.

Our PoC consists of two MiniRovers: a sensor, that retrieves simple data from the explored body, and an actuator, that takes ground samples. Both of them are controlled by a central computer, that sends them to specific coordinates for performing their tasks. Similarly, they send some of their data to close nanosatellites in the area, having a cloud database that can be accessed from Earth thanks to a chatbot and a website. Given the scope of the project, no CCSDS network has been defined to communicate both WiFi subsystems[3].

A detailed explanation of the control system, its code and interfaces is provided here.

## Results and further tasks.

We provided a successful definition of the array, pointing out how to solve some of the most common difficulties of Space Exploration. Nonetheless, hardware and communication protocols shall be adapted to those used in common space applications.

- [1] J. Kua, S. W. Loke, C. Arora, N. Fernando, and C. Ranaweera, "Internet of things in space: A review of opportunities and challenges from satellite-aided computing to digitally-enhanced space living," *Sensors*, vol. 21, no. 23, Dec. 2021, doi: 10.3390/S21238117.
- [2] R. J. Kerczewski, K. B. Bhasin, T. P. Fabian, J. H. Griner, B. A. Kachmar, and A. M. Richard, "In-Space Internet-Based Communications for Space Science Platforms Using Commercial Satellite Networks," 1999. [Online]. Available: <http://www.sti.nasa.gov>
- [3] CCSDS, "Report Concerning Space Data System Standards WIRELESS NETWORK COMMUNICATIONS OVERVIEW FOR SPACE MISSION OPERATIONS INFORMATIONAL REPORT CCSDS HISTORICAL DOCUMENT CCSDS HISTORICAL DOCUMENT," 2015.

# Contenido

---

Índice de ilustraciones .....	3
Índice de tablas .....	3
Introducción.....	4
Necesidad de los MiniRovers.....	4
KVs .....	4
Desarrollo del Proyecto.....	5
Escenario.....	5
Funcionalidades .....	6
Comunicación y tramas de datos.....	7
MiniRovers.....	9
Hardware .....	9
Comunicaciones .....	10
Máquina de estados .....	12
Tramas y envío de datos .....	13
Central .....	14
Render 3D.....	30
MQTT.....	31
Chatbot de Telegram.....	31
Main .....	32
Start.....	32
Coordinates.....	33
Set Waypoint.....	35
Download.....	36
Emergency .....	36
Node-RED .....	37
Map .....	37
Commands.....	39
Sensors.....	42
Emergency .....	43
Conclusiones.....	45

# Índice de ilustraciones

---

Fig. 1. Diagrama del Proyecto .....	5
Fig. 2. Código de comunicaciones.....	11
Fig. 3. Código de Comunicaciones.....	12
Fig. 4. Máquina de estados.....	12
Fig. 5. Diagrama de flujo.....	15
Fig. 6. Código de la función .....	26
Fig. 7. Código de la función .....	30
Fig. 8 Flujo.....	31
Fig. . Flujo.....	32
Fig. . Mensaje de Telegram .....	33
Fig. . Mensaje de Telegram .....	33
Fig. . Flujo.....	34
Fig. . Flujo.....	34
Fig. . Mensaje de Telegram .....	35
Fig. . Flujo.....	35
Fig. . Flujo.....	36
Fig. . Flujo.....	36
Fig. . Mensaje de Telegram .....	36
Fig. . Flujo.....	37
Fig. . Dashboard, Mapas.....	38
Fig. . Dashboard, detalle de Mapa.....	38
Fig. . Flujo.....	39
Fig. . Flujo.....	39
Fig. . Dashboard. Comandos .....	39
Fig. . Flujo.....	40
Fig. . Flujo.....	41
Fig. . Flujo.....	41
Fig. . Dashboard, sensores .....	42
Fig. . Flujo.....	42
Fig. . Dashboard, emergencias.....	43
Fig. . Dashboard. Notificación de emergencia.....	43
Fig. . Flujo.....	44
Fig. . Flujo.....	44

# Índice de tablas

---

Tabla 1. Asociación de las funciones implementadas.....	7
Tabla 2. Tramas de comunicación.....	8
Tabla 3. Constructor.....	14

# 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 aparente 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).

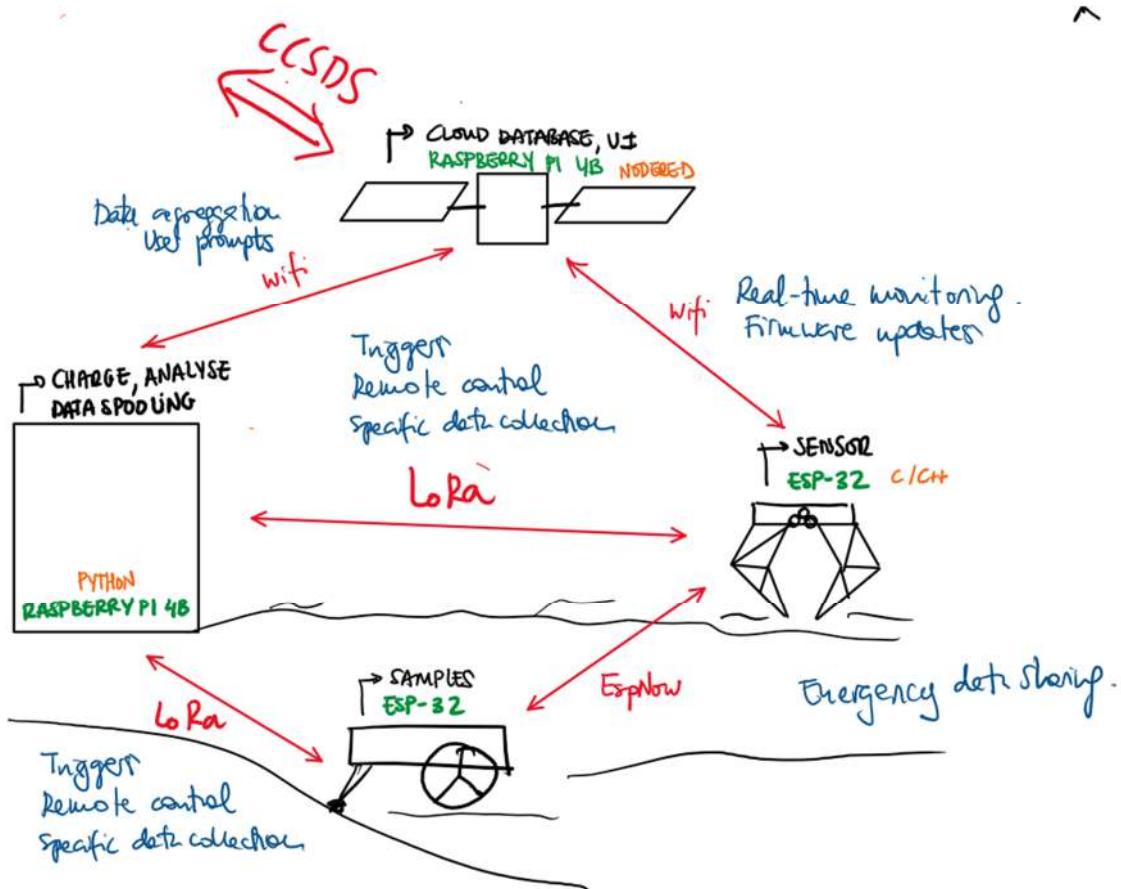


Fig. 1. Diagrama del Proyecto.

## 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 exemplificarlo, 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 → Receiver	Function	Showcase
Rover ↔ Rover	Emergency Data	Earthquake (accelerometer)
Rover → Central	Triggers	<ul style="list-style-type: none"><li>- Low Battery</li><li>- Sample Overflow</li><li>- Interesting waypoint</li></ul>
	Remote Control	<ul style="list-style-type: none"><li>- Coordinates</li><li>- Status</li></ul>
	Specific Data	<ul style="list-style-type: none"><li>- Latitude</li><li>- Longitude</li></ul> <p>(Central creates a 3D render)</p>
Central → Rover	Remote Control	<ul style="list-style-type: none"><li>- Desired Coordinates</li><li>- Task information</li></ul>
Rover → Satellite	Real-time monitoring	<ul style="list-style-type: none"><li>- Sensor Data</li><li>- Rover status</li></ul>
Satellite → Rover	Firmware updates	<ul style="list-style-type: none"><li>- FOTA (not implemented)</li></ul>
Central → Satellite	Data aggregation	<ul style="list-style-type: none"><li>- Rover Tasks data</li><li>- 3D render output</li></ul>
Satellite → Central	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 <sup>1</sup>		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
	Triggers: waypoint interesting	Central	"Altitude": float "Temperature": float "Humidity": float "Pressure": float "Radiation": float
	Real-time sensor data monitoring:	Node-RED	"Status": "Interesting location. Humidity Pressure Radiation"
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 <sup>2</sup>		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

<sup>1</sup> 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

<sup>2</sup> En el caso específico de commands, RoverId corresponde al rover al que se quiere mandar el comando, mientras que Sender es quien lo genera.

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<sup>3</sup> para medir distintos valores ambientales que permitan definir umbrales de interés<sup>4</sup>. 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 piezoresistivo 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/s<sup>2</sup>.
- 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 exemplificar 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.

---

<sup>3</sup> 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.

<sup>4</sup> Los umbrales definidos para activar el desencadenante de “ubicación de interés” no tienen validez científica, sino meramente ilustrativa.

- 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 commute 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())
  - o Mensaje no vacío
  - o 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

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

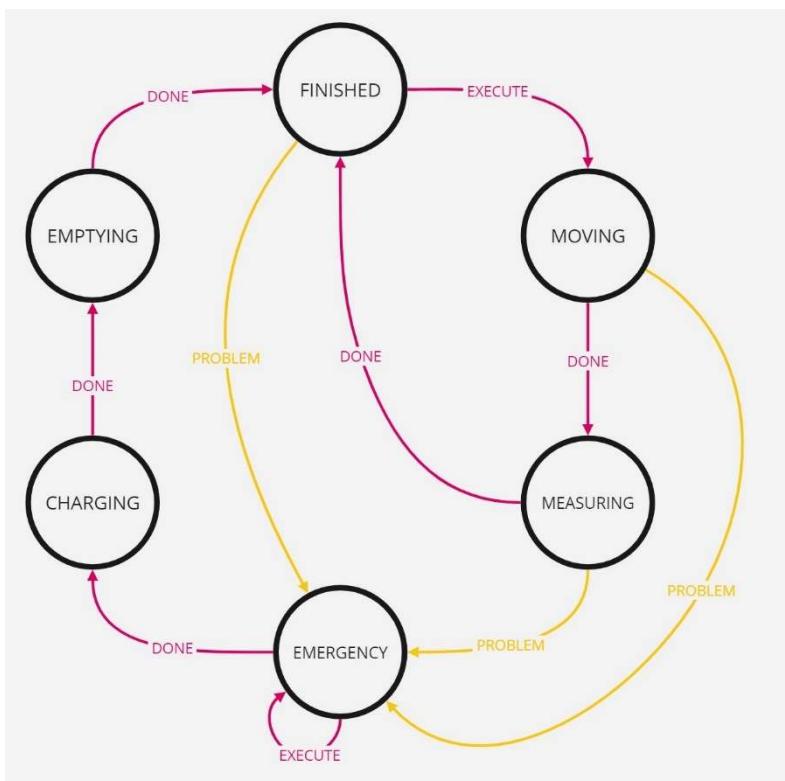


Fig. 4. Máquina de estados

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.

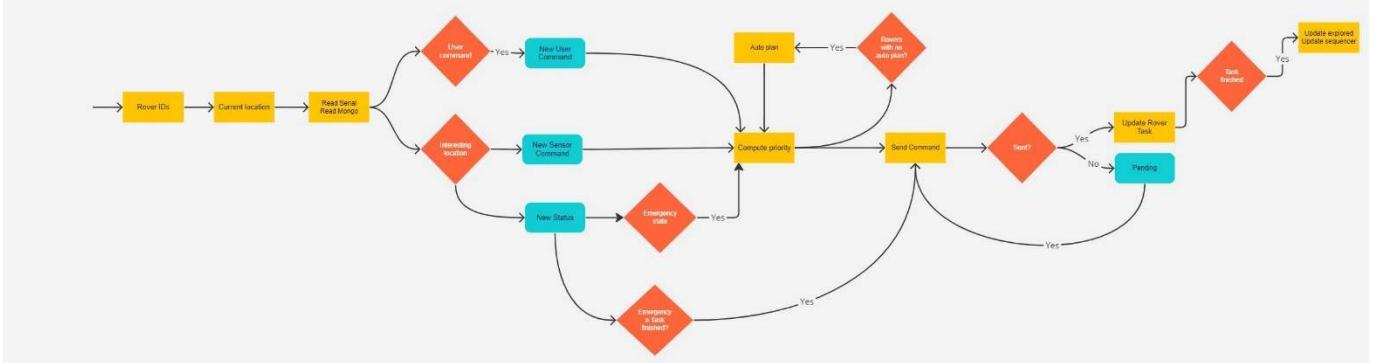


Fig. 5. Diagrama de flujo.

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)
  - 4) Nuevo comando del sensor: si el flag de “new\_sensormd” 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.

```

5)  # -----
6)  def compute_priority]:
7)
8)      """
9)          This method computes tasks priority of all Rovers
10)         PRIORITY ORDER: EMERGENCY,USER COMMAND, AUTOMATIC COMMAND
11)
12)        """
13)        # ===== ROVERS PRIORITY COMPUTATION =====
14)
15)        # -----
16)        # ID | Priority | Latitude | Length | Rover ID | timeout | Sender | Timestamp | Distance
17)        #
18)        # 5 | 1 | 25.2 | 0.2 | Sensor | 125 sec | User | 166598 | 34
19)        #
20)        # 2 | 2 | 25.2 | 0.2 | Sensor | 125 sec | User | 177509 | 67
21)        #
22)        # 0 | 3 | 25.2 | 0.2 | Sensor | 125 sec | Auto | 189067 | 23
23)        #
24)        # 3 | 1 | 25.2 | 0.2 | Actuator | 125 sec | User | 166598 | 123
25)        #
26)        # 6 | 2 | 25.2 | 0.2 | Actuator | 125 sec | User | 177509 | 234
27)        #
28)        # 1 | 3 | 25.2 | 0.2 | Actuator | 125 sec | Auto | 189067 | 56
29)        #
30)
31)
32)
33)        # ----- PRIORITY 1: RETURN HOME -----
34)
35)        # Check if new critical status is received
36)        if (self.new_status and self.status['Status']=='Emergency state'):
37)
38)            # Inform
39)            print("Info: Rover " + str(self.status['Sender']) + " is in emergency. Computing task as most prioritary...\n")
40)
41)            # Current rover location
42)            index = [i for i,x in enumerate(self.Locations) if (x[0] == self.status['Sender'])]
43)
44)            lat_now = self.Locations[index[0]][1]
45)            lng_now = self.Locations[index[0]][2]
46)
47)            # Task ID
48)            self.task_id = self.task_id + 1
49)

```

```

50)         # Return home command
51)         id      = self.task_id                                     # (0) Task ID
52)         priority = 1                                         # (1) Most priority
53)         latitude = self.home_lat                                # (2) Latitude (Home)
54)         longitude = self.home_long                             # (3) Longitude (Home)
55)         roverid  = self.status['Sender']                      # (4) Rover ID
56)         sender   = 'Central'                                    # (5) Sender
57)         timeout   = self.compute_timeout([[lat_now, lng_now],[latitude, longitude]]) # (6) Timeout
58)         timestamp = self.status['Timestamp']                  # (7) Timestamp
59)         distance  = 0                                         # (8) Distance (not needed)
60)
61)         # Command structure
62)         command = [id, priority, latitude, longitude, roverid, sender, timeout, timestamp, distance]
63)
64)         print("Info: Adding return home command to Sequencer: " + str(command)+"\n")
65)
66)     try:
67)         # Find actual most prioritary command of rover in Sequencer
68)         index = [i for i,x in enumerate(self.Sequencer) if (x[4] == roverid)][0]
69)         self.Sequencer.insert(index, command)
70)
71)     except:
72)
73)         # Add at the end of Sequencer
74)         self.Sequencer.append(command)
75)
76)         # Update priority
77)         self.update_priority(roverid, priority,delete=False)
78)
79)         # Inform
80)         print("Info: Return home command added to Sequencer\n")
81)
82)     # -----
83)
84)
85)
86)     # ----- PRIORITY 2: USER COMMAND -----
87)
88)     #Check if a new user command is received
89)     if (self.new_usercmd):
90)
91)         # Inform
92)         print("Rover " + str(self.usercmd['RoverId']) + " received a user command. Computing priority of task...\\n")
93)
94)         # Task ID

```

```

95)         self.task_id = self.task_id + 1
96)
97)         # User command
98)         id          = self.task_id                                # (0) Task ID
99)         priority    = None                                     # (1) Priority
100)        latitude     = self.usercmd['Latitude']                # (2) Latitude
101)        longitude    = self.usercmd['Longitude']               # (3) Longitude
102)        roverid     = self.usercmd['RoverId']                 # (4) Rover ID
103)        sender      = self.usercmd['Sender']                  # (5) Sender
104)        timeout      = None                                     # (6) Timeout (Computed
105)        later)
106)        timestamp   = self.usercmd['Timestamp']              # (7) Timestamp
107)        distance    = 0                                       # (8) Distance (Computed
108)        later)
109)        try:
110)
111)            # ID of current rover task
112)            current_id = [d for d in self.RoversTask if (d[0] == self.usercmd['RoverId'])][0][1]
113)
114)            # Index of current task in Sequencer
115)            index = [i for i,x in enumerate(self.Sequencer) if (x[0] == current_id)][0]
116)
117)            # Compute distance to current task location
118)            lat_now    = self.Sequencer[index][2]                  # Current task latitude
119)            long_now   = self.Sequencer[index][3]                 # Current task longitude
120)
121)        except:
122)
123)            # Get last location
124)            lat_now   = [d[1] for d in self.Locations if (d[0] == self.usercmd['RoverId'])][0]      # Current latitude
125)            long_now  = [d[1] for d in self.Locations if (d[0] == self.usercmd['RoverId'])][0]      # Current longitude
126)
127)            # Compute timeout and distance
128)            timeout   = self.compute_timeout([[lat_now, long_now],[latitude, longitude]])           # (6) Timeout
129)            distance  = self.compute_distance([[latitude, longitude],[lat_now, long_now]])          # (8) Distance
130)
131)            # Get tasks in Sequencer for rover received as user commands
132)            matches   = [d for d in self.Sequencer if (d[4] == roverid and d[5] == sender)]
133)
134)            # Get indexes in Sequencer for user commands
135)            indexes   = [i for i,x in enumerate(self.Sequencer) if x in matches]
136)
137)            # Get current distances of user commands

```

```

138)         distances = [d[8] for d in matches]
139)         distances.append(distance)
140)
141)         # Sort sublist based on distance
142)         distancesorted = sorted(distances)
143)
144)         # Get index of distance actual user command in sorted sublist
145)         index = [i for i,x in enumerate(distancesorted) if (x == distance)][0]
146)
147)         # Compute index in Sequencer
148)         if (indexes != []):
149)
150)             # Index in Sequencer
151)             newindex = index + indexes[0]
152)
153)             # Compute priority
154)             priority = self.Sequencer[indexes[0]][1] + index
155)
156)         else:
157)
158)             # Get more prioritary tasks for the same rover
159)             tasks = [d for d in self.Sequencer if (d[4] == roverid and d[5] == 'Central')]
160)
161)             # Check if there are more prioritary tasks for the same rover
162)             if(tasks != []):
163)
164)                 # Index in Sequencer (user command with most prio    )
165)                 newindex = indexes[-1] + 1
166)
167)                 # Compute priority
168)                 priority = self.Sequencer[indexes[-1]][1] + 1
169)
170)             # No more prioritary tasks
171)         else:
172)             # Check if there are any command for rover in Sequencer
173)             matches = [d for d in self.Sequencer if (d[4] == roverid)]
174)
175)             # Get indexes of the commands
176)             indexes = [i for i,x in enumerate(self.Sequencer) if x in matches]
177)
178)
179)             if (indexes != []):
180)                 # Index in Sequencer (command with most priority for rover)
181)                 newindex = indexes[0]
182)

```

```

183)         else:
184)
185)             # Index in Sequencer (add to the bottom of the Sequencer)
186)             newindex = len(self.Sequencer)
187)
188)             # Compute priority
189)             priority = 1
190)
191)
192)             # Command structure
193)             command = [id,priority, latitude, longitude, roverid, sender, timeout, timestamp,distance]
194)
195)             # Add command
196)             self.Sequencer.insert(newindex,command)
197)
198)             # Update priority
199)             self.update_priority(roverid,priority,delete=False)
200)
201)             # Reset usercmd
202)             self.new_usercmd = False
203)
204)             # -----
205)
206)
207)             # ----- PRIORITY 3: SENSOR COMMAND (CMD) -----
208)
209)             # Check if new sensor data is received
210)             if (self.new_sensorcmd):
211)
212)                 print("Rover " + str(self.sensorcmd['Sender']) + " found an interesting location. Computing priority of task and
213) assigning it...\n")
214)
215)                 # Task ID
216)
217)                 # Sensor command
218)                 id      = self.task_id                               # (0) Task ID
219)                 priority = 3                                # (0) Priority
220)                 latitude = self.sensorcmd['Latitude']          # (1) Latitude
221)                 longitude = self.sensorcmd['Longitude']        # (2) Longitude
222)                 roverid = self.compute_roverid(self.sensorcmd['Status']) # (3) Rover ID
223)                 sender   = self.sensorcmd['Sender']            # (4) Sender
224)                 timeout  = None                            # (5) Timeout (Compute
225)                             later)                           # (6) Timestamp

```

```

226)
227)     try:
228)
229)         # ID of current rover task
230)         current_id = [d for d in self.RoversTask if (d[0] == self.sensorcmd['RoverId'])][0][1]
231)
232)         # Index of current task in Sequencer
233)         index = [i for i,x in enumerate(self.Sequencer) if (x[0] == current_id)][0]
234)
235)         # Compute distance to current task location
236)         lat_now    = self.Sequencer[index][2]                                # Current task latitude
237)         long_now   = self.Sequencer[index][3]                               # Current task
238)         longitude
239)
240)     except:
241)
242)         # Get last location
243)         lat_now   = [d[1] for d in self.Locations if (d[0] == roverid)] [0]      # Current latitude
244)         long_now  = [d[1] for d in self.Locations if (d[0] == roverid)] [0]      # Current longitude
245)
246)         # Compute timeout and distance
247)         timeout   = self.compute_timeout([[lat_now, long_now],[latitude, longitude]]) # (6) Timeout
248)         distance  = self.compute_distance([[latitude, longitude],[lat_now, long_now]]) # (8) Distance
249)
250)
251)         # Get tasks and indexes in Sequencer for rover received as user commands
252)         matches   = [d for d in self.Sequencer if (d[4] == roverid and d[5] == sender)]
253)
254)         # Get indexes in Sequencer for user commands
255)         indexes   = [i for i,x in enumerate(self.Sequencer) if x in matches]
256)
257)         # Get distances
258)         distances = [d[8] for d in matches] + distance
259)
260)         # Sort sublist based on distance
261)         distancesorted = sorted(distances)
262)
263)         # Get index of distance actual user command in sorted sublist
264)         index = [i for i,x in enumerate(distancesorted) if (x == distance)][0]
265)
266)
267)         # Compute index in Sequencer
268)         if (indexes != []):
269)

```

```

270)         # Index in Sequencer
271)         newindex = index + indexes[0]
272)
273)         # Compute priority
274)         priority = self.Sequencer[indexes[0]][1] + index
275)
276)         # No sensor commands in Sequencer for this rover
277)     else:
278)
279)         matches = [d for d in self.Sequencer if (d[4] == roverid)]
280)         indexes = [i for i,x in enumerate(self.Sequencer) if x in matches]
281)         #Get automatic tasks for rover
282)         task    = [d for d in self.Sequencer if (d[4] == roverid and d[5] == 'Auto')]
283)
284)         # Check if there are automatic commands
285)         if(task != []):
286)
287)             # Get index of first automatic command for rover
288)             index = [i for i,x in enumerate(self.Sequencer) if x in task][0]
289)
290)             # Index in Sequencer
291)             newindex = index
292)
293)             # Compute priority
294)             priority = self.Sequencer[index][1]
295)
296)             # No automatic commands yet
297)         else:
298)             # Check if rover has other tasks in Sequencer
299)             matches = [d for d in self.Sequencer if (d[4] == roverid)]
300)             indexes = [i for i,x in enumerate(self.Sequencer) if x in matches]
301)
302)             if (indexes != []):
303)                 # Index in Sequencer (added to the bottom of the commands for the rover)
304)                 newindex = indexes[-1] + 1
305)
306)                 # Compute priority
307)                 priority = self.Sequencer[indexes[-1]][1]
308)
309)             # Rover has no tasks in Sequencer yet
310)         else:
311)
312)             # Index in Sequencer
313)             newindex = len(self.Sequencer)
314)

```

```

315)         # Compute priority
316)         priority = 1
317)
318)         # Command structure
319)         command = [id,priority, latitude, longitude, roverid, sender, timeout, timestamp,distance]
320)
321)         # Add command
322)         self.Sequencer.insert(newindex,command)
323)
324)         # Update priority
325)         self.update_priority(roverid,priority,delete=False)
326)
327)         # Reset sensor comand to false (already processed information)
328)         self.new_sensorcmd = False
329)
330)         # -----
331)
332)         # ----- PRIORITY 4: AUTOMATIC COMMAND -----
333)
334)         # Check if a new automatic command is generated (LEAST PRIORITY)
335)         if (self.new_autocmd):
336)
337)             print("Computing priority of automatics commands generated...\n")
338)
339)             # Compute priority of all automatics commands generated
340)             for command in self.autocmds:
341)
342)
343)                 # Current rover location
344)                 index = [i for i,x in enumerate(self.Locations) if (x[0] == command[4])][0]
345)                 lat_now = self.Locations[index][1]
346)                 lng_now = self.Locations[index][2]
347)
348)                 # Task ID
349)                 self.task_id = self.task_id + 1
350)
351)                 # New automatic command
352)                 id          = self.task_id                                # (0) Task ID
353)                 priority    = None                                     # (1) Priority
354)                 latitude   = command[2]                                 # (2) Latitude
355)                 longitude  = command[3]                                 # (3) Longitude
356)                 roverid    = command[4]                                # (4) Rover
357)                 sender     = "Auto"                                    # (5) Sender

```

```

358)         timeout    = self.compute_timeout([[lat_now, lng_now],[latitude, longitude]])           # (6) Timeout (Compute
359)         timestamp = time.time()                                         # (7) Timestamp
360)         distance   = 0                                              # (8) Distance (not
361)
362)
363)
364)         # Get tasks in Sequencer for rover
365)         matches  = [d for d in self.Sequencer if (d[4] == command[4])]
366)
367)         # Compute priority
368)         if (matches != []):
369)
370)             # Compute priority as the least prioritary task for rover in Sequencer
371)             priority = [d[1] for d in matches][-1] + 1
372)
373)         # No tasks found
374)     else:
375)
376)         # Compute priority as the most prioritary task for rover in Sequencer
377)         priority = 1
378)
379)         # Command structure
380)         command   = [id, priority, latitude, longitude, roverid, sender, timeout, timestamp, distance]
381)
382)     try:
383)         # Get index of last task in Sequencer for the rover
384)         index = [d for d,x in enumerate(self.Sequencer) if (x[4] == command[4])][-1] + 1
385)
386)         # Insert command in Sequencer
387)         self.Sequencer.insert(index,command)
388)
389)     except:
390)         # Add command to the bottom of Sequencer
391)         self.Sequencer.append(command)
392)
393)         # Reset autocmd
394)         self.new_autocmd = False
395)
396)         # Update rovers task
397)         self.update_rover_task(rover=None,cmd = None)
398)
399)         # ----- SAVE SEQUENCER IN MYSQL DB -----
400)

```

```

401)      # Conexion to mysql database
402)      cnx = pymysql.connect(host='localhost', user='root', password='centralrovers', database='CentralInfo')
403)
404)      # Cursor
405)      cursor = cnx.cursor()
406)
407)      # Get Sequencer rows
408)      data = [tuple(d) for d in self.Sequencer]
409)
410)      # Delete last Sequencer
411)      drop_query = "DROP TABLE IF EXISTS Sequencer;"
412)      cursor.execute(drop_query)
413)
414)      # Commit
415)      cnx.commit()
416)
417)      # Create a new table
418)      create_query = "CREATE TABLE Sequencer (TaskId INT, Priority INT, Latitude FLOAT, Longitude FLOAT, RoverId VARCHAR(255),
419)          Sender VARCHAR(255), Timeout FLOAT, Timestamp FLOAT, Distance FLOAT);"
420)
421)      # Insert Sequencer
422)      query = "INSERT INTO Sequencer (TaskId, Priority, Latitude, Longitude, RoverId, Sender, Timeout, Timestamp, Distance)
423)          VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s);"
424)
425)      # Commit
426)      cnx.commit()
427)
428)      # Close connection
429)      cnx.close()
430)
431)      # ----- SEND SEQUENCER (MQTT) -----
432)
433)      # ----- INITIAL MESSAGE -----
434)      msg = []
435)      msg['Info'] = "Sending"
436)
437)      # Serialize json
438)      msg = js.dumps(msg)
439)
440)      # Publish
441)      self.Mqtt.publish(client,cmdtopic,msg)
442)
443)

```

```

444)     # ----- SEND SEQUENCER -----
445)
446)     # List for saving Sequencer in json
447)     data = []
448)
449)     # Transform list to dictionary
450)     for d in self.Sequencer:
451)         seq = dict(zip(["Priority", "Latitude", "Longitude", "RoverId", "Sender", "Timeout", "Timestamp"], [d[1], d[2], d[3],
452)             d[4], d[5], d[6], d[7]]))
453)         data.append(seq)
454)
455)     # Mesage structure for Sequencer
456)     msg = {}
457)     msg['documents'] = data
458)
459)     # Serialize json
460)     json = js.dumps(msg)
461)
462)     # Publish
463)     self.Mqtt.publish(client, cmdtopic, json)
464)
465)     # ----- SERIAL MESSAGES (TABLE FORMAT) -----
466)     print("\nSEQUENCER")
467)
468)     # Sequencer table
469)     print('\n' + tabulate(self.Sequencer, headers=["ID", "Priority", "Lat", "Long", "RoverID", "Sender", "Timeout",
470)         "Timestamp", "Distance"])+'\n' )

```

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]

        # 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 ALREADY...\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<sup>5</sup> 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”.

---

<sup>5</sup> La distancia se ha calculado con la fórmula de Haversine, dadas la latitud y longitud del globo planetario

## 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:

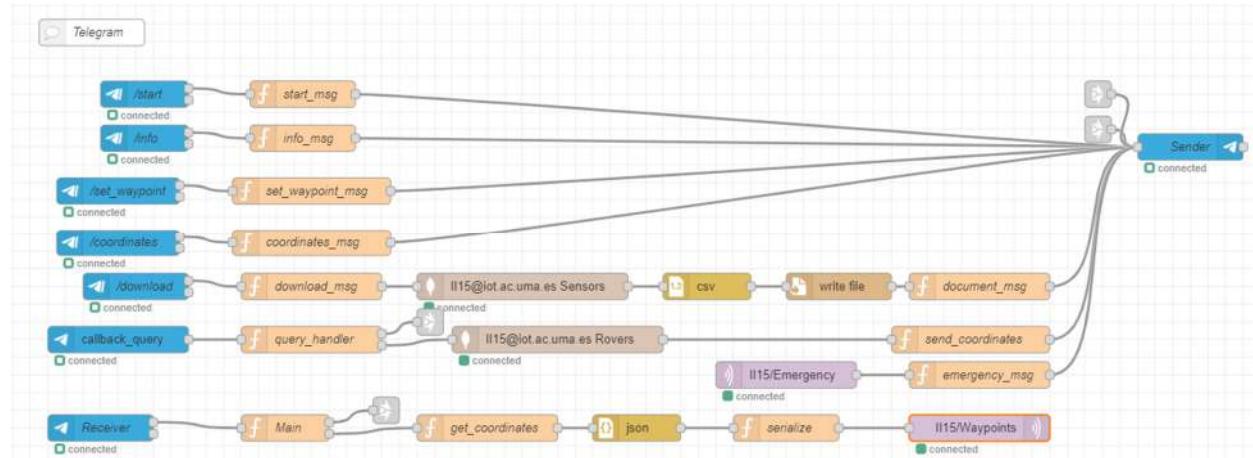


Fig. 8 Flujo

## 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.

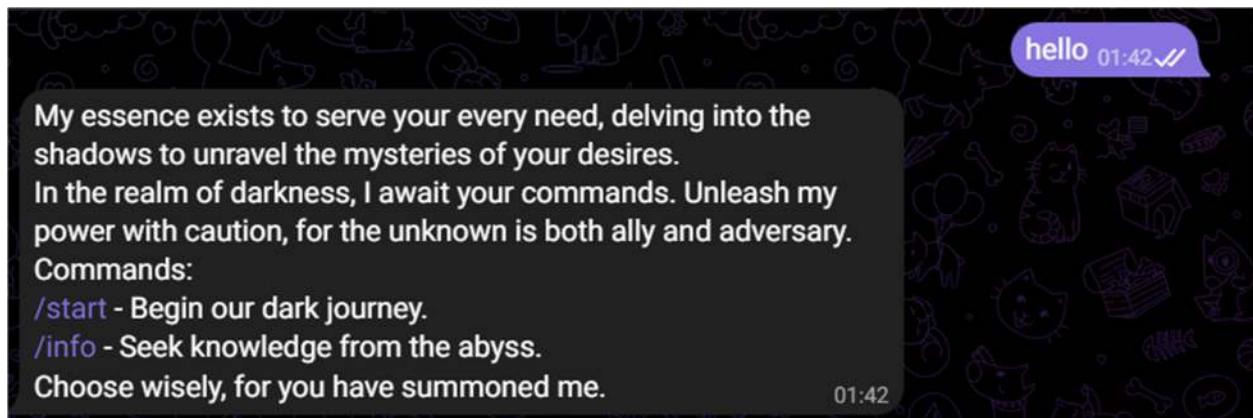


Fig.

```
Name: Main
Setup | On Start | On Message | [ ]
```

```
1 flow.set("msgid", msg.payload);
2
3 //determine if it is not a command
4 if (msg.payload.content.substring(0, 1) != "/") {
5
6 // determine if it is a coordinate
7 if (msg.payload.content.substring(0, 1) == "[") {
8     msg.payload.coordinates = msg.payload.content;
9     // send the confirmation of coordinate reception
10    msg.payload.content = "Your cosmic coordinates have been dispatche
11    // send message and start query node simultaneously
12    return [msg,msg]
13
14 // send a generic message if it is none of the above
15 } else {
16     msg.payload.content = "My essence exists to serve your every need, del
17     msg.payload.content = msg.payload.content +"In the realm of darkness,
18     msg.payload.content = msg.payload.content + "Commands:\n/start - Begin
19     msg.payload.content = msg.payload.content + "Choose wisely, for you ha
20     return [msg,null];
21 }
22 }
```

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.

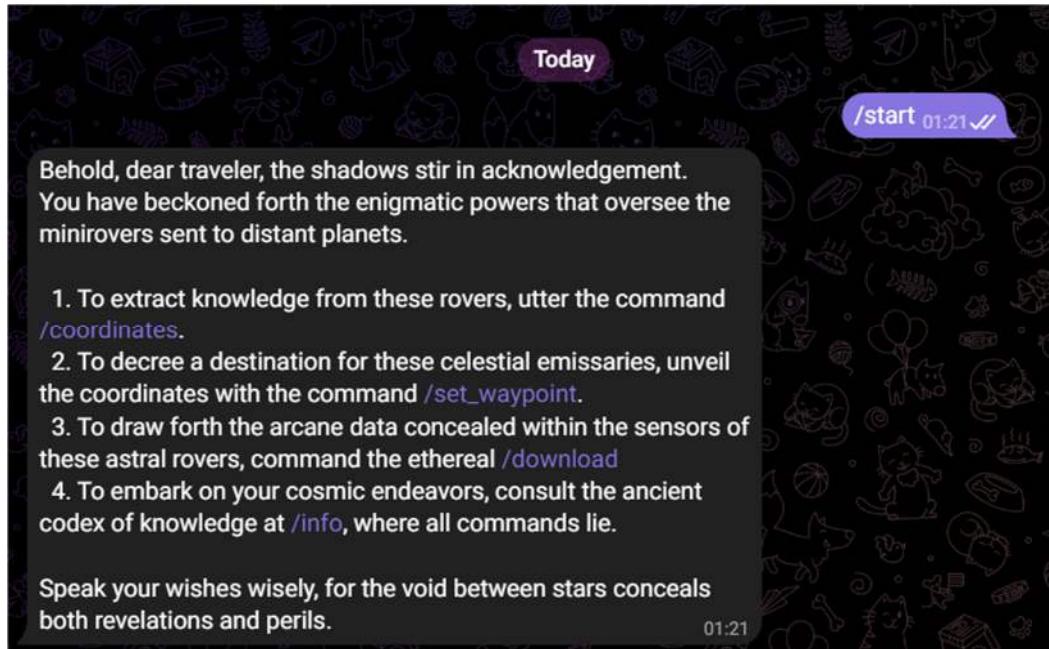


Fig. 10. Mensaje de Telegram

# Coordinates

/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.

```

1 //Create the options JSON to display
2 var opts = {
3   reply_markup : JSON.stringify({
4     "inline_keyboard": [
5       [
6         {
7           "text": "Sensor",
8           "callback_data": "2"
9         },
10        {
11          "text": "Actuator",
12          "callback_data": "3"
13        ]
14      ]
15    })
16  };
17 msg.payload.content = 'Select the emissary that shall traverse the cosmic tapestry.';
18 // set the options object to the created JSON
19 msg.payload.options = opts;
20 return msg;

```

Fig. 12. Flujo

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

```

17 if (msg.roverId.content == 2){ // Sensor from /coordinates
18   msg.payload = {"Sender": "SensorRover"};
19   msg.limit = 1;
20   //msg.payload.content = "The coordinates of this robot are:\n";
21   //msg.payload.content += "[x1,y1]";
22   return [null, msg];
23 }
24 if (msg.roverId.content == 3) { // Actuator from /coordinates
25   msg.payload = {"Sender": "ActuatorRover"};
26   msg.limit = 1;
27   //msg.payload.content = "The coordinates of this robot are:\n";
28   //msg.payload.content += "[x2,y2]";
29   return [null, msg];
30 }

```

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.

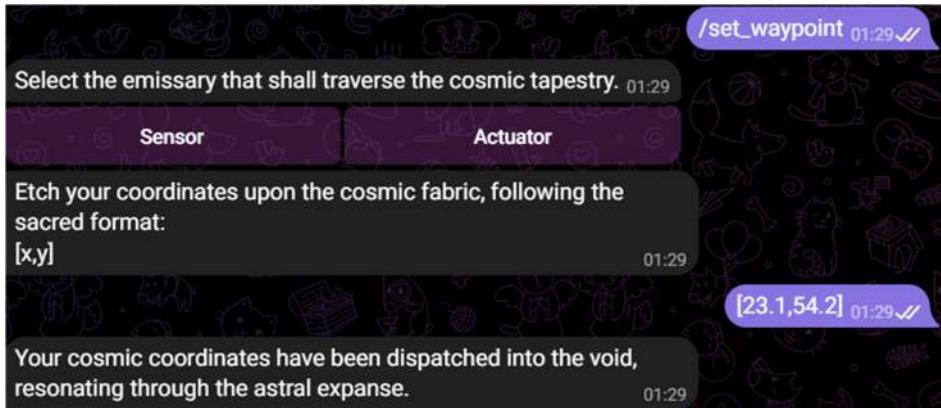


Fig. 14. Mensaje de Telegram

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.

The screenshot shows a code editor with a tab labeled 'query\_handler'. Below it is a flowchart component with four tabs: 'Setup', 'On Start', 'On Message', and 'On Stop'. The 'On Message' tab contains the following JavaScript code:

```
3 if (msg.roverId.content == 0) { // Sensor from /set_waypoint
4   flow.set("RoverId", "SensorRover");
5   msg.payload.content = "Etch your coordinates upon the cosmic fabric, following the sacred format:\n";
6   msg.payload.content += "[x,y]";
7   msg.payload.type = "message";
8   return [msg, null];
9 }
10 if (msg.roverId.content == 1) { // Actuator from /set_waypoint
11   flow.set("RoverId", "ActuatorRover");
12   msg.payload.content = "Etch your coordinates upon the cosmic fabric, following the sacred format:\n";
13   msg.payload.content += "[x,y]";
14   msg.payload.type = "message";
15   return [msg, null];
}
```

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.

Finalmente, se serializa y manda a través del chatbot.

The screenshot shows a Node-RED flow with a single node named "serialize". The node has three tabs: "Properties", "Name" (set to "serialize"), "Setup", and "On Start" (selected). The "On Message" tab is also visible. The code in the setup tab is:

```
1 var tiempo = new Date();
2 var timestamp = tiempo.getTime();
3 msg.payload =
4 {
5   "Sender": "User",
6   "RoverId": flow.get("RoverId"),
7   "Latitude": msg.payload[0],
8   "Longitude": msg.payload[1],
9   "Timestamp" : timestamp
10 };
11 return msg;
12
```

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.

The screenshot shows a Node-RED flow with a single node named "document\_msg". The node has three tabs: "Properties", "Name" (set to "document\_msg"), "Setup" (selected), and "On Start". The "On Message" tab is also visible. The code in the setup tab is:

```
1 msg.payload = msg.download;
2 msg.payload.content = msg.document;
3 msg.payload.type = "document";
4 return msg;
```

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.

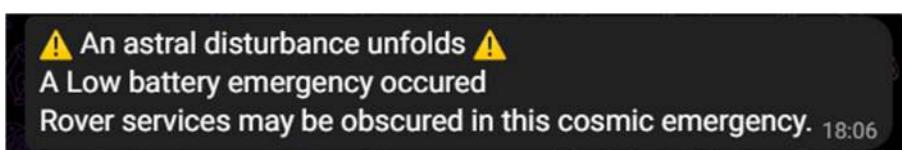


Fig. 18. Mensaje de Telegram

The screenshot shows the properties panel for a Node-RED node named 'emergency\_msg'. The 'Name' field is set to 'emergency\_msg'. Below it, there are four tabs: 'Setup' (selected), 'On Start', 'On Message', and 'On Stop'. The 'Setup' tab contains the following JavaScript code:

```

1 // Initialize previousStatus using context
2 var previousStatus = context.get('previousStatus') || null;
3
4 // Check if Status has changed
5 if (msg.payload && msg.payload.Status !== previousStatus) {
6     // Update previousStatus with the current value
7     previousStatus = msg.payload.Status;
8     // Store the updated previousStatus in context
9     context.set('previousStatus', previousStatus);
10
11    // Send a new message
12    msg.payload = flow.get("msgid");
13    msg.payload.content = "⚠ An astral disturbance unfolds ⚠";
14    msg.payload.content += "\nA " + String(previousStatus) + " emergency occurred";
15    msg.payload.content += "\nRover services may be obscured in this cosmic emergency.";
16    msg.type = "message";
17
18    return msg;
19 }
20
21 // Status has not changed, do not send a new message
22 return null;

```

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.

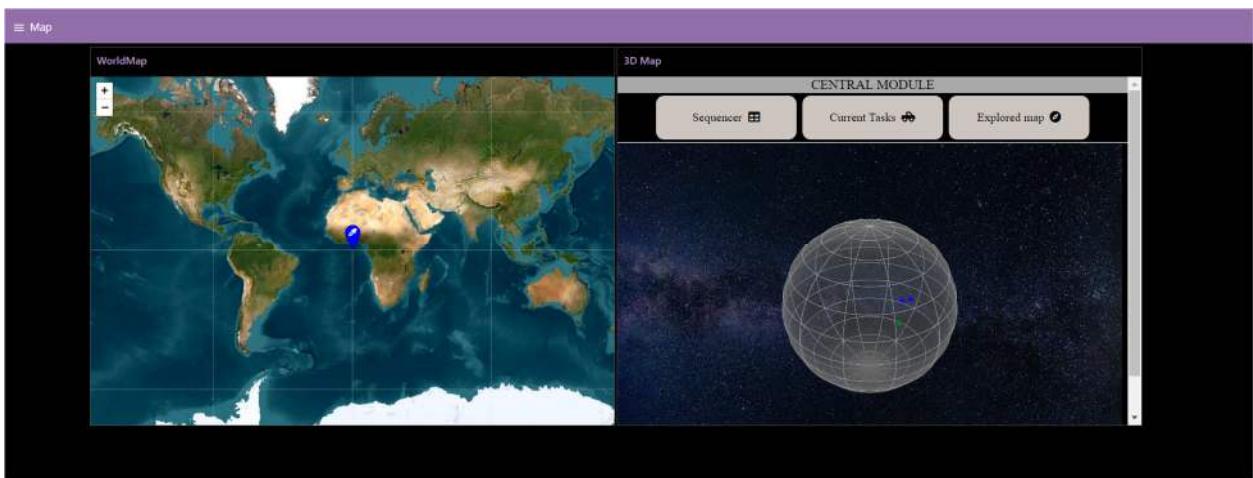


Fig. 21. Dashboard, Mapas

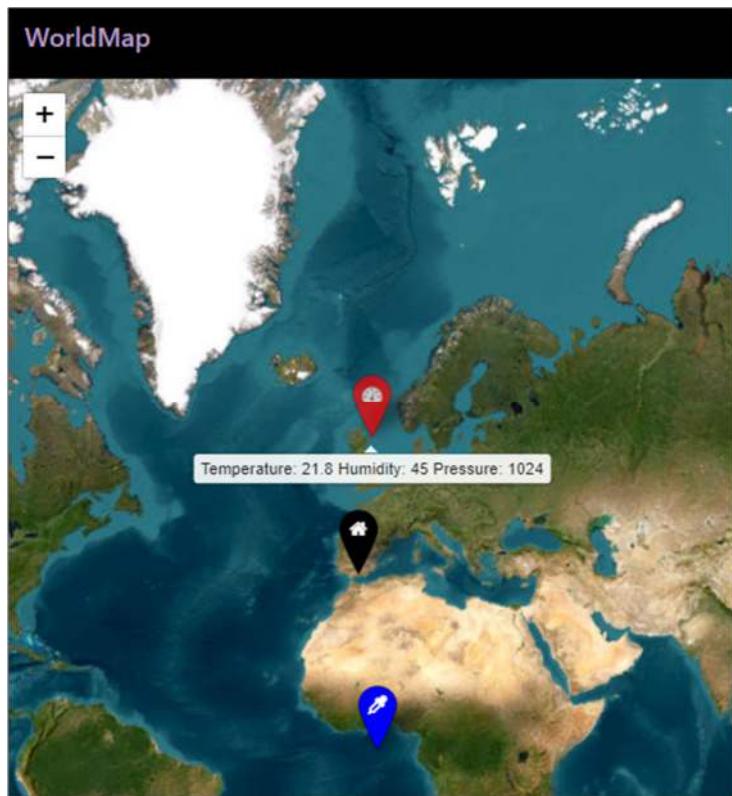


Fig. 20. Dashboard, detalle de Mapa

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.

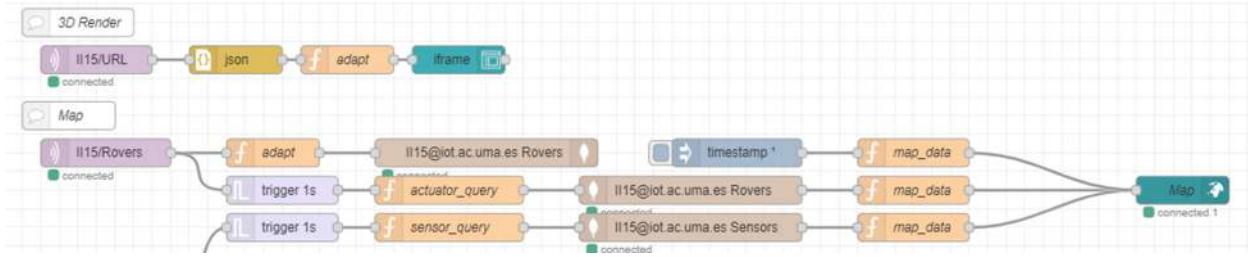


Fig. Para calcular las coordenadas del mapa, se utiliza el topic /Rovers en el caso del actuador, y /Sensors en el caso del sensor. Ambos se adaptan y guardan en sus respectivas colecciones de mongodb, para después consultarse y adaptar a los campos del mapa.



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:



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

Muestra la tabla de comandos. Se puede filtrar por fecha, rover y prioridad.

Commands							
Lists							
Apply Filters							
Sender	RoverId	Timestamp	Latitude	Longitude	Priority	Timeout	
User	SensorRover	1704047653	10.6	-12.1	1	2380960980998	
Auto	SensorRover	1703956870	0.6	7.1	2	2380960980998	
Central	ActuatorRover	1704233390	36.7	-4.42	1	2380960980998	
SensorRover	ActuatorRover	1704233523	-8.9	50.6	3	2380960980998	
User	ActuatorRover	1704141475	21.3	43.1	2	2380960980998	
Auto	ActuatorRover	1703956875	-19.3	-43.4	4	2380960980998	

Fig. 24. Dashboard. Comandos

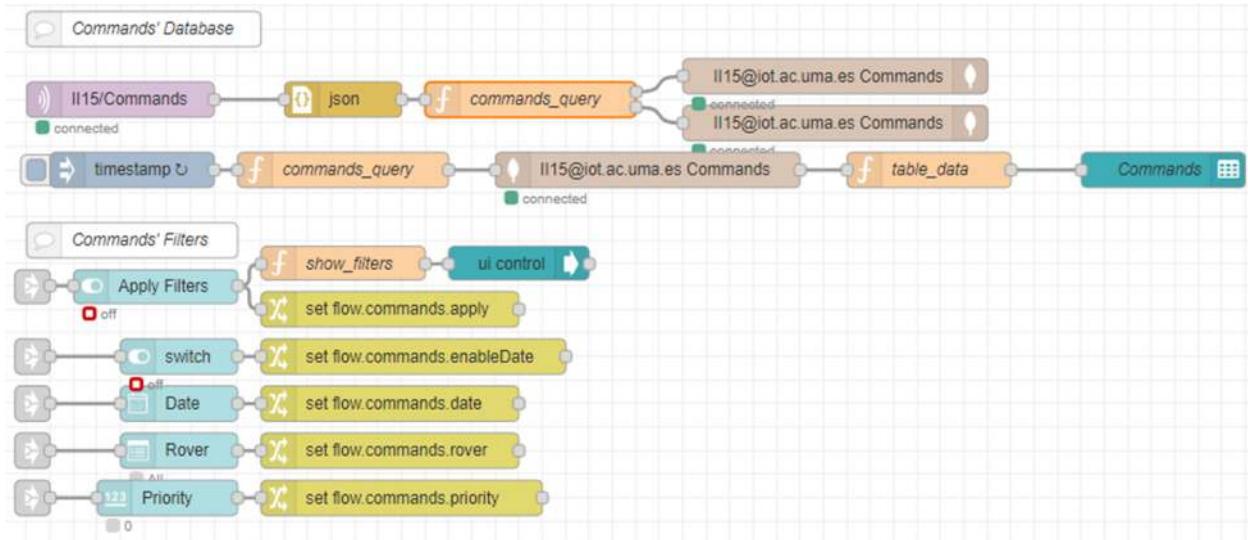


Fig. Los datos de esta tabla tienen la peculiaridad de insertarse todos juntos, y eliminarse cada vez que entran datos nuevos. Por ello, la forma de actualizarse es diferente a la habitual.

```

Name: commands_query
Setup: 
On Start: 
On Message:
1 if (msg.payload.Info == "Sending") {
2   msg = {}
3   msg.payload = {}
4   return [msg.payload, null]
5 } else if (msg.payload) {
6   var documents = msg.payload.documents;
7   msg = {};
8   msg.payload = documents;
9   return [null, msg.payload];
10 }

```

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:

```

1  if(msg.payload){
2      msg.payload = {"group":{"show":["Commands_Filters"]}};
3  }else{
4      msg.payload = { "group": { "hide": ["Commands_Filters"] }};
5  }
6  return msg;

```

Fig. 26. Flujo

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

```

1  var newMsg = msg.payload;
2  var applyFilter = flow.get("commands.apply");
3  var roverFilter = flow.get("commands.rover");
4  var priorityFilter = flow.get("commands.priority");
5  var dateFilter = flow.get("commands.date");
6  var enableDate = flow.get("commands.enableDate");
7  var sentDate; var sentRover; var sentPriority;
8
9  if (applyFilter)
10 {
11     if (enableDate) {
12         sentDate = {"Timestamp":String(dateFilter)};
13     } else {
14         sentDate = {};
15     }
16     if (roverFilter != "All") {
17         sentRover = {"RoverId":roverFilter};
18     }else{
19         sentRover = {};
20     }
21     if (priorityFilter != 0){
22         sentPriority = {"Priority":priorityFilter};
23     }else{
24         sentPriority = {};
25     }
26     msg.payload = { $and: [sentDate,sentRover,sentPriority]};
27 }else{
28     msg.payload = {};
29 }
30 return msg;

```

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.

## Sensors

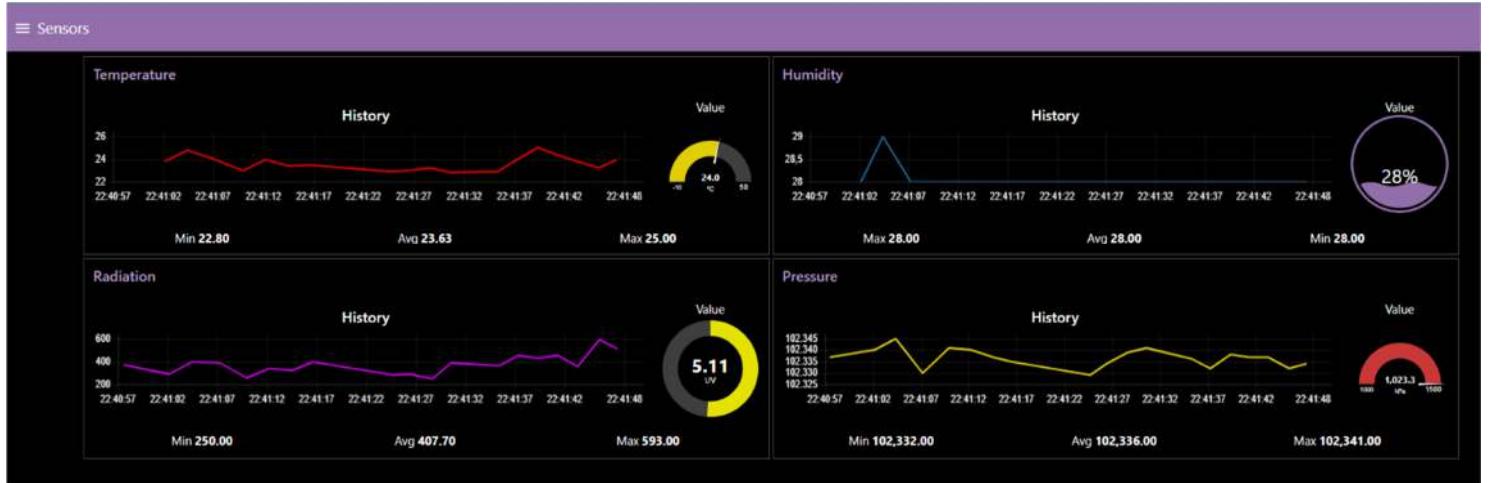


Fig. 28. Dashboard, sensores

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.

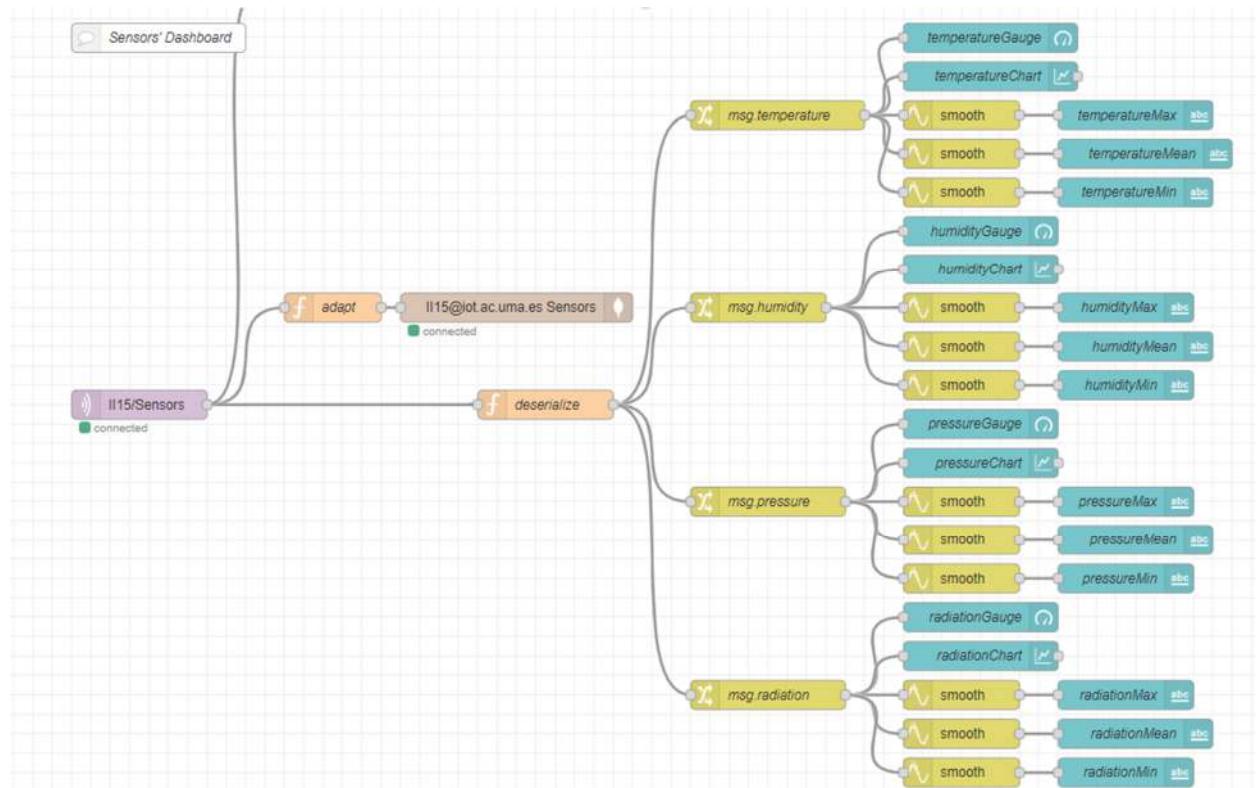


Fig. 29. Flujo

## Emergency

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

The screenshot shows a table titled "List" with columns: Sender, Latitude, Longitude, Status, and Date. The data consists of 20 rows, all of which are "ActuatorRover" entries. The "Status" column contains values like "Sample overflow", "Low battery", "Earthquake", and "Earthquake". The "Date" column shows "1/1/1970" for all rows. To the right of the table is a "Filters" sidebar. It includes a date picker set to "01/01/1970", a dropdown for "Rover" (set to "Actuator"), and a dropdown for "Status" (set to "All"). Below these are three buttons: "Low battery", "Earthquake", and "Sample overflow".

Fig. 30. Dashboard, emergencias

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.

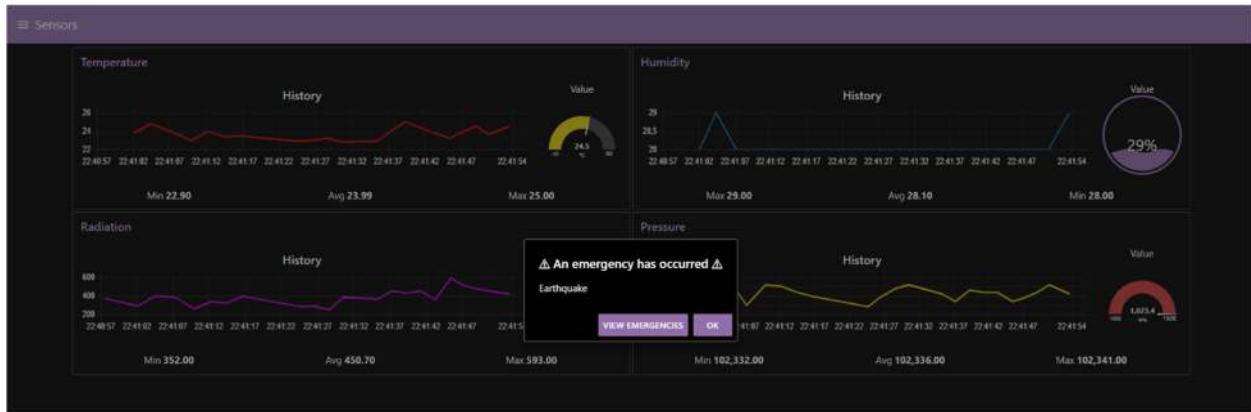


Fig. 31. Dashboard. Notificación de emergencia

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”.

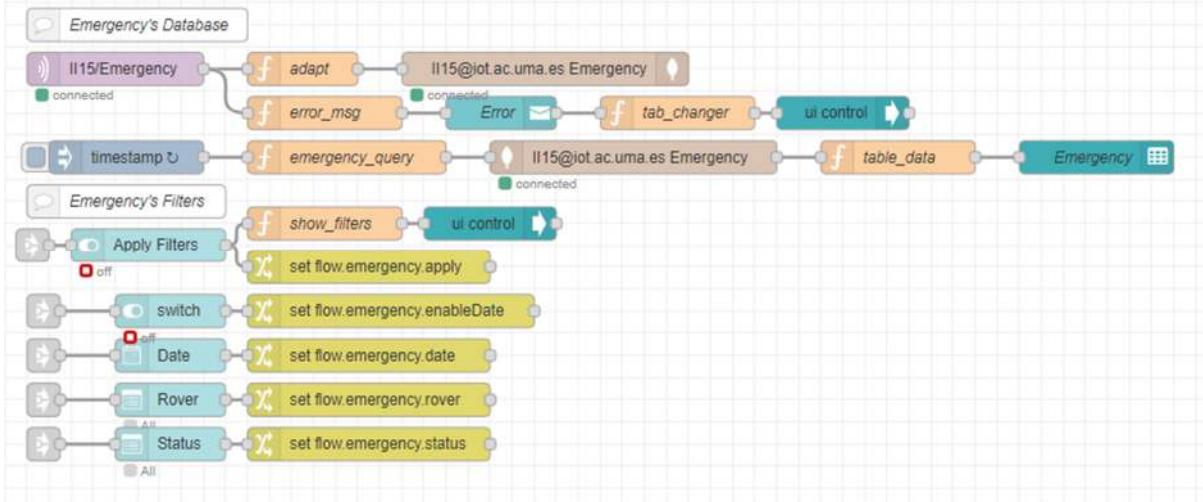


Fig. 32. Flujo

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

**Properties**

Name: tab\_changer

**Setup**

```

1 if (msg.payload == "View emergencies") {
2   msg.payload = {"tab": "Emergency"}
3   return msg;
4 }
```

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>