Código principal

Librerías

Inicialización mqtt

Inicialización mongo

Puerto serie

Diccionario

- Roversinfo: información puerto serie
- newroverinfo: flag
- Status: permite identificar si la info 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

(((Clases y funciones de clase)))

HandleMqtt

- Connect_mqtt
    - On_connect
- Publish
- Disconnect_mqtt

HandleSerial

- read_serial(): lee byte a byte hasta que se encuentra un "\r". Llegado ese punto, se procesa
- process_data(): si ack, entonces deserializa, comprueba que status sea distinto al anterior y, de ser así, activa newroverinfo y desactiva ack. De lo contrario, desactiva newroverinfo
- send_data(): se le pasan la variable de datos y, si no está vacío, se envía.

SequencePlanner

- __init__(): constructor con
    - Listas
        - Secuenciador (Taskid, priority, latitude,longitude,roverid, sender,timeout,timestamp,distance)
        - roversId: rovers disponibles
        - roversTask (RoverId, Current task id, current task latitude, current task longitude)
        - explored: (roverid,lati,long,last time explored,timestamp)
        - Locations: (roverid, current lat, current long)
    - Variables
        - Lat y long rovers max
        - Lat long home
        - Direction (1-> heads Distance 1 north & distance 1 east)

        - Usercmd
        - Sensorcmd
        - Status
        - Autocmds
        - Homónimas new_
        - Pending: flag de tareas pendientes por mandar
        - Task_id
        - SerialModule → objeto HandleSerial
        - Mqtt → objeto HandleMqtt

- Updaterovers(): actualiza lista de rovers desde mysql
- Process_location(): actualiza ubicación real rovers en la lista location con un query a mongo
- Get_data(self):
    - Coge usercmd de la colección
    - Roversinfo de serial
    - Checkdata()
        - Si llega un usercmd y el timestamp es mayor, new_usercmd=true
        - Si roversinfo llega y tiene interesting location en el status, entonces es new_sensorcmd y si no new_status. Se activa ack para poder procesar
    - Guardar comandos en sus respectivas variables.
- Auto_plan
    - Update_rover_task
        - Si un rover está en task finished, se introduce como parámetro de entrada su nombre. La función extrae todos los rovers que tienen tareas asignadas y se asigna una tarea nueva al rover que esté en dicho “task finished”
        - Compruebo tareas automáticas
        - Cojo los roverid que tengan tareas automáticas, y busco cuáles no tienen
        - Si alguno no tiene, se comprueba si tiene una tarea actual y, de ser así se toman sus coordenadas para asignar el plan automático. De no tener tarea actual, se toman las coordenadas actuales del robot para la asignación del plan automático
    - Se generan unas coordenadas aleatorias, manteniendo una coordenada con el mismo valor y otra se varía entre 0 y 1.
    - Se acotan las latitudes máximas y mínimas, cambiando la dirección en caso de exceso
    - Se comprueba si el sitio se ha explorado en los últimos 5 días, en cuyo caso se vuelve a generar.
    - Se crea una lista con todos los valores automáticos que se han generado y se establece el new_autocmd a true
- Compute_priority
    - Si hay estado de emergencia
        - Cojo las coordenadas actuales para hacer el timeout
        - Estructuro el mensaje
        - Busco los índices del secuenciador que correspondan con tareas del rover en cuestión y, escojo la más prioritaria, para insertar el nuevo comando delante de la misma.

        - Update_priority(): si hay que borrar, se decrementa la prioridad, y si hay que actualizar, se incrementa
    - Si hay nuevo comando de usuario
        - Actualizo task_id
        - Estructuro el mensaje
        - Si hay tarea asignada, me toma las coordenadas de dicha tarea, si no hay asignada, se toman las actuales.
        - Se computa el valor de timeout
        - Cojo todos los comandos del secuenciador que sean del rover, cojo los índices de esas tareas y cojo las distancias de esa tareas.
        - Introduzco la nueva distancia en una lista, ordeno la lista de distancias y tomo el índice mínimo, para luego introducir la tarea detrás del último comando de central y en el índice de distancia que le corresponde, así como la prioridad
    - Si hay nuevo comando de sensor
        - Compute_roverid():
            - Si el status es interesting location, se borra y se separan (Split) las palabras que vengan detrás, asignando cada una a un flag.
            - De la lista de rovers con su respectivo flag, se seleccionan aquellos que correspondan a los flags recibidos
            - Elimino los duplicados
        -
        - Si hay comandos anteriores del sensor, lo ordeno por distancia
        - Si no, y hay automáticas, lo meto delante de ellas. De no haber automáticas, la asigno al final.
    - Si hay nuevo comando automático
        - Si hay tareas, se asigna al final
        - Si no, se crea la primera.
    - Guardamos secuenciador en BBDD
    - Envia por mqtt con un “info”:sending primero y después la lista completa
- Send_cmd
    - Si el estado es de emergencia o ha terminado una tarea, envía un comando.
    - Si ha terminado una tarea,
        - Update_explored:Actualiza explrorados
            - Actualiza la lista
            - Actualiza la última vez que ha sido explorado
        - Update_sequencer:Actualiza secuenciador
        - se reduce el orden de prioridad y se elimina la tarea anterior.
        - Si el rover no tiene tareas, activo el pending
    - Serializa
    - Invoca a update_rover_task
    - Update_distances
        - Cojo la tarea y coordenadas actuales y ejecuto compute_distance(), que computa la distancia entre coordenadas según la fórmula Haversine

    - Si se ha enviado el dato, se actualiza pending a false y new_status también.
- Send_timeout()
    - Si el timestamp es mayor al timeout, se envía un mensaje de emergencia con status "Timeout"
- Serialize_msg()
    - Defino los campos
    - Creo el timestamp
    - Realizo el compute_timeout
    - Compute_distance(): computa la distancia entre coordenadas según la fórmula Haversine
    - Calculo el rover que puede hacer la tarea en función de la disponibilidad y la distancia.
- Loop
    - Updaterovers
    - Processlocation
    - Getdata
    - Autoplant
    - Compute priority
    - Comprueba estado
    - Si new_status o pending
        - Send_cmd

. - SerialRead()

    - Crea objeto serial
    - Lee y guarda en roversinfo
- Main
    - Crea threads de central y serial

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.