Práctica 1 POO

Alejandro Guadalupe García

Índice

Introducción

Usando la red Meshtastic conseguir comunicarse a través de ella fácilmente gracias a un menú, transformar el código dado aun modelo de clases y capturar los datos y mensajes recibidos y exportarlos a un archivo json.

Github

En github se han hecho varios commits a lo largo del trabajo en el programa con las actualizaciones que se realizaron y la descripción de ellas.

Enlace al Github

Objetivos

- Rediseñar un código basándolo en clases.
- Obtener los mensajes a través de MQTT y Meshtastic.
- Crear un menú para elegir si se quiere usar la red de Meshtastic o obtener datos a través de MQTT.
- Guardar coordenadas y mensajes en un archivo json.
- Guardar datos de sensores.

Alejandro Guadalupe García

Metodología

1. Modificar el código dado y transformarlo en uno basado en dos clases (Dispositivo y Comunicador).

Clase Dispositivo. Para crear la clase se crea el constructor y se declaran los atributos. En este caso la función set_topic esta en la clase Dispositivo, para crearla se han eliminado las variables globales y declarado con el puntero self.

```python
class Dispositivo:
    def __init__(self, root_topic, channel, debug):
        self.root_topic = root_topic
        self.channel = channel
        self.debug = debug
        self.random_hex_chars = "a8a1"
        self.node_name = '!abcd' + self.random_hex_chars
        self.node_number = int(self.node_name.replace("!", ""), 16)
        self.set_topic()

    def set_topic(self):
        # Prepara variables para MQTT
        if self.debug: print("set_topic")
        self.node_name = '!' + hex(self.node_number)[2:] # Identificador del nodo, en hexadecimal  para MQTT
        self.subscribe_topic = self.root_topic + self.channel + "/#" # Donde escuchar
        self.publish_topic = self.root_topic + self.channel + "/" + self.node_name # Donde publicar
```

Archivo config.json, en este archivo se guarda toda la configuración, después se llama desde el método constructor __init__ en cada clase y se declaran como atributos.

```json
{
    "mqtt_port": 1883,
    "mqtt_broker": "mqtt.meshtastic.org",
    "root_topic": "msh/EU_868/ES/2/e/",
    "mqtt_username": "meshdev",
    "mqtt_password": "large4cats",
    "channel": "TestMQTT",
    "key": "ymACgCy9Tdb8jHbLxUxZ/4ADX+BWLOGVihmKHcHTVyo=",
    "client_short_name": "AGG",
    "client_long_name": "Alejandro",
    "client_hw_model": 255,
    "message_text": "Hola, estoy conectado",

    "lat": "0",
    "lon": "0",
    "alt": "0",

    "BROKERsensores": "broker.emqx.io",
    "TOPICSsensores": ["sensor/data/sen55", "sensor/data/gas_sensor"],

    "debug" : true,
    "auto_reconnect" : true,
    "auto_reconnect_delay": 1,
    "print_service_envelope": false,
    "print_message_packet": false,
    "print_node_info": true,
    "print_node_position": true,
    "print_node_telemetry": true
}
```

Clase Comunicador. En el constructor se declaran los atributos guardados en config.json y se crea el objeto de la clase dispositivo.

```python
class Comunicador:
    def __init__(self):
        with open("static/config.json", "r", encoding="utf-8") as archivo:
            config = json.load(archivo)

        self.debug = config["debug"]
        self.auto_reconnect = config["auto_reconnect"]
        self.auto_reconnect_delay = config["auto_reconnect_delay"]
        self.print_service_envelope = config["print_service_envelope"]
        self.print_message_packet = config["print_message_packet"]

        self.print_node_info =  config["print_node_info"]
        self.print_node_position = config["print_node_position"]
        self.print_node_telemetry = config["print_node_telemetry"]

        self.lat = config["lat"]
        self.lon = config["lon"]
        self.alt = config["alt"]

        self.mqtt_port = config["mqtt_port"]
        self.root_topic = config["root_topic"]
        self.channel = config["channel"]
        self.key = config["key"]
        self.mqtt_broker = config["mqtt_broker"]
        self.mqtt_username = config["mqtt_username"]
        self.mqtt_password = config["mqtt_password"]
        self.message_text = config["message_text"]

        self.client_short_name = config["client_short_name"]
        self.client_long_name = config["client_long_name"]
        self.client_hw_model = config["client_hw_model"]

        self.global_message_id = random.getrandbits(32)

        self.client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2, client_id="", clean_session=True, userdata=None)
        self.dispositivo = Dispositivo(self.root_topic, self.channel, self.debug)
```

En el método on_connect se llama a un método del objeto dispositivo creado en el constructor.

```python
def on_connect(self, client, userdata, flags, reason_code, properties):
    self.dispositivo.set_topic()
    if self.client.is_connected():
        print("client is connected")

    if reason_code == 0:
        if self.debug: print(f"Connected to sever: {self.mqtt_broker}")
        if self.debug: print(f"Subscribe Topic is: {self.dispositivo.subscribe_topic}")
        if self.debug: print(f"Publish Topic is: {self.dispositivo.publish_topic}\n")
        self.client.subscribe(self.dispositivo.subscribe_topic)
```

En el método on_message para que siga funcionando sin las variables globales se han cambiado las variables por estos atributos creados en el constructor. Lo mismo se ha hecho en el resto de métodos de la clase Comunicador.

```
def on_message(self, client, userdata, msg):
    # Interpreta los mensajes recibidos de meshtastic a través de MQTT
    se = mqtt_pb2.ServiceEnvelope()
    try: # Se asegura de que el mensaje es correcto
        se.ParseFromString(msg.payload)
        if self.print_service_envelope:
            print ("")
            print ("Service Envelope:")
            print (se)
        mp = se.packet
        if self.print_message_packet:
            print ("")
            print ("Message Packet:")
            print(mp)
    except Exception as e:
        print(f"*** ServiceEnvelope: {str(e)}")
        return

    if mp.HasField("encrypted") and not mp.HasField("decoded"): # Si el me
        self.decode_encrypted(mp)

    # Attempt to process the decrypted or encrypted payload
    portNumInt = mp.decoded.portnum if mp.HasField("decoded") else None #
    handler = protocols.get(portNumInt) if portNumInt else None # Obtiene

    pb = None
    if handler is not None and handler.protobufFactory is not None:
        pb = handler.protobufFactory()
        pb.ParseFromString(mp.decoded.payload)

    if pb:
        # Clean and update the payload
        pb_str = str(pb).replace('\n', ' ').replace('\r', ' ').strip()
        mp.decoded.payload = pb_str.encode("utf-8")
    print(mp)
    #print(mp.decoded.payload.decode("utf-8"))
    #print(mp.decoded.portnum)
```

2. Añadir métodos para el guardado de los datos y mensajes, y para el refresh del terminal en la clase Dispositivo.

En este caso este método es estático y guarda cualquier dato recibido en el archivo que guarde el argumento dado. Los archivos donde se guardan se crean si no existen.

```python
@staticmethod
def guardarDatos(nuevo_datos, nombreArchivo):
    try:
        with open(nombreArchivo, "r", encoding="utf-8") as archivo:
            try:
                datosExistentes = json.load(archivo)
                if not isinstance(datosExistentes, list):
                    datosExistentes = [datosExistentes]
            except json.JSONDecodeError:
                datosExistentes = []
    except FileNotFoundError:
        datosExistentes = []

    datosExistentes.append(nuevo_datos)

    with open(nombreArchivo, "w", encoding="utf-8") as archivo:
        json.dump(datosExistentes, archivo, indent=4, ensure_ascii=False)
```

```python
@staticmethod
def clearConsole():
    command = 'clear'
    if os.name in ('nt', 'dos'):
        command = 'cls'
    os.system(command)
```

3. Crear una nueva clase Interfaz con el menú y la llamada a la clase Comunicador en forma de objeto.

```
class InterfazTerminal:
    def __init__(self):
        self.ordenador = Comunicador()

        with open("static/config.json", "r", encoding="utf-8") as archivo:
            config = json.load(archivo)
        self.BROKERsensores = config["BROKERsensores"]
        self.mqtt_port = config["mqtt_port"]
        ordenadorSensores = ComunicadorSensores()

        self.ordenador.client.on_connect = self.ordenador.on_connect
        self.ordenador.client.on_disconnect = self.ordenador.on_disconnect
        self.ordenador.client.on_message = self.ordenador.on_message

        self.client = mqtt.Client()
        self.client.on_connect = ordenadorSensores.on_connect
        self.client.on_message = ordenadorSensores.on_message

        self.ordenador.connect_mqtt()
        print("Escuchando...")

        time.sleep(1)
```

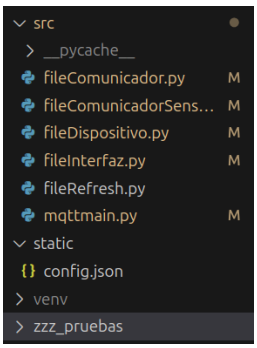
4. Crear otra clase para recibir datos de los sensores.

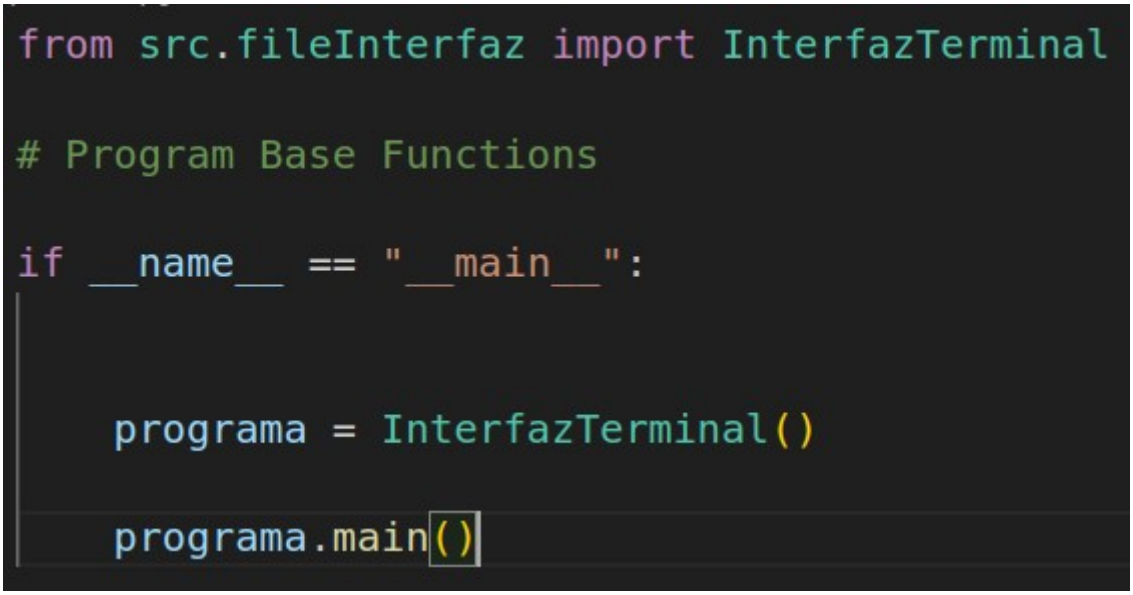
Para recibir datos desde los sensores correctamente se ha usado la misma metodología que en la clase comunicador.

Se crea un constructor y se llama al archivo config.json con los datos necesarios y se añade self. en los atributos que sean necesarios.

```python
class ComunicadorSensores:
    def __init__(self):
        with open("static/config.json", "r", encoding="utf-8") as archivo:
            config = json.load(archivo)
        self.broker = config["BROKERsensores"]
        self.port = config["mqtt_port"]
        self.topics = config["TOPICSsensores"]

    # Callback cuando se establece la conexión con el broker
    def on_connect(self, client, userdata, flags, rc):
        if rc == 0:
            print("Conexión exitosa al broker MQTT")
            # Suscribirse a los temas
            for topic in self.topics:
                client.subscribe(topic)
                print(f"Suscrito al tema '{topic}'")
        else:
            print(f"Error de conexión, código: {rc}")

    # Callback cuando se recibe un mensaje en los temas suscritos
    def on_message(self, client, userdata, msg):
        clearConsole()
        print("Ctrl+C para salir")
        print(f"Mensaje recibido en el tema '{msg.topic}':")
        print(msg.payload.decode("utf-8"))

        try:
            # Decodificar y convertir el mensaje de JSON a diccionario
            payload = json.loads(msg.payload.decode("utf-8"))
            print(json.dumps(payload, indent=4))  # Mostrar el mensaje formateado

            Dispositivo.guardarDatos(payload, nombreArchivo)
            #print(f"Datos guardados en {nombreArchivo}")

        except json.JSONDecodeError as e:
            print(f"Error decodificando JSON: {e}")
```

5. Crear un archivo para cada clase y uno para el main.

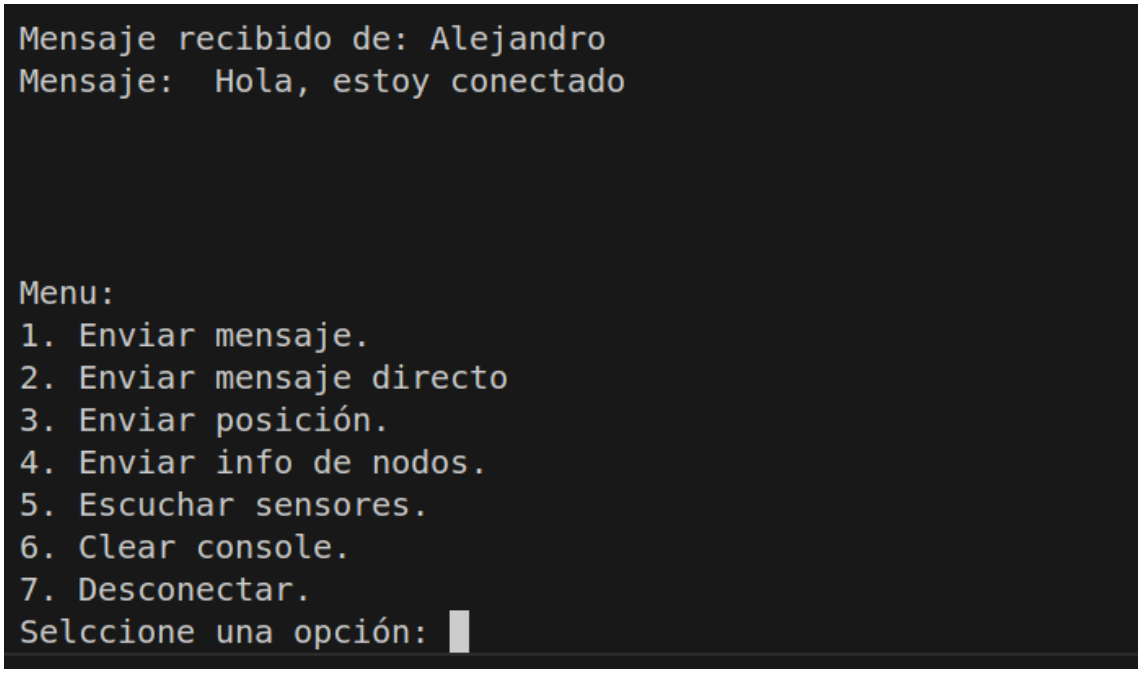


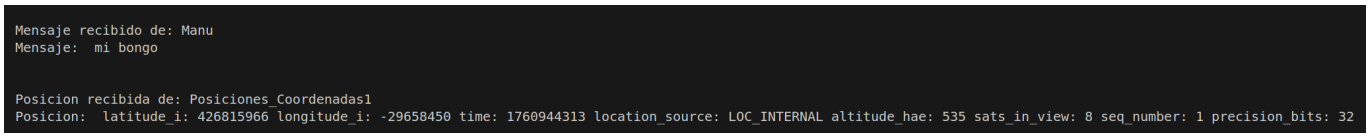


```python
from src.fileInterfaz import InterfazTerminal

# Program Base Functions

if __name__ == "__main__":


    programa = InterfazTerminal()

    programa.main()
```

Resultados

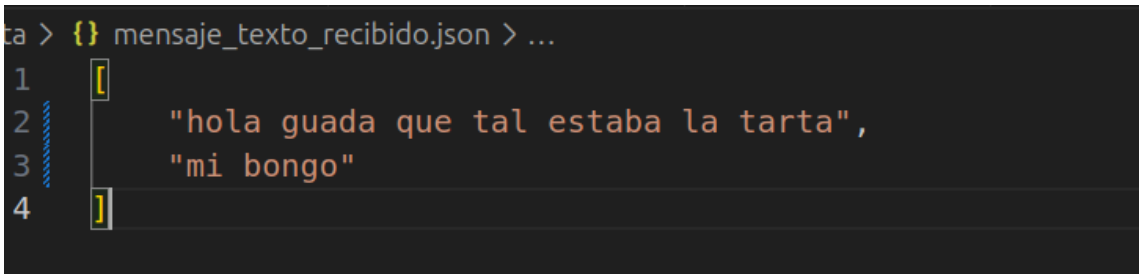
Consola.



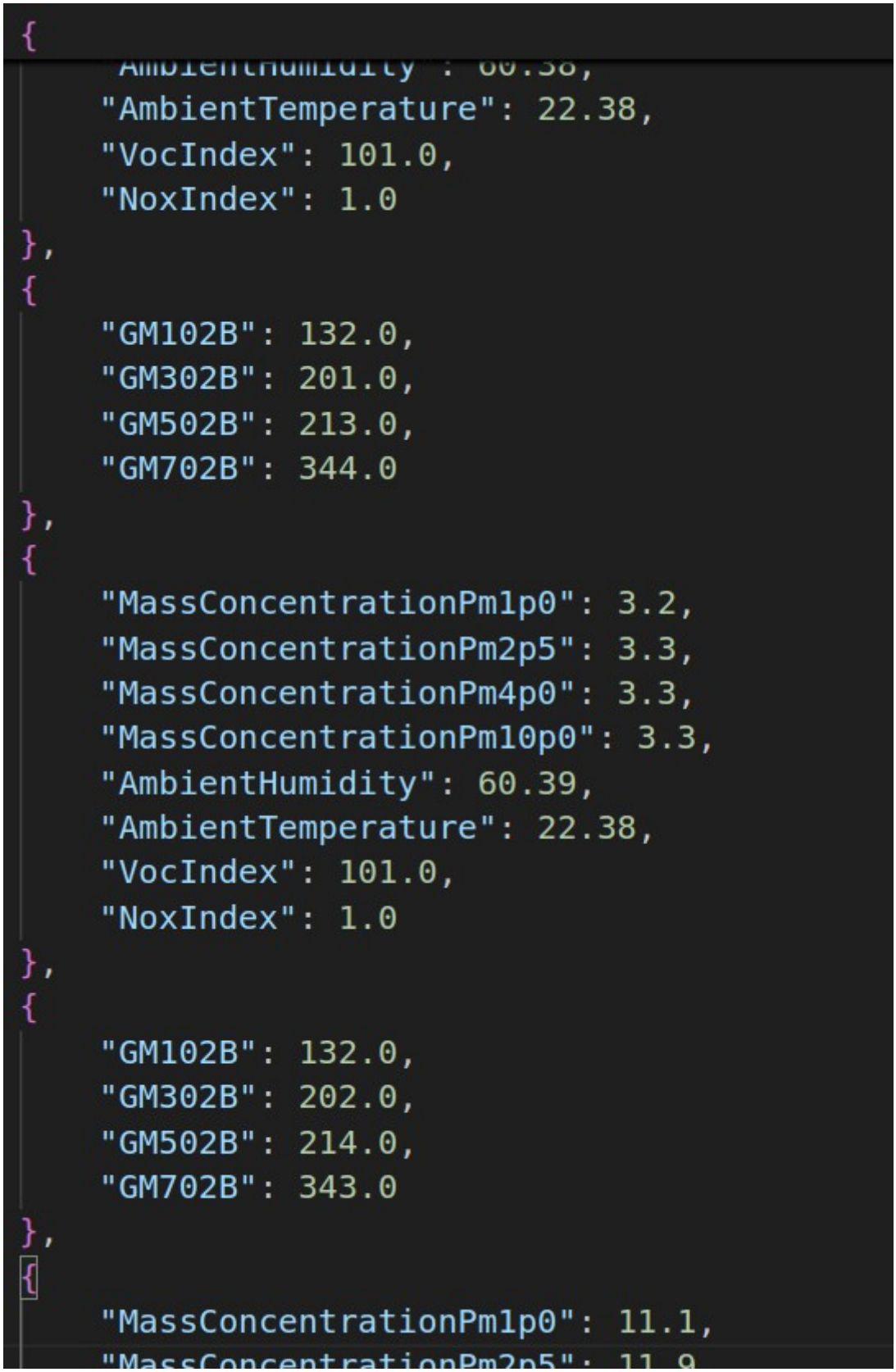


Los archivos .json se crean si no existen.

.json con mensajes:



.json con datos de los sensores:



Diseño de clases

1. Clase dispositivo

Esta clase crea el nodo, y el identificador de donde publicar y escuchar dados el canal y el tema raíz.

También se encarga del guardado de todos los datos recibidos por la clase Comunicador y comunicadorSensores.

2. Clase comunicador

Esta clase gestiona la comunicación, el enviar y el recibir mensajes, el conectarse y el desconectarse de la red Meshtastic, con los protocolos MQTT.

3. Clase comunicadorSensores

Esta clase escucha y muestra los datos recibidos de sensores por MQTT.

4. Clase interfazTerminal

Esta clase hace la llamada a las dos clases de comunicación y muestra un menú para que el usuario elija si enviar y recibir mensajes con el objeto creado de la clase comunicador a través de Meshtastic o recibir datos de los sensores a través de MQTT.

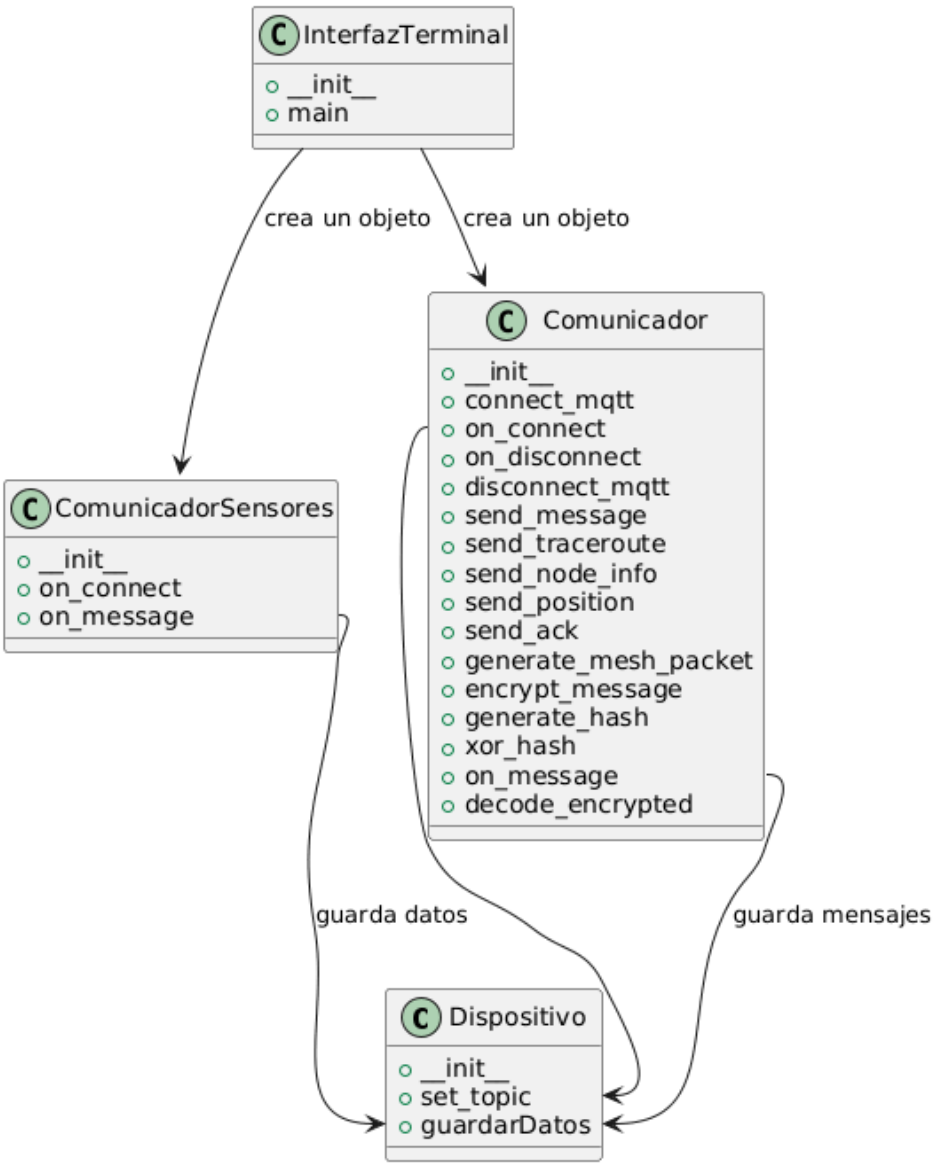
Como se relacionan las clases

Cada objeto de la clase comunicador crea en el constructor un objeto de la clase dispositivo para crear un nodo propio usando el método set_topic.

La clase comunicador crea un objeto de la clase comunicador para gestionar la red principal y uno de la clase comunicadorSensores para gestionar los datos de los sensores.

Las clases Comunicador y ComunicadorSensores usan el método estático guardarDatos para guardar mensajes y datos de los sensores.

Diagrama UML



Alejandro Guadalupe García

Extras

Guardado de contactos

Guarda los contactos en un json con su código y numero para poder elegir si enviarle un mensaje directo.

```python
@staticmethod
def guardarContactos(contacto, nombreArchivo, numero_nodo):
    try:
        with open(nombreArchivo, "r", encoding = "utf-8") as archivo:
            try:
                datosExistentes = json.load(archivo)
                if not isinstance(datosExistentes, list):
                    datosExistentes = [datosExistentes]
            except json.JSONDecodeError:
                datosExistentes = []
    except FileNotFoundError:
        datosExistentes = []

    # Verifica si el contacto existe
    for existente in datosExistentes:
        if contacto == existente["codigo"]:
            return

    nuevo_contacto = {"codigo": contacto, "numero": numero_nodo, "nombre": ""}
    datosExistentes.append(nuevo_contacto)
    print("Nuevo contacto conseguido")

    with open(nombreArchivo, "w", encoding = "utf-8") as archivo:
        json.dump(datosExistentes, archivo, indent = 4, ensure_ascii = False)
```

Para ponerle nombre al contacto, desde el archivo contactos.json, hay que rellenar el espacio del dato nombre.

```json
[
    {
        "codigo": "!abcda8a1",
        "numero": 2882381985,
        "nombre": "Alejandro"
    },
    {
        "codigo": "!bc868731",
        "numero": 3162933041,
        "nombre": "Datos_aparato_bateria"
    },
    {
        "codigo": "!2ae93dc9",
        "numero": 719928777,
        "nombre": "Posiciones_Coordenadas"
    }
]
```

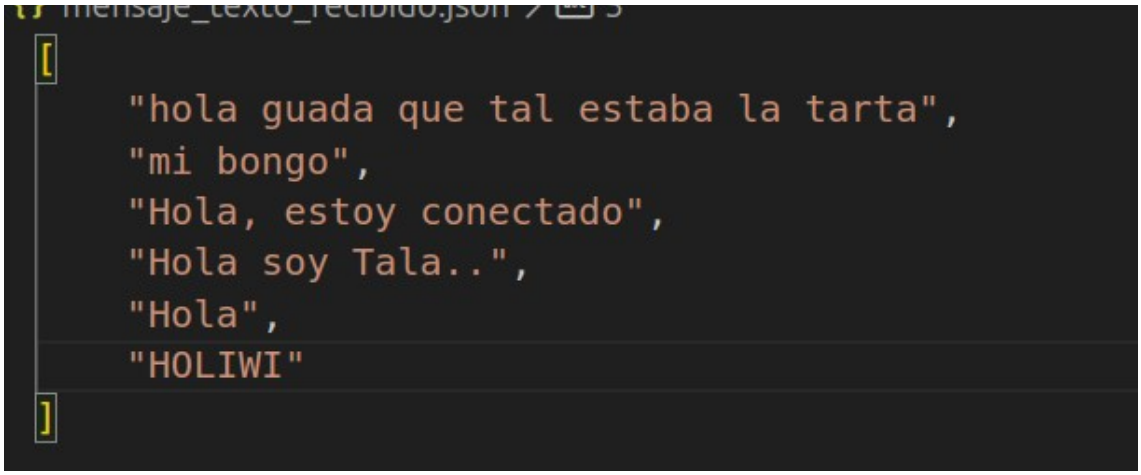
Envío de mensajes directos

```
Contacto a elegir (escribe el numero (0-...)):
Numero: 0 Nombre: Alejandro
Numero: 1 Nombre: WioTracker-001
Numero: 2 Nombre: JaimeMovil
Numero: 3 Nombre: CosaRara
Numero: 4 Nombre: Posiciones_Coordenadas2
Numero: 5 Nombre: Manu
Numero: 6 Nombre: Tala
Elige el numero del contacto:
```

Guardado de mensajes por tipo de mensaje

Los mensajes se guardan en un json de solo mensajes.

```python
if mp.decoded.portnum == 1:
    print("Mensaje recibido de:", contacto)
    print("Mensaje: ", mp.decoded.payload.decode("utf-8"))
    print("\n")
    # Mesaje de texto
    nombreArchivo = "data/mensaje_texto_recibido.json"
    self.dispositivo.guardarDatos(mp.decoded.payload.decode("utf-8"), nombreArchivo)

elif mp.decoded.portnum == 3:
    print("Posicion recibida de:", contacto)
    print("Posicion: ", mp.decoded.payload.decode("utf-8"))
    print("\n")
    # Mesaje de posición GPS
    nombreArchivo = "data/mensaje_posicion_recibido.json"
    self.dispositivo.guardarDatos(mp.decoded.payload.decode("utf-8"), nombreArchivo)

elif mp.decoded.portnum == 4:
    print("Telemetria recibida de:", contacto)
    print("Telemetria: ", mp.decoded.payload.decode("utf-8"))
    print("\n")
    # Mesaje de telemetria
    nombreArchivo = "data/mensaje_telemetria_recibido.json"
    self.dispositivo.guardarDatos(mp.decoded.payload.decode("utf-8"), nombreArchivo)

else:
    print("Mensaje de otro tipo recibido de:", contacto)
    print("Mensaje:", mp.decoded.payload.decode("utf-8"))
    print("\n")
```


mensaje_texto_recibido.json
[
"hola guada que tal estaba la tarta",
"mi bongo",
"Hola, estoy conectado",
"Hola soy Tala..",
"Hola",
"HOLIWI"
]

Conclusiones

La aplicación recibe y envía mensajes y funciona, y recibe datos de los sensores, sin problemas y los muestra en el terminal.

Al finalizar la práctica, se han logrado avances en el uso de las clases, la comunicación a través de MQTT y Meshtastic y su funcionamiento y el guardado de datos en formato json.

Anexos

Enlace de github por si se borra el hipervínculo:

https://github.com/AlexGG327/Practica1_POO

Librerías a instalar para que funcione el programa.

```
pip install meshtastic

pip install paho-mqtt

pip install cryptography
```

o bien usar:

```
pip install -r requirements.txt
```