Documentación Técnica del Sistema IoT con MQTT y MongoDB

Integrantes:

Alexander Lozada Caviedes

David Felipe Rojas Bravo

1. Descripción General

Este sistema IoT está diseñado para gestionar datos de sensores ambientales (temperatura y humedad) dentro de un entorno de invernadero. La arquitectura incluye la recepción de datos mediante MQTT, almacenamiento en bases de datos MongoDB (local y en la nube), reenvío de datos a un servidor Flask y a un broker EMQX. Está compuesto por dos módulos principales: `mongo_utils.py` para operaciones de base de datos, y `mqtt.py` para procesamiento de mensajes MQTT. Esta separación modular permite que cada parte sea extensible y mantenible de forma independiente, y que el sistema escale fácilmente en función de nuevas necesidades o sensores adicionales.

2. Tecnologías y Librerías Utilizadas

Este sistema se apoya en varias bibliotecas clave del ecosistema Python, elegidas por su madurez, compatibilidad y enfoque en tareas específicas del stack IoT:

- **paho-mqtt**: Esta biblioteca permite la conexión eficiente y en tiempo real con brokers MQTT. Su inclusión es esencial para lograr la comunicación asincrónica entre sensores y el sistema central.
- **requests**: Elegida para realizar peticiones HTTP con facilidad, permite al sistema interactuar con servicios REST, como una API Flask, enviando datos de forma estructurada y segura.
- **python-dotenv**: Ofrece una forma práctica de cargar variables de entorno desde un archivo `.env`, lo que promueve buenas prácticas en la gestión de configuraciones sensibles y portabilidad entre entornos.
- **pymongo**: Esta es la biblioteca oficial de MongoDB para Python. Proporciona una API robusta y bien documentada para la manipulación de documentos, colecciones y bases de datos.
- **ssl**: Utilizada para establecer conexiones seguras (TLS/SSL) con el broker EMQX. Es crucial en sistemas productivos donde la transmisión segura de datos es obligatoria.
- **threading** y **time**: Estas bibliotecas permiten la ejecución periódica de tareas en segundo plano, como la sincronización de datos pendientes, sin interrumpir el flujo principal del sistema.

La elección de estas librerías responde al objetivo de construir un sistema robusto, seguro y capaz de operar en tiempo real bajo condiciones de red variables.

3. Flujo de Datos y Arquitectura del Sistema

El flujo de datos sigue un patrón bien definido que garantiza resiliencia y modularidad. El proceso comienza con la publicación de un mensaje MQTT por parte de un dispositivo sensor.

Un cliente MQTT local, configurado en `mqtt.py`, se conecta al broker definido en las variables de entorno y se suscribe al tópico especificado. Cuando un mensaje es recibido, se ejecuta el callback `on_local_message`, que redirige el procesamiento a la función `procesar_payload()`.

Allí, el mensaje se interpreta dependiendo del contenido y del tópico. Si el mensaje proviene de una cámara o contiene una imagen, este se reenvía directamente al servidor Flask, pensado como servicio de procesamiento o almacenamiento multimedia. Si el mensaje incluye datos numéricos como temperatura o humedad, se encapsula en un diccionario que contiene los campos `zona`, `tipo` y `valor`, y se agrega un campo `timestamp` para facilitar la trazabilidad y sincronización posterior.

Este objeto estructurado se almacena inicialmente en la base de datos local mediante `guardar_dato()`, utilizando MongoDB. Luego, si la conexión a la base de datos en la nube está disponible, el mismo documento (sin el campo `_id`) es replicado allí. Si la conexión remota falla, el sistema no se detiene: un hilo secundario, iniciado mediante `sincronizar_periodicamente()`, reintenta periódicamente la conexión y sincroniza los datos pendientes cuando sea posible.

Esta arquitectura desacoplada está diseñada para ambientes con conectividad intermitente. Al priorizar el almacenamiento local y permitir la sincronización asíncrona con la nube, el sistema asegura que los datos no se pierdan, incluso si la red está temporalmente fuera de servicio. Esta capacidad lo hace ideal para entornos rurales, industriales o de investigación con infraestructura limitada.

4. Módulo `mongo_utils.py`

Este módulo se encarga de toda la lógica relacionada con la persistencia de datos. Al inicio, carga las configuraciones de conexión a través de las variables de entorno `MONGO_URL_LOCAL` y `MONGO_URL_NUBE`, y establece las conexiones con ambas bases de datos MongoDB. Si la conexión a la nube falla, el sistema se adapta y continúa funcionando con la base local.

La función `guardar_dato(payload)` es crítica para la integridad de datos. Agrega un campo `timestamp` en formato ISO 8601, el cual actúa como clave lógica para evitar duplicación y facilitar sincronización. Luego guarda el dato localmente. Si la base de datos en la nube está operativa, el sistema realiza una copia del documento eliminando el campo `_id`, ya que este es generado automáticamente en cada instancia y causaría un conflicto si se intentara insertar duplicado.

La función `sincronizar_datos()` realiza una lectura completa de los documentos locales y verifica, usando el `timestamp`, si cada uno ya existe en la base de datos en la nube. Aquellos que no han sido sincronizados son insertados remotamente. Este mecanismo de recuperación asegura la redundancia de los datos, incluso tras cortes de red prolongados.

Finalmente, `sincronizar_periodicamente(interval)` ejecuta este proceso en segundo plano con un intervalo configurable, utilizando hilos para no afectar el rendimiento general del sistema. Este patrón garantiza que la sincronización no bloquee el flujo principal de recolección y procesamiento de datos.

5. Módulo `mqtt.py`

El módulo `mqtt.py` gestiona la lógica de comunicación con el broker MQTT y la distribución del mensaje recibido. Inicia un cliente MQTT con las credenciales y parámetros definidos en las variables de entorno, conectándose al broker local y suscribiéndose al tópico configurado.

La función `on_local_message()` actúa como callback al recibir un mensaje MQTT. Esta invoca a `procesar_payload()`, que realiza la decodificación del mensaje (generalmente en formato JSON) y analiza su contenido. Si el mensaje contiene una imagen, este se reenvía a un servidor Flask utilizando `enviar_a_flask()` mediante una petición HTTP POST. Este reenvío se encapsula en un bloque `try-except` para manejar errores sin detener el flujo principal.

En el caso de lecturas de sensores, se crean objetos JSON con los campos `zona`, `tipo` y `valor`, que se almacenan con `guardar_dato()` y se reenvían al broker EMQX mediante `enviar_a_emqx()`. Esta función establece una conexión segura (TLS) usando certificados y credenciales, y publica el mensaje en el tópico configurado.

Este módulo es responsable de coordinar el flujo de entrada y salida de datos en tiempo real, además de garantizar una distribución efectiva y segura hacia múltiples consumidores del ecosistema IoT.

6. Variables de Entorno (.env)

El uso de variables de entorno permite configurar fácilmente el sistema para distintos entornos de desarrollo, pruebas o producción, sin necesidad de modificar el código fuente. Cada parámetro afecta directamente la forma en que los módulos interactúan entre sí y con servicios externos.

- **MQTT Local**:
- `LOCAL_MQTT_HOST`: Dirección IP o nombre de host del broker MQTT local (por ejemplo, `localhost`).
- `LOCAL_MQTT_PORT`: Puerto de escucha del broker local, comúnmente `1883`. Es el punto de entrada para los sensores.
- `LOCAL_TOPIC`: Tópico específico donde los sensores publican sus mensajes.
- **EMQX**:
- `EMQX_BROKER`: Dirección del broker EMQX, que puede ser público o en la nube.
- `EMQX_PORT`: Puerto habilitado para conexiones seguras TLS (comúnmente `8883`).
- `EMQX_TOPIC_PREFIX`: Tópico raíz bajo el cual se publicarán los datos salientes.
- `EMQX_USERNAME`, `EMQX_PASSWORD`: Credenciales que permiten autenticar al cliente.
- `EMQX_CLIENT_ID`: ID único que identifica a esta instancia del cliente MQTT ante el broker.
- `EMQX_CA_CERT`: Ruta al certificado de la autoridad certificadora que valida la conexión segura.
- **MongoDB**:
- `MONGO_URL_LOCAL`: URI de la base de datos local, utilizada como almacenamiento principal ante fallas de red.
- `MONGO_URL_NUBE`: URI de la base de datos remota, ideal para redundancia y visualización externa.

- `MONGO_DB`: Nombre lógico de la base de datos (por defecto, `hidroponico`).
- `MONGO_COLLECTION`: Colección donde se almacenan los documentos (por defecto, `data_sensors`).
- **Flask API**:
- `FLASK_ENDPOINT`: URL donde el servidor Flask está escuchando peticiones POST con datos (ej. `http://localhost:5000/registro`).

7. Dependencias (requirements.txt)

Para garantizar la ejecución del sistema sin conflictos de dependencias, se proporciona un archivo `requirements.txt` con los siguientes paquetes:

```
paho-mqtt

requests

python-dotenv

pymongo
```

Para instalar los paquetes correctamente, ejecutar:

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

Esto asegura que todas las librerías requeridas estén disponibles en el entorno.

8. Consideraciones Finales

Este sistema está preparado para ambientes intermitentes o sin conectividad permanente con la nube, ya que prioriza la persistencia local y sincroniza cuando es posible. Su arquitectura modular facilita la integración con nuevos servicios o sensores, y su diseño desacoplado garantiza confiabilidad y tolerancia a fallos. El uso extensivo de variables de entorno refuerza su portabilidad entre entornos de desarrollo y producción, mientras que el procesamiento asíncrono y la comunicación segura lo convierten en una solución robusta para aplicaciones industriales o agrícolas sensibles.