ML4IoT-HW03

Hojjat Miryavifard s334026
Ali Behlooei Dolatsaraei s334033
Sadjad Zamani s331775

1. MQTT vs REST Protocol Selection

MQTT is a more suitable communication protocol than REST for this temperature and humidity monitoring application for several key reasons:

1.1. Message Size Efficiency

MQTT utilizes a lightweight binary protocol with minimal headers and a publish/subscribe architecture. In contrast, REST requires larger HTTP headers and complete request/response cycles. For frequent sensor readings, MQTT's smaller message footprint significantly reduces network overhead and bandwidth consumption.

1.2. Connection Handling

MQTT maintains a persistent TCP connection between the client and broker, eliminating the need to establish new connections for each data transmission. REST, being stateless, requires a new TCP handshake for every HTTP request. With sensor readings occurring every 2 seconds, MQTT's persistent connections provide better efficiency and lower latency.

1.3. Network Reliability

The MQTT protocol includes built-in mechanisms for handling unreliable network conditions, including message QoS levels and session persistence. These features are particularly valuable for IoT devices that may experience intermittent connectivity. REST lacks native support for such reliability features, requiring additional application-level implementations.

1.4. Power Efficiency

MQTT's lightweight nature and persistent connections result in lower power consumption compared to REST's connection overhead. This is crucial for IoT devices that may operate on battery power or have limited energy resources.

1.5. Scalability

MQTT's publish/subscribe model enables efficient one-to-many communication patterns in an asynchronous manner. Multiple subscribers can receive sensor data without additional overhead on the Raspberry Pi. REST's request/response communication would require separate requests for each consumer of the sensor data.

2. REST API HTTP Method Selection

For the `/data/{mac_address}` endpoint that retrieves historical temperature and humidity data, GET was selected as the most appropriate HTTP method after careful analysis of RESTful design principles. This choice reflects both theoretical best practices and practical implementation needs.

GET is designed for retrieving data without modifying the server's state, making it ideally suited for our historical data access endpoint. When clients request temperature and humidity readings for a specific date range, they retrieve the data and the content remains unchanged. The operation maintains two crucial properties: it is *safe*, meaning it does not alter server data.

The endpoint's structure requires start and end dates to define the query period, For example, a request might look like:

```
/data/0xf61e0bfe09?start_date=2023-12-21&
    end_date=2023-12-22
```

Other HTTP methods are not suitable for this use case. POST is designed for resource creation, which does not match our read-only operation. PUT and PATCH are meant for updating existing resources, but our endpoint never modifies historical data. DELETE would be inappropriate as we are maintaining, not removing, historical records.

Using GET improves performance with response caching, simplifies testing and debugging, and aids maintenance through easy logging. It has a user-friendly interface for accessing sensor data.

In contrast for the /sensors/ endpoint, we used a **POST** method because it is called when we want to add Raspberry PI sensor node.