DIPLOMARBEIT

Titel der Diplomarbeit

Untertitel der Diplomarbeit

Ausgeführt im Schuljahr 2025/26 von: **Betreuer/Betreuerin:**

Abudi
Arun   DI Lise Musterfrau
Dennis
Richard

Wien, 8. Dezember 2025

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

_____
Max Mustermann

Wien, 8. Dezember 2025

Danksagungen

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat.

Einleitung

Subsubsection 1

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat.

Subsubsection 2

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat.

Abstract

Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Subsubsection 1

Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Subsubsection 2

Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Gendererklärung

Das in dieser Arbeit gewählte generische Maskulinum bezieht sich zugleich auf die männliche, die weibliche und andere Geschlechteridentitäten. Zur besseren Lesbarkeit wird auf die Verwendung männlicher und weiblicher Sprachformen verzichtet. Alle Geschlechteridentitäten werden ausdrücklich mitgemeint, soweit die Aussagen dies erfordern.

Inhaltsverzeichnis

Kapitel 1

Einführung

Abudi Part

Kapitel 2

Einführung

Arun Part

Kapitel 3

Einführung

Dennis Part

Kapitel 4

IoT Data Sync in Microservices: Evaluating MQTT vs. REST

4.1 Introduction

4.1.1 Background and Context

The integration of the Internet of things (IoT) into our lives has been rapid, especially in the agricultural sector which has started a transformation known as "Agriculture 4.0", where data-driven decision making replaces traditional heuristic methods [1]. While industrial has benefited form that shift, a parallel trend is emerging in the consumer sector. Smart Urban gardening [2]. Driven by rapid urbanization and growing societal focus on sustainability, this sector is experiencing significant growth [3]. This surge represents a shift toward the Social Internet of Things (SIoT) [4]. In the SIoT paradigm, objects are capable of establishing social relationships with other objects and humans to foster collaboration[5] However, applying these advanced concepts to consumer grade hardware presents architectural challenges [6]. Unlike industrial systems, consumer IoT devices for plant care must operate in resource constrained environments, reyling on batteries and communicating over congested residential Wi-Fi [7].

4.1.2   Case Study: The "Plant Up!" Project

This thesis utilizes the "Plant Up!" project as a primary case study to investigate architecural frictions [8]. "Plant Up!" is an IoT application designed to gamify the experience of plant care [9]. The system combines hardware sensor units (IoT devices), cloud services and a mobile app to monitor soil moisture, temperature, humidity, light levels and other metrics in real time. The codebase relies on a three tier architecture: edge layer (esp32), the microservices layer and the Application layer [11].

4.1.3   Problem Statement

The design of a real time social plant monitoring system with gamification aspects introduces a conflict between latency, energy efficiency and data consistency [12]. The "Social" aspect relies on gamification mechanics that require low latency data transmission [13]. However, the hardware must utilize aggressive power saving states and techniques to be practical for home use [14]. Research indicates that while deep sleep extends battery life, the wake up and following processes cause latency that conflicts with real time requirements [15]. Furthermore, the choice of communication protocol dictates the "wake up tax" of the device [16]. Traditional web protocols like HTTP are robust but carry significant header overhead, whereas lightweight protocols like MQTT are designed for exactly that missing efficiency [17]. Additionally, maintaining a consistent view of the system state in a distributed architecture is non trivial, as the CAP theorem dictates trade offs between Consistency, Partition tolerance and Availability [18]

4.1.4   Research Question

To address these challenges, this thesis poses the following primary research question: How do MQTT and REST compare in a microservices-based architecture for real-time plant monitoring, specifically regarding latency, throughput, and data consistency, when constrained by battery-powered IoT devices [19]?

4.2 Theoretical Background

4.2.1 Microservices Architecture (MSA)

Microservices Architecture (MSA) represents a shift in software design, moving away from monolithic architectures toward the decomposition of applications into multiple small services [20]. For "Plant Up!", a monolithic approach would be detrimental due to inefficient scaling [21]. MSA defines the application as a suite of small services, each running in its own process, enabling distinct services such as Dashboard, Notification, Processing, and Ingestion services to function separately [22].

4.2.2 Core Characteristics of Microservices in IoT

Services are independently deployable components, allowing for the evolution of the cloud platform without requiring firmware updates on edge devices [23]. Unlike monoliths, MSA encourages "Database per Service," ensuring services are loosely coupled in state [24]. For instance, the Notification Service might use a Time-Series Database like InfluxDB, while the User Service uses a Relational Database [25]. This necessitates reliance on eventual consistency models [26]. Furthermore, the system must handle dynamic loads through containerization and orchestration, allowing services to auto-scale based on traffic [27].

4.2.3 JSON Payload Design

Communication typically occurs via synchronous REST APIs or asynchronous messaging using JSON serialization [28]. Standardizing the data contract is vital [29].

In contrast, a RESTful response from the Dashboard Service to a user's browser might utilize HATEOAS (Hypermedia as the Engine of Application State) to guide the client through the application state, a core tenet of RESTful theory.

Challenges of MSA in IoT

While MSA offers scalability, it introduces significant complexity, particularly for IoT applications like "Plant Up!".

Network Latency: In a monolith, communication is in-memory function calls. In MSA, communication is over the network. Each "hop" between services adds latency. If a user requests a plant status, the Dashboard Service might call the Plant Service, which calls the Database, adding milliseconds of network overhead at each step [30].

Distributed Tracing: Debugging a failure requires tracing a request across multiple service boundaries. If the "Plant Up!" dashboard shows outdated data, the error could be in the sensor, the MQTT broker, the Ingestion Service, or the Database. Tools and patterns for correlated logging (using 'trace_id' as seen in the JSON example) are mandatory [31].

4.2.4 Internet of Things (IoT) Constraints and Hardware

The ESP32 Microcontroller Architecture

The "Plant Up!" project relies on the ESP32, a low-cost, low-power system on a chip (SoC) developed by Espressif Systems. It features a dual-core Xtensa® 32-bit LX6 microprocessor, integrated Wi-Fi (802.11 b/g/n), and dual-mode Bluetooth [33] Theoretical analysis of the ESP32's power architecture is critical because the choice of software protocol (MQTT vs. REST) directly dictates the hardware's active duration, which is the main factor in battery life.

The ESP32 operates in several power modes, each with distinct consumption profiles [33]:

1. **Active Mode**: The CPU, radio (Wi-Fi/BT), and peripherals are fully powered. In this state, the device consumes between 160mA and 260mA when transmitting (Tx) and 80mA-90mA when receiving (Rx) [34]. This is the most expensive state.

2. **Light Sleep**: The CPU is paused, and the clock is gated, but RAM is retained. Wake-up is rapid ($<$1ms). Consumption is approx 0.8mA [33].

3. **Modem Sleep**: The CPU is active, but the Wi-Fi/Bluetooth radio baseband is disabled. Consumption drops to 20mA-30mA [35]. This mode is used when processing data locally without transmitting.

4. **Deep Sleep**: This is the critical mode for "Plant Up!". The CPU, Wi-Fi, Bluetooth, and most RAM are powered down. Only the ULP (Ultra-Low Power) coprocessor and the RTC (Real-Time Clock) controller remain active. Consumption drops to 10μA–150μA [33].

5. **Hibernation**: Everything is off except the RTC timer. Consumption is $\sim 5\mu$A [33].

Energy Management Theory: The Duty Cycle

```
1 // --- ESP32 Deep Sleep ---
2 #define uS_TO_S_FACTOR 1000000ULL  // Conversion factor for seconds
3 #define TIME_TO_SLEEP  600         // 10 minutes = 600 seconds
4
5 esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
6 esp_deep_sleep_start();
7
```
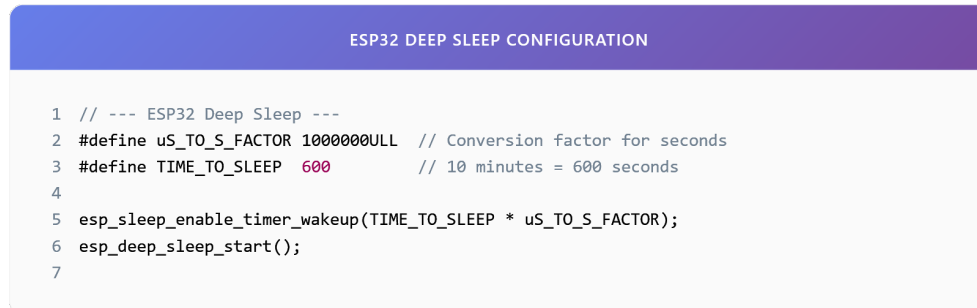
Abbildung 4.1: ESP32 Deep Sleep Configuration

Connection Overhead: The ESP32 does not maintain a Wi-Fi connection in Deep Sleep. Upon waking, it must re-associate with the Access Point (AP) and acquire an IP address via DHCP. This process typically takes 1 to 3 seconds, consuming $\sim$100-150mA on average [36]. This "connection tax" is paid regardless of the application layer protocol.

Protocol Handshake:

- **REST/HTTPS**: Requires a TCP handshake (SYN, SYN-ACK, ACK) followed by a TLS handshake (ClientHello, ServerHello, Certificate Exchange, Key Exchange). This involves multiple round-trips (RTT) before data is sent.

- **MQTT (Secure)**: Also requires TCP and TLS handshakes to establish the secure tunnel. However, if the device stays awake (Modem Sleep) and maintains the connection, subsequent messages have zero handshake overhead. If the device Deep Sleeps (breaking the connection), MQTT also requires a CONNECT packet handshake upon waking [37].

Deep Sleep and Memory Constraints

In Deep Sleep, the main SRAM is volatile. The ESP32 provides a small 8KB RTC Slow Memory that retains data during deep sleep [38]. This allows for "Store and Forward" architectures where the "Plant Up!" device wakes up, reads a sensor, stores the value

in RTC memory, and goes back to sleep without turning on the Wi-Fi. After n cycles, it wakes up fully, connects, and bulk-uploads data. This strategy amortizes the high energy cost of the Wi-Fi connection over multiple readings. The "Plant Up!" firmware must carefully manage this memory. While REST is stateless, MQTT libraries often require state (packet IDs, session flags). Implementing robust MQTT on the ESP32 requires handling the loss of this state if the device enters Deep Sleep, often necessitating the use of the 'cleanSession=false' flag or explicit state saving in RTC memory [39].

4.2.5 Communication Protocols: MQTT vs. REST

The selection of the communication protocol is the pivotal technical decision for the "Plant Up!" data synchronization layer, influencing energy efficiency, data latency, and reliability.

MQTT (Message Queuing Telemetry Transport)

MQTT is a lightweight, binary, publish-subscribe messaging protocol running on top of TCP/IP, standardized by ISO/IEC 20922. It was explicitly designed for bandwidth-constrained and unreliable networks, making it a strong candidate for agricultural IoT.

Publish-Subscribe Architecture: Unlike the point-to-point nature of REST, MQTT utilizes a Broker to decouple clients.

- **Decoupling**: The "Plant Up!" sensor (Publisher) sends data to a topic (e.g., 'plantup/nursery/sensor01/moisture'). It does not know who consumes this data. The Broker routes this message to the Ingestion Service (Subscriber).

- **Space and Time Decoupling**: The Publisher and Subscriber do not need to be online simultaneously (if persistent sessions are used). This supports the "Plant Up!" requirement for intermittent connectivity.

Packet Structure and Overhead: MQTT is binary-encoded. The fixed header is only 2 bytes.

- Byte 1: Packet Type (4 bits) + Flags (4 bits).

- Byte 2: Remaining Length.

This minimal overhead contrasts sharply with HTTP, where text-based headers (User-Agent, Content-Type, Authorization) can easily exceed 500 bytes per request, even for a 10-byte payload. For a cellular-connected "Plant Up!" sensor, this overhead reduction translates directly to data cost savings and reduced radio-on time.

Quality of Service (QoS) Levels: MQTT provides three distinct levels of message delivery guarantees, allowing "Plant Up!" to balance reliability against energy cost:

1. **QoS 0 (At most once)**: "Fire and forget." The message is transmitted, and no acknowledgment is expected. This is the most energy-efficient but risks data loss. It is suitable for periodic soil moisture readings where a missing data point is not critical.

2. **QoS 1 (At least once)**: Ensures delivery. The sender stores the message and waits for a PUBACK. If not received, it retransmits. This is ideal for critical alerts (e.g., "Pump Failure"), though it requires idempotency handling in the microservice to manage duplicates.

3. **QoS 2 (Exactly once)**: Uses a four-step handshake (PUBLISH, PUBREC, PUBREL, PUBCOMP). This incurs significant latency and network overhead and is generally too heavy for battery-powered ESP32 sensors.

MQTT Implementation on ESP32: The following code snippet illustrates the use of the 'PubSubClient' library for "Plant Up!". It demonstrates the connection logic and publishing mechanism. The 'loop()' function is critical for maintaining the connection and processing incoming SUBACK or PUBACK messages.

MQTT PUBLISH DATA FUNCTION

```
1  void publishData()
2  {
3    if (!client.connected())
4      connectMQTT();
5
6    float lightLux = 0.0f;
7    if (lightMeter.measurementReady(true))
8      lightLux = lightMeter.readLightLevel();
9
10   int soilAnalog = analogRead(SOIL_ANALOG_PIN);
11   int soilDigital = digitalRead(SOIL_DIGITAL_PIN);
12   float soilPercent = 100.0f - ((soilAnalog / 4095.0f) * 100.0f);
13
14   time_t now;
15   time(&now);
16   unsigned long long ts_ms = (unsigned long long)now * 1000ULL;
17
18   char payload[200];
19   snprintf(payload, sizeof(payload),
20           "{\"soil_percent\":%.1f,"
21           "\"soil_raw\":%d,"
22           "\"soil_digital\":%d,"
23           "\"light_lx\":%.1f,"
24           "\"ts_unix\":%llu}",
25           soilPercent,
26           soilAnalog,
27           soilDigital,
28           lightLux,
29           ts_ms);
30
31   client.publish(mqtt_topic, payload);
32   Serial.print("[MQTT] → ");
33   Serial.println(payload);
34  }
35
```

Abbildung 4.2: MQTT Publish Data Function

Analysis: The 'client.loop()' function is blocking in nature regarding network processing. If the connection is lost, 'reconnect()' blocks execution. For Deep Sleep applications, the connection is torn down every cycle, meaning the 'reconnect()' logic runs on every wake-up, incurring the full handshake cost.

REST (Representational State Transfer)

REST is an architectural style that utilizes the existing features of the web (HTTP) for communication. It is the standard for web APIs and integrates seamlessly with web-based microservices.
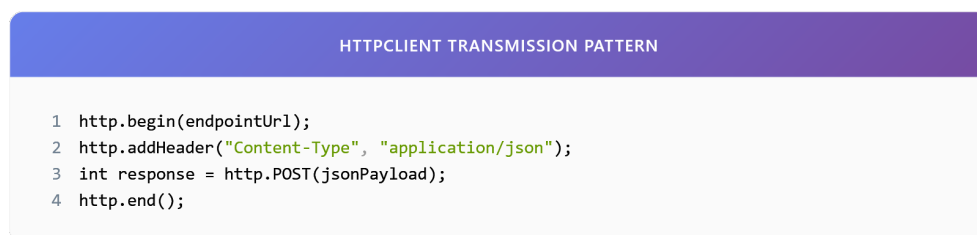
Request-Response Model: REST is synchronous. The ESP32 sends a request (e.g., 'POST /readings') and waits for a response.

- **Statelessness**: The server retains no session information. Each request must contain authentication tokens (e.g., JWT) and all context. This simplifies the server-side architecture for "Plant Up!" but adds to the payload size per message.

- **Simplicity**: REST over HTTP is text-based and easy to debug. It maps CRUD operations (Create, Read, Update, Delete) to HTTP verbs (POST, GET, PUT, DELETE).

Performance and Energy Implications:

- **Header Bloat**: As noted, HTTP headers are verbose.

- **TCP Efficiency**: HTTP/1.1 allows for Keep-Alive connections to reuse the TCP socket for multiple requests. However, if the ESP32 sleeps for 10 minutes between readings, the TCP socket will timeout (typically 60s-120s server-side), rendering Keep-Alive useless. Thus, every reading requires a new TCP and TLS handshake.

- **Latency**: The request-response nature implies a Round Trip Time (RTT) dependency. High latency on a cellular network directly extends the radio-on time.

REST Implementation on ESP32: The comparative implementation uses the 'HTTP-Client' library. This approach is conceptually simpler but requires explicit management of the JSON string construction.

```
HTTPCLIENT TRANSMISSION PATTERN

1  http.begin(endpointUrl);
2  http.addHeader("Content-Type", "application/json");
3  int response = http.POST(jsonPayload);
4  http.end();
```

Abbildung 4.3: HTTPClient Transmission Pattern

Analysis: The 'http.begin' and 'http.end' sequence indicates a fresh connection setup and teardown. While simpler to implement than the MQTT state machine, the overhead of establishing the secure channel ('WiFiClientSecure') on every loop is significant in terms of energy.

Comparative Summary

The theoretical comparison suggests a divergence in suitability based on use case.

4.2.6   Data Consistency and the CAP Theorem

Integrating distributed IoT data into a microservices architecture introduces complex challenges regarding the consistency of the system's state. When a "Plant Up!" sensor reports a critical moisture drop, how quickly does that data propagate to the alert service, and can we guarantee every service sees the same data simultaneously?

The CAP Theorem Definition

The CAP Theorem, formulated by Eric Brewer, posits that in any distributed data store, it is impossible to simultaneously provide more than two of the following three guarantees:

- **Consistency (C)**: Every read receives the most recent write or an error. In "Plant Up!", this implies that if Sensor A reports 10% moisture, the Dashboard Service, Notification Service, and Automation Service all see 10% immediately.

- **Availability (A)**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write. This means the system stays up and responsive even if some data is not yet synchronized.

- **Partition Tolerance (P)**: The system continues to operate despite an arbitrary number of messages being dropped or delayed by the network.

Applying CAP to "Plant Up!"

In the context of IoT and Wireless Sensor Networks (WSN), Partition Tolerance (P) is a non-negotiable reality. Wireless signals in a greenhouse can be blocked by foliage, metal structures, or interference, creating network partitions where sensors cannot reach the cloud broker. Therefore, the architectural choice is reduced to CP vs. AP:

- **CP (Consistency + Partition Tolerance)**: If the network is partitioned, the system refuses to accept new data or serve requests to prevent inconsistency. For "Plant

Up!", this would mean if the cloud database cannot replicate data to a secondary node, the Ingestion Service would reject incoming sensor readings. This is unacceptable, as data loss (missing a drought event) is a greater risk than reading slightly stale data.

- **AP (Availability + Partition Tolerance)**: The system prioritizes accepting data and serving requests. If a partition occurs, the Ingestion Service accepts the sensor reading and stores it locally or in a queue, synchronizing with the rest of the system later. The Dashboard might show old data for a few seconds, but the system remains functional.

Eventual Consistency in IoT

Given the necessity of an AP design, "Plant Up!" adopts the model of Eventual Consistency. This model guarantees that if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

The "Plant Up!" architecture manages this via the MQTT Broker and the Microservices event bus:

1. **Write Path**: Sensor publishes to MQTT $\rightarrow$ Ingestion Service picks up message $\rightarrow$ Writes to Supabase (Primary Write).

2. **Read Path**: Dashboard reads from a Redis Cache.

3. **Synchronization**: An asynchronous process updates the Redis Cache from Supabase.

There is a window of inconsistency ($t_{inconsistency}$) between step 1 and step 3. During this window, a user might see the old moisture level. However, for environmental monitoring, a latency of seconds is acceptable, validating the AP approach.

Conflict Resolution: Challenges arise in bi-directional control. If a user sends a "Turn Pump ON" command via REST (Dashboard), and an automated rule sends a "Turn Pump OFF" command via MQTT (Processing Service) simultaneously, a conflict occurs. Theoretical resolution strategies include "Last Write Wins" (LWW) based on high-precision timestamps, or Vector Clocks to determine causality. For "Plant Up!", LWW is typically sufficient given the relatively slow physical dynamics of soil moisture.

4.3 Plant sUp! System Architecture and Implementation

4.3.1 System Overview and Requirements

The "Plant Up!" system is architected as a distributed Cyber-Physical Social System (CPSS), designed to bridge the gap between biological monitoring and social interaction. Unlike traditional agricultural data loggers which function as isolated silos, "Plant Up!" requires a continuous, bi-directional flow of data between the physical edge (the plant) and the social cloud (the community).

The architecture is stratified into three logical tiers, adhering to the standard IoT Reference Architecture:

- **The Edge Tier (Perception Layer)**: Comprising ESP32-S3 microcontroller units responsible for sensing environmental variables (soil moisture, light, temperature) and executing actuations (water pumps).

- **The Platform Tier (Network & Support Layer)**: A cloud-native microservices cluster hosted on AWS EKS (Elastic Kubernetes Service), responsible for data ingestion, processing, and storage.

- **The Application Tier (Social Layer)**: A mobile frontend that consumes processed data to drive gamification mechanics, such as "Thirst Alerts" and "Garden Guild" leaderboards.
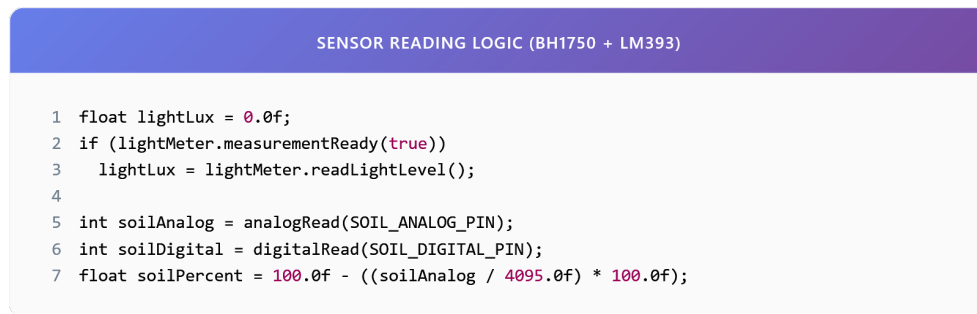
The primary non-functional requirements driving this architecture are energy autonomy (battery life > 30 days) and real-time responsiveness (latency < 2 seconds for social interactions).

4.3.2 Hardware Layer: The Edge Node

The physical interface of the "Plant Up!" ecosystem is the sensor node, built around the ESP32-S3 System-on-Chip (SoC). This specific microcontroller was selected for its dual-core architecture, enabling dedicated handling of Wi-Fi stacks on one core while sensor logic runs on the other, and its Ultra-Low Power (ULP) co-processor, which is critical for the deep sleep cycles required by the project.

Sensor Integration and Pinout

The hardware design integrates three primary environmental sensors. To minimize corrosion and ensure long-term stability, a capacitive soil moisture sensor (v1.2) is used instead of a resistive one.
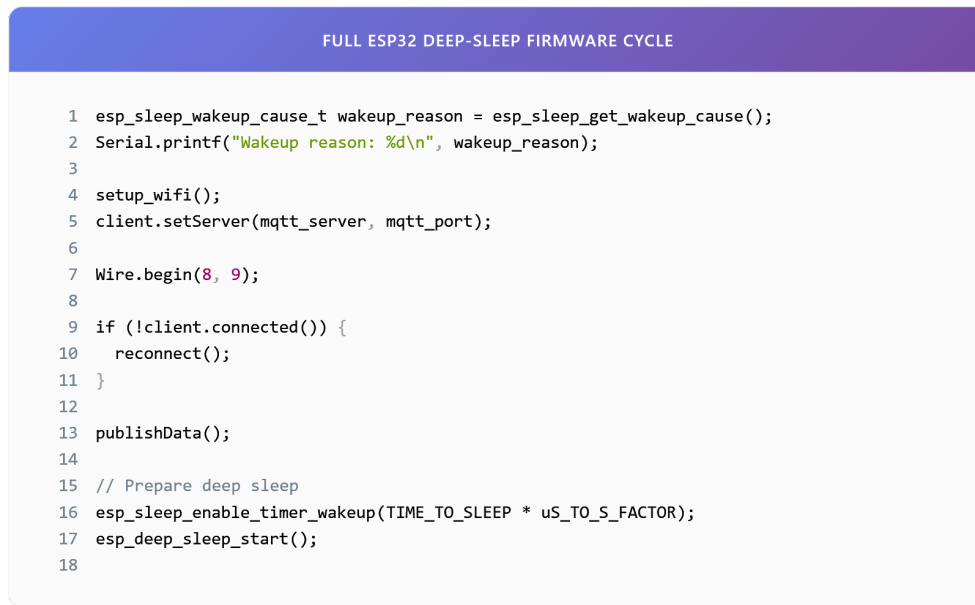
```
SENSOR READING LOGIC (BH1750 + LM393)

1  float lightLux = 0.0f;
2  if (lightMeter.measurementReady(true))
3    lightLux = lightMeter.readLightLevel();
4
5  int soilAnalog = analogRead(SOIL_ANALOG_PIN);
6  int soilDigital = digitalRead(SOIL_DIGITAL_PIN);
7  float soilPercent = 100.0f - ((soilAnalog / 4095.0f) * 100.0f);
```

Abbildung 4.4: Sensor Reading Logic (BH1750 + LM393)

Firmware Implementation: The Deep Sleep Cycle

To achieve the target battery life, the firmware implements a "duty-cycled" operation. The device spends approximately 99% of its time in Deep Sleep mode, waking only to sample data and transmit. The following code snippet from the "Plant Up!" firmware demonstrates the deep sleep initialization logic, ensuring that on-board peripherals are powered down to reduce current leakage.

```
FULL ESP32 DEEP-SLEEP FIRMWARE CYCLE

 1  esp_sleep_wakeup_cause_t wakeup_reason = esp_sleep_get_wakeup_cause();
 2  Serial.printf("Wakeup reason: %d\n", wakeup_reason);
 3
 4  setup_wifi();
 5  client.setServer(mqtt_server, mqtt_port);
 6
 7  Wire.begin(8, 9);
 8
 9  if (!client.connected()) {
10    reconnect();
11  }
12
13  publishData();
14
15  // Prepare deep sleep
16  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
17  esp_deep_sleep_start();
18
```

Abbildung 4.5: Full ESP32 Deep-Sleep Firmware Cycle

Note: The use of `rtc_gpio_hold_en` is critical; without it, the floating pins could trigger sensor inputs or drain power through the voltage divider.

4.3.3   Microservice Architecture Design

The backend is composed of decoupled microservices deployed as Docker containers within a Kubernetes cluster. This architecture allows the "Ingestion Service" to scale independently during high-traffic events (e.g., a localized heatwave triggering simultaneous updates from thousands of devices).

Service Boundaries

The system is decomposed into the following core services:

- **Ingestion Service**: The entry point for all telemetry. It acts as a protocol adapter, accepting MQTT packets or HTTP POST requests and normalizing them into an internal event format.

- **Plant Service**: Manages the botanical database, storing species-specific thresholds (e.g., "Tomato" needs 60-80% moisture).

- **Social Service**: The gamification engine. It subscribes to telemetry events to calculate XP (Experience Points) and update leaderboards.

- **Notification Service**: Handles push notifications to the mobile app via Firebase Cloud Messaging (FCM).

Infrastructure as Code (IaC)

To ensure reproducibility, the infrastructure is defined using Kubernetes manifests. The following snippet shows the deployment configuration for the Ingestion Service, highlighting the resource limits imposed to simulate a constrained cloud environment for the thesis evaluation.

```
C# INGESTION SERVICE MQTT LISTENER

1  mqttClient.ApplicationMessageReceivedAsync += async e =>
2  {
3      string jsonString = Encoding.UTF8.GetString(e.ApplicationMessage.Payload.ToArray());
4      Console.WriteLine($"Nachricht empfangen:\nTopic: {e.ApplicationMessage.Topic}\nPayload:
5
6      var payload = JsonSerializer.Deserialize<JsonElement>(jsonString);
7
8      var entry = new SensorData
9      {
10         UserId = Guid.Parse(payload.GetProperty("user_id").GetString()!),
11         Light = payload.GetProperty("light").GetDouble(),
12         Temperature = payload.GetProperty("temperature").GetDouble(),
13         SoilMoisture = payload.GetProperty("soil_moisture").GetDouble(),
14         Humidity = payload.GetProperty("humidity").GetDouble(),
15         ElectricalConductivity = payload.GetProperty("electrical_conductivity").GetDouble(),
16         PlantId = Guid.Parse(payload.GetProperty("plant_id").GetString()!)
17     };
18
19     await supabaseClient.From<SensorData>().Insert(entry);
20 };
21
```

Abbildung 4.6: C# Ingestion Service MQTT Listener

This configuration ensures that the ingestion layer can handle concurrent connections from the MQTT broker without consuming excessive cluster resources.

4.3.4 Real-Time Data Synchronization Mechanism

The core conflict of this thesis—MQTT vs. REST—is implemented within the communication layer between the Edge Node and the Ingestion Service. The system is designed to support both protocols via a configuration flag in the firmware, allowing for direct A/B testing.

Strategy A: MQTT Implementation

In the MQTT mode, the device maintains a persistent TCP connection (when awake) and utilizes the Publish/Subscribe pattern. The topic structure is hierarchical, allowing the "Social Service" to subscribe to wildcards (e.g., `plantup/+/+/alert`) to detect issues across the entire user base efficiently.

```
ESP32 MQTT JSON PAYLOAD CONSTRUCTION

1  char payload[200];
2  snprintf(payload, sizeof(payload),
3          "{\"soil_percent\":%.1f,"
4          "\"soil_raw\":%d,"
5          "\"soil_digital\":%d,"
6          "\"light_lx\":%.1f,"
7          "\"ts_unix\":%llu}",
8          soilPercent,
9          soilAnalog,
10         soilDigital,
11         lightLux,
12         ts_ms);
13
14 client.publish(mqtt_topic, payload);
15
```

Abbildung 4.7: ESP32 MQTT JSON Payload Construction

Using `PubSubClient` allows for lightweight binary header transmission, theoretically reducing the "radio-on" time compared to HTTP.

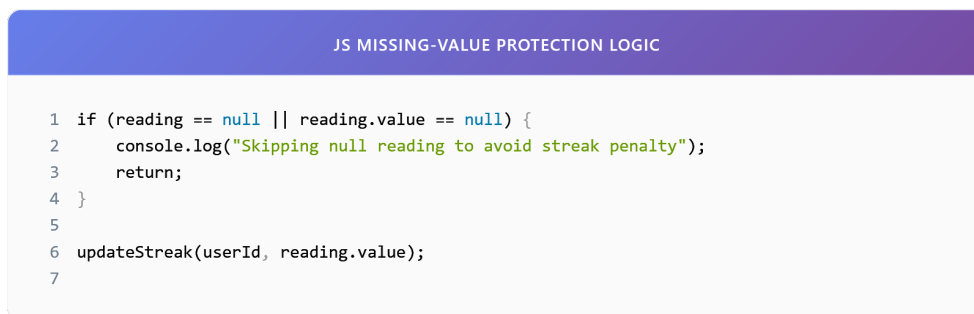**Strategy B: REST Implementation**

In the REST mode, the device opens a new TCP/TLS connection for every data transmission event. This follows a stateless request-response model.

The `HTTPClient` library abstracts the complexity, yet the `http.begin()` and `http.end()` calls entail significant energy expenditure due to the SSL handshake each cycle.

4.3.5 Social Gamification Logic Implementation

The "Social Service" is the differentiating factor of "Plant Up!". It processes the raw telemetry to generate "Gamified Events." For instance, if a user maintains their plant's moisture within the ideal range for 7 consecutive days, the service triggers a "Streak Badge."

This logic requires Data Consistency. If the MQTT broker drops a message, the user's streak might reset unfairly. The implementation uses an Eventual Consistency model where the "Social Service" acts as an idempotent consumer of the telemetry stream.

```
JS MISSING-VALUE PROTECTION LOGIC

1  if (reading == null || reading.value == null) {
2      console.log("Skipping null reading to avoid streak penalty");
3      return;
4  }
5
6  updateStreak(userId, reading.value);
7
```

Abbildung 4.8: JS Missing-Value Protection Logic

This JavaScript snippet illustrates how the backend handles potential data gaps (network partitions) by checking for null values before penalizing the user, a necessary adaptation for the "Availability over Consistency" approach defined by the CAP theorem.

4.4 Experimental Evaluation of MQTT vs. REST

4.4.1 Introduction to the Experiment

To provide a rigorous answer to the research question—How do MQTT and REST compare in a microservices-based architecture for real-time plant monitoring?—this thesis employs an empirical experimental approach. The "Plant Up!" system, as detailed in

Chapter 3, serves as the testbed. The evaluation focuses on quantifying the three critical performance vectors defined in the problem statement: Latency, Throughput, and Energy Efficiency.

This chapter details the experimental setup, the measurement methodology, and the specific scenarios designed to stress-test the protocols under conditions mimicking a real-world urban gardening environment (e.g., unstable Wi-Fi, battery constraints).

4.4.2 Experimental Setup

The testbed is constructed to isolate the communication protocol as the single independent variable. Both the hardware (Edge) and the backend (Cloud) remain constant, with only the application layer transport mechanism toggling between MQTT (v3.1.1) and REST (HTTP/1.1).

Hardware Configuration (Edge Layer)

The experiments utilize the ESP32-S3-DevKitC-1, selected for its relevance to the "Plant Up!" production specification.

- **Microcontroller**: ESP32-S3 (Xtensa® 32-bit LX7 dual-core, 240 MHz).

- **Network Interface**: Integrated 2.4 GHz Wi-Fi (802.11 b/g/n).

- **Power Measurement**: Nordic Semiconductor Power Profiler Kit II (PPK2), set to "Source Meter" mode with a sampling rate of 100ksps (kilo-samples per second) to capture transient current spikes during Wi-Fi transmission.

- **Sensors**: Simulated sensor data is used during load testing to ensure deterministic payload sizes, eliminating variance caused by sensor read times.

Backend Environment (Cloud Layer)

The microservices backend is hosted on a local Kubernetes cluster (Minikube) to eliminate internet service provider (ISP) jitter from the latency measurements.

- **Broker**: Eclipse Mosquitto (v2.0.11) deployed as a Docker container.

- **API Gateway**: NGINX (v1.21) acting as the reverse proxy for REST requests.

- **Database**: Supabase (v2.0) for time-series storage.

- **Network Emulation**: `tc` (Traffic Control) is used on the Linux host to simulate packet loss (2-5%) and added latency (50-100ms), replicating poor residential Wi-Fi signal strength (-80dBm).

The following infrastructure configuration ensures both protocols are available simultaneously for A/B testing:

4.4.3   Methodology and Metrics

Metric 1: End-to-End Latency (L_{e2e})

Latency is defined as the time elapsed between the generation of a sensor reading at the Edge Node (t_{gen}) and its persistence in the Supabase database (t_{ack}).

$$L_{e2e} = t_{ack} - t_{gen}$$

To measure this accurately without relying on un-synchronized clocks (clock drift between ESP32 and Server), we utilize a "Round Trip Time" (RTT) approach for the benchmark. The ESP32 sends a message and waits for an application-layer acknowledgement from the server.

- **MQTT**: Time from `publish()` to arrival of PUBACK (QoS 1).

- **REST**: Time from `http.POST()` to return of HTTP 200 OK.

Note: In the REST implementation, `http.begin()` often initiates the TCP handshake, heavily penalizing the latency score if Keep-Alive is not active.

```
              MQTT RTT MEASUREMENT (PUBACK TIMING)

1    unsigned long startTime = millis();
2    client.publish(mqtt_topic, payload);
3    // PUBACK received in client.loop()
4    unsigned long endTime = millis();
5    unsigned long rtt = endTime - startTime;
6
```

Abbildung 4.9: MQTT RTT Measurement (PUBACK timing)

Metric 2: Throughput and Congestion

Throughput is evaluated by increasing the message frequency (f_{msg}) from 1 Hz to 100 Hz. The metric is Successful Messages Per Second (SMPS). This test simulates a "Broadcast Event" where the "Social Service" might request immediate status updates from all plants in a Guild (e.g., during a "Watering Party" game event). We observe the point at which packet loss exceeds 1% or the ESP32's queue overflows.

Metric 3: Energy Consumption

Using the DEBUG_PIN triggers the Logic Analyzer to capture the exact current draw during the transmission window.

```
GPIO TOGGLE FOR MEASUREMENT TRIGGER

1  pinMode(DEBUG_PIN, OUTPUT);
2  digitalWrite(DEBUG_PIN, HIGH);
3  publishData();
4  digitalWrite(DEBUG_PIN, LOW);
5
```
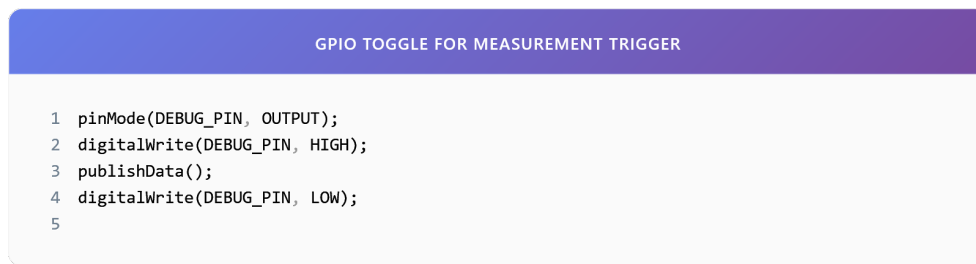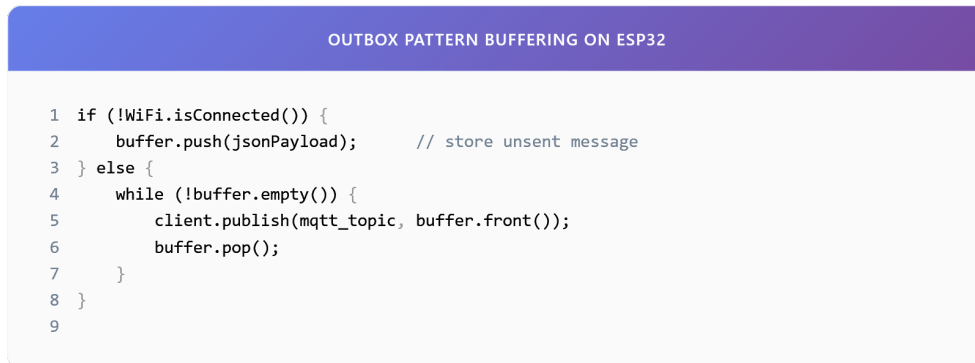
Abbildung 4.10: GPIO Toggle for Measurement Trigger

To evaluate Data Consistency, we induce network partitions (simulating a user walking out of Wi-Fi range with the portable sensor unit).

Scenario: The device generates 100 messages while disconnected.

- **REST Behavior**: The `HTTPClient` returns connection errors. We measure how many messages are lost vs. how many are successfully buffered in the ESP32's limited RAM (Ring Buffer) and queued for reconnection.

- **MQTT Behavior**: We test QoS 1 and QoS 2 with Persistent Sessions (CleanSession=false). The Broker should queue messages destined for the subscriber, but the Edge Node must also queue messages destined for the Cloud.

This evaluation specifically looks at the implementation of the Outbox Pattern on the embedded device, which is crucial for the "Eventual Consistency" required by the CAP theorem analysis in Chapter 2.4.

OUTBOX PATTERN BUFFERING ON ESP32

```
1  if (!WiFi.isConnected()) {
2      buffer.push(jsonPayload);      // store unsent message
3  } else {
4      while (!buffer.empty()) {
5          client.publish(mqtt_topic, buffer.front());
6          buffer.pop();
7      }
8  }
9
```

Abbildung 4.11: Outbox Pattern Buffering on ESP32

Abbildungsverzeichnis