📘 Projektdokumentation: MQTT-Dashboard

1. Einleitung

Ziel dieses Projekts war die Entwicklung eines webbasierten Dashboards zur Visualisierung von Temperaturwerten, die über das MQTT-Protokoll übertragen werden. Die Datenquelle sollte ein Raspberry Pi oder ein vergleichbares Gerät sein, welches regelmäßig Temperaturdaten an einen lokalen MQTT-Broker sendet.

Ein Python-Backend auf Basis von Flask empfängt diese Daten, stellt sie über eine API zur Verfügung und ermöglicht eine grafische Darstellung im Browser mit Hilfe von Chart.js. Darüber hinaus wurde ein System zur Statusanzeige und Einstellbarkeit von Parametern wie Grenzwerte, Update-Intervall und Werteanzahl implementiert.

2. Technische Umsetzung

2.1 Systemübersicht

Das gesamte System besteht aus:

- **Publisher**: Python-Skript, das simulierte Temperaturdaten über MQTT versendet.
- **MQTT-Broker**: Vermittelt die Kommunikation zwischen Publisher und Subscriber.
- **Dashboard/WebUI**: Flask-Webserver, der Daten empfängt und im Frontend mit Chart.js darstellt.

2.2 Backend (Flask + MQTT)

Das Backend nutzt Flask als HTTP-Server und paho-mqtt, um mit dem MQTT-Broker zu kommunizieren. Der relevante Codeausschnitt für den Subscriber:

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(BROKER, PORT, keepalive = 60)
client.loop_start()
```

Nach erfolgreicher Verbindung mit dem Broker abonniert der Client das Topic test/topic. Die Nachrichten werden im Callback on_message verarbeitet:

```
def on_message(client, userdata, msg):
        try:
                payload = msg.payload.decode()
                data = json.loads(payload)
                messages.append(data["value"])
        except json.JSONDecodeError:
                print("json fail")
```

Ein REST-Endpunkt liefert den letzten Wert:

```python
@app.route("/get_cur", methods = ["POST", "GET"])
def cur():
        global messages
        try:
                return jsonify({"val": messages[-1]})
        except:
                return jsonify({"val": "--"})
```

Eine zusätzliche Besonderheit: Die IP-Adresse wird automatisch erkannt und beim Start in das JavaScript eingetragen, damit fetch() funktioniert:

```python
import subprocess

v = subprocess.run(["hostname", "-I"], text = True, capture_output = True)
ip = v.stdout.split(" ")[0]
```

2.3 Frontend (HTML, JS, Chart.js)

Das Dashboard ist eine statische HTML-Seite mit JavaScript und CSS. Sie bietet:

- Live-Anzeige der Temperatur:

```javascript
let currentTemp = data.val;
curr_temp_div.innerText = `${currentTemp.toFixed(2)}`;
```

- Durchschnittsberechnung:

```javascript
let mean_e = document.getElementById("temp_mean");
let m = 0;
for (let i = 0; i<tempData.length; i++) {
    m += tempData[i];
}
m /= tempData.length;
mean_e.innerText = m.toFixed(2);
```

- Statusanzeige:

```
let status_elem = document.getElementById("status");
status_elem.innerHTML = "";
if (currentTemp >= upper_bound) {

    status_elem.className = "status_warm";
    status_elem.innerHTML = '<svg xmlns="http://www.w3.org/2000/svg" viewBox=
} else if (currentTemp < upper_bound && currentTemp > lower_bound) {

    status_elem.className = "status_good";
    status_elem.innerHTML = '<svg xmlns="http://www.w3.org/2000/svg" viewBox=
} else if (currentTemp <= lower_bound) {

    status_elem.className = "status_cold";
    status_elem.innerHTML = '<svg xmlns="http://www.w3.org/2000/svg" viewBox=
}
```

- Historisches Diagramm mit Chart.js:

```
let chart;
function createChart() {
    const ctx = document.getElementById('tempGraph').getContext('2d');
    chart = new Chart(ctx, {
        type: 'line', // Liniendiagramm
        data: {
            labels: timeLabels, // Zeitstempel
            datasets: [{
                label: 'Temperatur (°C)',
                data: tempData, // Temperaturdaten
                borderColor:'#1982c4', // Linienfarbe
                fill: true,
                tension: 0.1
            }]
        },
        options: {
            responsive: true,
            scales: {
                x: {
                    type: 'category', // Zeitstempel als Kategorien verwenden
                    position: 'bottom'
                },
                y: {
                    beginAtZero: false, // Y-Achse beginnt nicht bei 0
                    title: {
                        display: true,
                        text: 'Temperatur (°C)'
                    }
                }
            }
        }
    });
}
```

Eingaben über HTML-Inputs ermöglichen die Anpassung der Anzeigeparameter:

```
<div class = "block_title">Wertanzahl</div>
<input type="number" id = "max_val_input" onchange="set_max_dp()" value="10">
```

2.4 Datenfluss

1. **MQTT-Publisher** sendet JSON-Daten wie { "value": 42.5 }
2. **MQTT-Client** im Flask-Backend verarbeitet die Nachricht
3. **Flask API** stellt den aktuellen Wert über /get_cur bereit
4. **JavaScript-Fetch** holt regelmäßig neue Werte und aktualisiert die Anzeige

3. Herausforderungen

3.1 Fehlende Vorerfahrung

MQTT war für alle Projektteilnehmer Neuland. Besonders die Kombination mit Flask stellte eine Herausforderung dar, da hier parallele Prozesse und asynchrone Datenflüsse gemeistert werden mussten.

3.2 Zeitdruck

Das Projekt musste in kurzer Zeit realisiert werden. Einige Funktionen wie eine dauerhafte Datenspeicherung (z. B. in SQLite) wurden zunächst ausgelassen.

3.3 IP-Problem bei fetch()

Da fetch('http://127.0.0.1:9000/get_cur') nicht im Netzwerk funktioniert, wurde der Pfad bei jedem Serverstart automatisch angepasst – eine pragmatische, aber funktionierende Lösung für den lokalen Testbetrieb.

4. Fazit & Ausblick

4.1 Ergebnisse

Das MQTT-Dashboard funktioniert zuverlässig im lokalen Netzwerk. Es visualisiert Temperaturdaten in Echtzeit, berechnet Durchschnittswerte und signalisiert kritische Zustände visuell.

4.2 Erweiterungsideen

- Speicherung in einer Datenbank
- Anzeige vergangener Werte durch Scrollfunktion im Chart
- Push-Benachrichtigungen
- Authentifizierung für Zugriffsschutz

4.3 Persönliches Fazit

Die Arbeit am Projekt war technisch herausfordernd, aber äußerst lehrreich. Die Integration von Netzwerkkommunikation, Webentwicklung und Datenvisualisierung in einem realistischen Szenario vermittelte praxisnahes Wissen über IoT-Systeme. Besonders stolz sind wir auf das Zusammenspiel der Komponenten trotz der anfänglichen Schwierigkeiten.