Smart Home Control System mit MQTT, Sensorik und GUI

Name and Student IDs:

Florent Salihi (k12223924) - Teamleader

Stefan Djukic (k12203945)

Harun Imamovic (k12213168)

Arian Kycyku (k12220337)

Philipp Riener (k12206255)

GitHub-Repository: https://github.com/l3ntii/smarthome

Introduction:

Im Rahmen dieses Projekts wurde ein vernetztes eingebettetes System entwickelt, das verschiedene Sensoren zur Überwachung der Umgebungsbedingungen verwendet und in Abhängigkeit davon automatische Reaktionen durch Aktoren auslöst. Zusätzlich können alle relevanten Informationen und Steuerfunktionen über eine zentrale grafische Benutzeroberfläche (GUI) beobachtet und kontrolliert werden.

Das System besteht aus fünf Kernfunktionen:

1. **Lichtautomatisierung:**
   Mittels Photoresistor können wir eine Lichtmessung machen. Bei Unterschreiten eines definierten Schwellwerts wird automatisch eine LED eingeschaltet, um eine Lichtquelle zu simulieren. Zusätzlich kann die Lichtsteuerung manuell über die GUI ein- oder ausgeschaltet werden.

2. **Temperaturgeregelte Klimasteuerung:**
   Ein Temperatursensor misst die Umgebungstemperatur. Bei Temperaturen unter zum Beispiel 20 °C wird eine LED aktiviert, die eine Heizung simuliert. Bei Temperaturen über 25 °C wird ein kleiner Motor eingeschaltet, der einen Ventilator darstellt.

3. **Sicherheitsabschaltung des Ventilators:**
   Um eine Gefährdung durch den laufenden Ventilator zu vermeiden, wird ein Abstandssensor verwendet, der erkennt, ob sich eine Person oder ein Objekt zu nahe nähert. In diesem Fall wird der Ventilator automatisch abgeschaltet, unabhängig von der Temperatursteuerung.

4. **Abstandsbasiertes Alarmsystem:**
   Ein weiterer Abstandssensor überwacht einen Eingangsbereich. Wenn das Alarmsystem über die GUI aktiviert wurde und sich jemand dem Sensorbereich nähert, wird ein Alarm ausgelöst. Hier wird ein Buzzer angesteuert sowie wird im GUI eine Nachricht mit Timestamp angezeigt, wenn der Alarm ausgelöst hat.

5. **Zentrale grafische Benutzeroberfläche:**
   Die GUI dient als zentrale Steuer- und Visualisierungseinheit. Sie zeigt alle Sensorwerte im fünf Sekundentakt an und erlaubt manuelle Steuerung aller Funktionen. Die Kommunikation zwischen den Komponenten erfolgt mittels MQTT über definierte Topics.

Full Description of the System Components:

Das System besteht aus zwei logisch getrennten Softwarekomponenten, die auf zwei Raspberry Pis verteilt sind: einem **Sensor-RPi** und einem **Aktor-RPi**. Beide Komponenten sind vollständig in Python implementiert und nutzen MQTT für die Kommunikation.

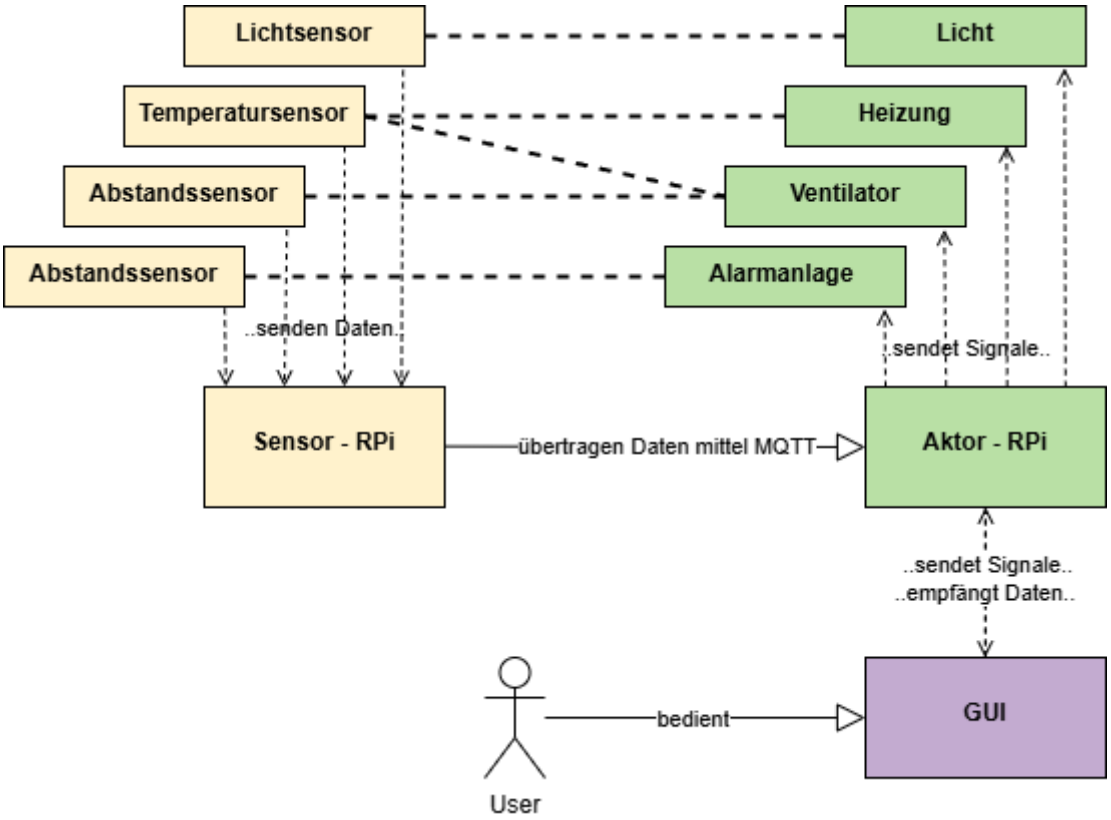

Abbildung 1: Systemarchitektur

Sensor-RPi (Verzeichnis: project/sensor-rpi/)

Der Sensor-RPi enthält die gesamte Logik zur Erfassung der Umgebungsdaten. Die wichtigsten Module sind:

- **main.py** – Startpunkt der Anwendung, initialisiert die Sensorobjekte und startet den MQTT-Client.
- **config.py** – Konfigurationsdatei für MQTT-Broker, MQTT-Themen und Messintervalle.
- **mqtt/client.py** – Verwaltet die MQTT-Verbindung und publiziert Sensordaten an den Broker.
- **sensors/** – Dieses Verzeichnis enthält die Implementierung der einzelnen Sensoren:
  - **temp.py** – liest die Temperatur über den HTS221-Sensor des Sense HAT aus.
  - **light.py** – misst die Umgebungshelligkeit mit einem Photoresistor über RC-Timing.
  - **safety.py** – verwendet den VL6180X (ToF-Sensor), um Personen bzw. Objekte in Lüfternähe zu erkennen.
  - **alarm.py** – nutzt den HC-SR04 (Ultraschall), um Bewegungen im überwachten Bereich zu detektieren.

Die Sensorwerte werden in regelmäßigen Intervallen gelesen und über definierte MQTT-Topics (z. B. sensors/temperature, sensors/light) an den Broker gesendet.

Aktor-RPi (Verzeichnis: project/aktor-rpi/)

Der Aktor-RPi verarbeitet sowohl eingehende Sensordaten als auch Benutzereingaben über eine grafische Oberfläche. Die Software ist modular aufgebaut:

- **main.py** – Startet das System ohne GUI. Es wird nur die Kommunikation eingerichtet, Sensorwerte empfangen und die Aktoren werden gesteuert.
- **config.py** – Definiert MQTT-Einstellungen, GPIO-Zuweisungen und Schwellwerte.
- **mqtt/subscriber.py** – Abonniert alle relevanten Sensor-MQTT-Topics und übergibt empfangene Werte an die Steuerlogik.
- **logic/**
    - **control.py** – Beinhaltet die Steuerlogik zur automatisierten Regelung der Aktoren (z. B. Temperatur > Schwelle → Heizung ein).
    - **actions.py** – Implementiert die konkrete GPIO-Steuerung für Licht, Heizung, Lüfter und Alarm.
- **state/state.py** – Enthält zentrale Variablen zum aktuellen Systemzustand (z. B. Alarm aktiv, Lüfter eingeschaltet). Wird auch von der GUI verwendet.
- **smarthome.py** – Implementiert und startet die grafische Benutzeroberfläche (Web-GUI) und ermöglicht die manuelle Steuerung. Dieses Modul initialisiert intern auch die benötigten Komponenten wie MQTT, Zustandsverfolgung und Steuerlogik.

Der Aktor-RPi kann sowohl im reinen Headless-Modus (nur main.py) als auch mit Benutzeroberfläche (smarthome.py) betrieben werden.

Anders als im Project Proposal ursprünglich geplant, ist hier die Architektur GUI und Systemlogik eng gekoppelt: Statt MQTT zu verwenden, greift die GUI direkt auf die gemeinsam genutzten Module state, actions und control zu. Dadurch entfallen Latenzen und Synchronisierungsprobleme zwischen Frontend und Logik.

Communication Protocols

Für die Kommunikation zwischen Sensor- und Aktor-RPi wird **MQTT** verwendet. Ein zentraler Broker (z. B. Mosquitto) läuft lokal im Netzwerk unter der IP-Adresse des Aktor-RPis. Die Topics sind wie folgt definiert:

Sensor-RPi (Publisher)

Topic	Verwendung	Einheit
sensor/temperature	Raumtemperatur	°C
sensor/light	Helligkeit	Integerwert
sensor/safety	Abstand vor dem Lüfter	Millimeter
sensor/alarm	Abstand im Eingangsbereich	Zentimeter

Diese Topics werden aus config.py geladen und vom MQTT-Client (mqtt/client.py) verwendet.

Aktor-RPi (Subscriber)

Der Aktor-RPi empfängt diese Daten über mqtt/subscriber.py. Die Verarbeitung erfolgt in control.py, die Aktoren werden über actions.py angesteuert.

I^2C

Neben der netzwerkbasierten Kommunikation über MQTT wird auf Hardware-Ebene das I^2C-Protokoll eingesetzt. Dieses serielle Bus-System ermöglicht die Verbindung mehrerer Sensoren über nur zwei Datenleitungen (SDA und SCL).

In diesem Projekt wird I^2C verwendet für:

- **Sense HAT** zur Temperaturmessung
- **VL6180X ToF-Sensor** zur Abstandserkennung vor dem Lüfter

Die Ansteuerung erfolgt über Python-Bibliotheken, die auf das I^2C-Interface des Raspberry Pi zugreifen. Die Nutzung von I^2C ermöglicht eine einfache, effiziente Erweiterbarkeit durch weitere Sensoren auf demselben Bus.

GUI-Kommunikation

Die Web-GUI (in smarthome.py) läuft lokal auf dem Aktor-RPi. Sie kommuniziert nicht über MQTT, sondern greift direkt auf die gemeinsamen Module state.py, actions.py und control.py zu. Damit sind alle Schaltzustände und Sensorwerte in Echtzeit abrufbar und manipulierbar.

Sensors / Actuators

Sensor	Beschreibung	Einheit	MQTT-Topic
Sense HAT	Temperaturmessung im Raum	°C	sensor/temperature
Photoresistor	Lichtmessung (analog, RC-Timing)	Integerwert	sensor/light
VL6180X (ToF)	Abstandserkennung vor dem Ventilator	mm	sensor/safety
HC-SR04	Bewegungserkennung im Eingangsbereich	cm	sensor/alarm

Aktor	Beschreibung	Steuerung durch
LED (Licht)	Leuchtet bei Dunkelheit oder manuell	automatisch + GUI
LED (Heizung)	Leuchtet bei niedriger Temperatur	automatisch + GUI
Servomotor (Lüfter)	Aktiv bei hoher Temperatur, Abschaltung bei Nähe	automatisch + GUI + Sicherheit
Buzzer (Alarm)	Erklingt bei Bewegung bei scharfem System	automatisch, wenn aktiv

Documentation of encountered problems and their solutions

1. Problem: Fehlende einheitliche Projektumgebung

- **Beschreibung:** Zu Beginn war unklar, wie die Projektstruktur aussehen soll und wo welche Dateien abgelegt werden. Ohne ein technisches Grundgerüst hätte jedes Teammitglied potenziell in eine andere Richtung programmiert.
- **Lösung:** Es wurde eine einheitliche Projektstruktur mit Ordnern für sensor-rpi und aktor-rpi definiert. Außerdem wurden config.py, state.py und eine klare Topic-Namenskonvention frühzeitig eingeführt. Das hat späteren Integrationsaufwand erheblich reduziert.

2. Problem: Unzuverlässiger ToF-Sensor (VL6180X)

- **Beschreibung:** Der Time-of-Flight-Sensor **VL6180X** hat teilweise keine Messwerte zurückgegeben und einen „Input/Output Error“ (I/O Error) ausgelöst.
- **Ursache:** Vermutlich ein Wackelkontakt oder instabile Verbindung auf dem I^2C-Bus.
- **Lösung:** Der Sensor wurde mechanisch leicht angehoben und besser fixiert, sodass kein Kontaktproblem mehr auftrat. Danach funktionierte die Messung stabil.

3. Problem: Aktoren konnten nicht über GUI angesteuert werden

- **Beschreibung:** Die grafische Benutzeroberfläche zeigte zwar Sensorwerte an, aber Klicks auf Buttons (z. B. Licht AN) führten nicht zur erwarteten Aktion.
- **Ursache:** Die GUI war zunächst nicht korrekt mit dem Zustands- und Steuerungssystem verbunden.
- **Lösung:** Direkter Zugriff auf das state- und actions-Modul aus der GUI wurde eingerichtet. Damit konnten manuelle Steuerbefehle sofort ausgeführt und der Zustand korrekt aktualisiert werden.

4. Problem: Lüfter schaltet sich nicht bei Annäherung ab

- **Beschreibung:** Der Ventilator (Servomotor) lief weiter, obwohl der ToF-Sensor eine Person in unmittelbarer Nähe erkannte.
- **Ursache:** Die Sicherheitslogik prüfte den Abstandswert nicht korrekt.
- **Lösung:** Der Schwellenwert in control.py wurde angepasst und die Logik so erweitert, dass der Abstand bei jedem Steuerzyklus geprüft wird und der Lüfter sofort abgeschaltet wird, wenn der Abstand unterschritten wird.

5. Problem: Alarm löst auch im deaktivierten Zustand aus

- **Beschreibung:** Obwohl das Alarmsystem über die GUI deaktiviert wurde, wurde bei erkannter Bewegung der Buzzer ausgelöst.
- **Ursache:** Der Status „Alarm aktiv“ wurde in der Steuerlogik nicht richtig abgefragt.
- **Lösung:** Die Steuerbedingung wurde angepasst, sodass der Buzzer nur bei aktiviertem Alarmsystem (aus state.py) ausgelöst wird. Seitdem funktioniert die Alarmsteuerung zuverlässig.

Future Work and Possible Improvements

1. Erweiterung der GUI-Funktionalität

Die aktuelle Benutzeroberfläche erfüllt ihre Grundfunktion, könnte jedoch durch folgende Punkte verbessert werden:

- Live-Visualisierung von Sensorwerten als Diagramme
- Logging-Funktion mit Export (CSV/JSON)
- App-Integration

2. Historie und Datenanalyse

Momentan werden alle Sensorwerte nur in Echtzeit angezeigt. Eine Speicherung und Analyse vergangener Messwerte würde zusätzliche Einblicke ermöglichen, z. B.:

- Temperaturverläufe im Tagesverlauf
- Bewegungshäufigkeit im Eingangsbereich
- Erkennung von Nutzungsmustern zur Automatisierung

3. Benachrichtigungssystem

Ein Alarmton vor Ort ist hilfreich, aber eine zusätzliche Benachrichtigung auf mobile Geräte (Push-Notification, E-Mail, Telegram-Bot) würde den Nutzen des Systems deutlich erhöhen – vor allem bei Abwesenheit.

4. Erweiterung des Sensorsystems

Das System könnte durch zusätzliche Sensoren weiterentwickelt werden, z. B.:

- Luftfeuchtigkeits- und CO_2-Sensor zur Luftqualitätsüberwachung
- Tür-/Fensterkontakte für ein vollständigeres Sicherheitssystem
- Bewegungserkennung über Kamera oder KI-basierte Auswertung

5. Benutzerrechte & Rollen

Aktuell kann jede Person mit Zugang zum Gerät die Steuerung übernehmen. Für eine realistische Anwendung wäre eine Rechteverwaltung sinnvoll (z. B. „Admin“, „Nur Anzeige“, „Keine Steuerung“).

6. Cloud-Anbindung & Fernzugriff

Zur ortsunabhängigen Steuerung könnte das System mit einem Cloud-Dienst verbunden werden. Eine Webplattform oder Smartphone-App könnte dabei als zentrale Steuer- und Analysezentrale dienen.

7. Sprachsteuerung

Integration eines Sprachassistenten (z. B. Google Assistant oder Alexa) zur sprachbasierten Steuerung der Aktoren (z. B. „Licht einschalten“) würde den Bedienkomfort erhöhen.

Zuständigkeiten

Kernfunktionen	Person
Lichtautomatisierung	Florent Salihi
Temperaturgeregelte Klimasteuerung	Stefan Djukic
Sicherheitsabschaltung des Ventilators	Harun Imamovic
Abstandsbasiertes Alarmsystem	Arian Kycyku
Zentrale grafische Benutzeroberfläche	Philipp Riener