Technical Report : OPCUA-Netzwerk

Johannes Horst, Manuel Zimmermann, Patrick Sabau, Saniye Ogul, Stefan Ries und Tobias Schotter

I. EINLEITUNG

Ziel dieses Projektes soll die Entwicklung eines Sensor-Netzwerks für den Heimbereich auf Basis von OPC-UA sein. Verschiedene Sensorknoten sollen mit entsprechender Sensorik und Aktorik ausgestattet sein, um bestimmte Daten des eigenen Zuhauses zu sammeln. Eine grafische Oberfläche ermöglicht dem Nutzer das Abrufen der Daten und außerdem Steuerungsfunktionalitäten, um z.B. eine Heizung oder eine Lüftungsanlage ein- bzw. auszuschalten. Hierdurch ergeben sich große Energiesparpotentiale, da man so nur lüftet bzw. heizt, wenn die Luftqualität/Temperatur dies erfordert.

OPC-UA steht hierbei für „Open Platform Communication – Unified Architecture“ und ist eine aktuelle Technologie aus dem Industrie-4.0-Umfeld. OPC-UA soll die plattformunabhängige Machine-to-Machine Kommunikation ermöglichen. Dies wird durch ein zu modellierendes Informationsmodell möglich, welches den realen Sachverhalt abbildet und von einem zentralen Server verwaltet wird. Clients können sich mit dem Server verbinden und dort relevante Informationen ablegen bzw. diese abrufen oder durch Publish/Subscribe auch vom Server Daten erhalten. Die Informationen werden auf dem Server hierbei in einer Baumstruktur verwaltet. Durch die entsprechende Modellierung der Knoten des Baums erhalten diese z.B. durch die Festlegung eigener Datentypen eine semantische Bedeutung.

II. OPCUA - SEMANTIK

Basis eines OPC-UA Server ist ein sog. Informationsmodell (Semantisches Modell), welches die zu einem Anwendungsfall vorliegenden Informationen gliedert. Analog zur Automatisierungspyramide werden die Informationen zunächst hierarchisch in einer Baumstruktur gegliedert. Ergänzt wird das Informationsmodell durch nicht hierarchische Verbindungen zwischen den Knoten, damit auch komplexere Sachverhalte abgebildet werden können.

Durch die Verbindung zweier Knoten mit einer entsprechenden Referenz entsteht eine semantische Bedeutung. So stellt beispielsweise die Verbindung „BME280 – measures - Temperature“, analog zu anderen semantischen Beschreibungen eine Verbindung von Subjekt, Prädikat und Objekt dar. Möchte man nun z.B. alle Sensoren finden, die die Temperatur messen können, so sucht man nach allen Verbindungen mit Subjekt x, die nach dem Schema „x – measures – Temperature“ aufgebaut sind, was in dem Modell in Abbildung 1 die zwei Knoten BME280 und DHT11 zurückliefern würde.

Durch das Festlegen von semantischen Regeln kann ein erstelltes Informationsmodell validiert werden. So kann beispielsweise festgelegt werden, dass in dem Beispiel aus Abbildung 1 ein Sensor mit Sensor Type „Digital Sensor“ keine „Connected To“-Referenz auf einen GPIO-Pin aufweisen darf, der vom Typ „AnalogPin“ ist.

```
IF x - has Sensor Type - DigitalSensor
AND x - connected to - y - has Pin Type - AnalogPin:
    VALIDATION = FALSE
```

Listing 1. Semantische Validierung Digitaler Sensor

Im vorliegenden Anwendungsfall heißt dies praktisch, dass ein Sensor, der ein digitales Signal ausgibt, nicht an einen Pin des Raspberry Pis angeschlossen werden darf, der nur für Analoge Signale geeignet ist.

III. HARDWARE

TODO

IV. SENSOREN/AKTOREN UND KOMMUNIKATION

Die verwendete Hardware-Plattform ist bei allen Sensorknoten ein Raspberry-Pi. Da der vorliegende Anwendungsfall keine große Leistungsfähigkeit der Hardware benötigt, wird für die Sensorknoten ein Model 3B verwendet. Um skalierbar zu bleiben und auch einen Puffer an Leistung für eventuelle zukünftige Funktionalitäten zu haben, wird für den Server ein Raspberry-Pi Model 4 verwendet.
Für das Auslesen der Sensoren und das Ansteuern der Aktoren werden zum einen direkt die GPIO-Pins des Raspberry-Pis verwendet und zum anderen dessen vorhandene I2C-Schnittstelle. Analoge Sensoren können vom Raspberry-Pi nicht direkt ausgelesen werden, da dieser über keine integrierten Analog-Digital-Converter (ADC) verfügt. Hierfür werden die im auf der Platine aufgebrachten Microcontroller vorhandenen ADCs verwendet. Deren Werte können ebenfalls über I2C abgefragt werden.
Die gemessenen Werte werden je nach Art des Sensors entweder zyklisch abgefragt und mittels OPC-UA an den Server übergeben oder bei Werteänderung azyklisch per OPC-UA übertragen.

A. OPC-UA Server

Die Server-Anwendung importiert das Informationsmodell, legt mit diesem den Server an und startet ihn. Realisiert wurde diese mit der Python Library "python-opcua" von FreeOpcUa. Das verwendete Informationsmodell wurde mit dem Modellierungstool Siome erstellt und orientiert sich an dem Beispiel aus Abbildung 1. Da weder Siome noch die verwendete Library direkt Validierungsfunktionalitäten

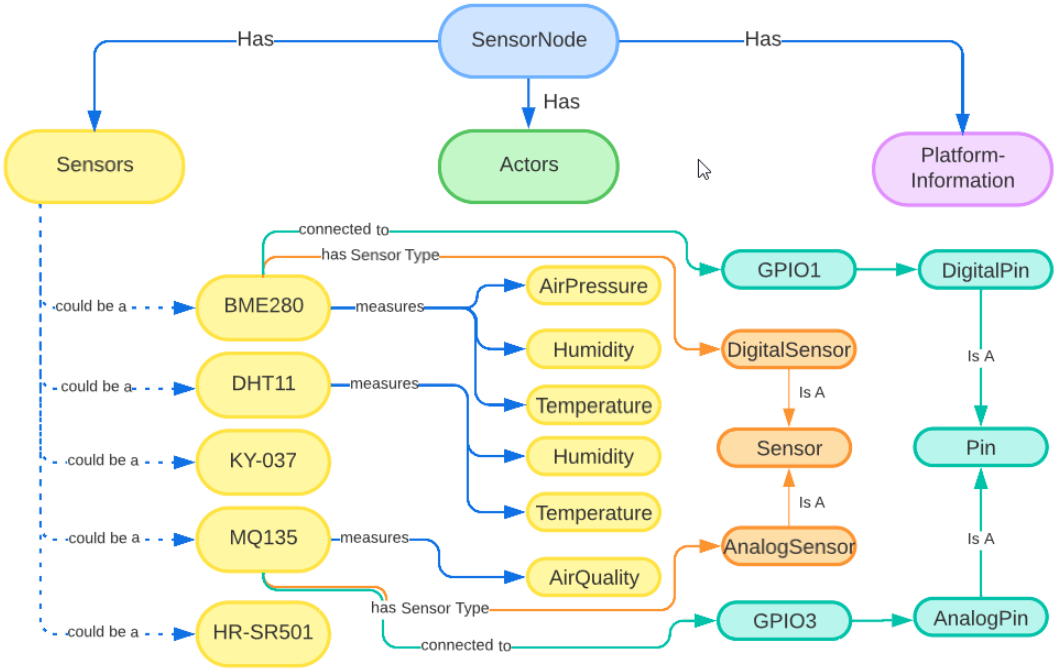

Fig. 1. Beispiel Semantisches Modell Sensornetzwerk

CO2-Gehalt	LED aktiv	Alarm
<800 ppm	Grün	-
>800 ppm & <1200 ppm	Gelb	-
>1200 ppm	Rot	+

für das erstellte Informationsmodell unterstützen, wurde aus Zeitgründen hierauf verzichtet.

B. OPC-UA Clients

Die Client Anwendung wurde ebenfalls in Python mit der gleichen Bibliothek verwirklicht. Außerdem wird hier neben dem Ansteuern/Auslesen der Sensoren/Aktoren auch die Kommunikation mit dem OPC-UA Server und eine Steuerungsfunktionalität direkt am Knoten mit den vorhandenen Buttons realisiert.

Im Hauptskript **Client.py** werden zunächst alle Sensor- und Aktorobjekte angelegt, sowie benötigte Threads gestartet und Callbacks hinterlegt.

Im **„measurement_thread“** werden Temperatur, Luftfeuchtigkeit, Luftdruck und Luftqualität einmal pro Minute abgefragt und die entsprechenden Werte auf dem Display aktualisiert, sowie an den OPC-UA Server übermittelt. Entsprechend des Wertes für Luftqualität werden die vorhandenen LEDs geschalten, bzw. zusätzlich über den vorhandenen Buzzer ein Alarmton ausgegeben.

Der **Bewegungssensor** stellt einen Event-Handler zur Verfügung, der abonniert werden kann. Bei Auftreten einer Bewegung wird das Event ausgelöst und ruft die hinterlegte Callback-Funktion auf, welche dem OPC-UA Server Anwesenheit signalisiert. Sobald eine Minute lang keine Bewegung mehr gemessen wird, wird das Event erneut ausgelöst und die Callback-Funktion setzt den entsprechenden Wert im Informationsmodell auf Abwesenheit.

Um auf dem **LCD-Display** unterschiedliche Texte (Screens) anzeigen zu können und außerdem eine Referenzmessung des Luftqualitätssensors auszulösen, wurde ebenfalls eine **Steuerungsfunktionalität** mittels der Buttons implementiert. Das LCD-Display führt intern eine Liste mit anzuzeigenden Texten, die durch Knopfdruck durchgeschalten werden können. Ebenfalls durch Knopfdruck kann die erwähnte Referenzmessung des Luftqualitätssensors eingestellt und gestartet werden. Für das Ansteuern des **LCD-Displays** aus OPC-UA heraus wurde hierfür ein eigener Screen hinterlegt. Bei Änderung des entsprechenden Wertes im Informationsmodell wird mittels **Publish/Subscribe** der anzuzeigende Text aktualisiert.

V. BACKEND + DATENBANK

Das Backend wurde mit dem Framework **FastAPI** in Python umgesetzt und ist die einzige Schnittstelle zur Datenbank **MongoDB**.

A. Datenbank

Um eine MongoDB lokal (für die Entwicklung) zu starten:

- Herunterladen von MongoDB über die offizielle Website oder einen Package Manager (z.B. ABT)
- Ausführung der in der ReadMe-Datei hinterlegten Konsolenbefehle

Die Datenbank wird dabei in der Standardkonfiguration genutzt, somit ist es nicht erforderlich Nutzer oder anderes anzulegen. Lediglich sollte die Datenbank über den Standardport 27017 erreichbar sein.

MongoDB enthält zu Beginn einige Tabellen die zur Konfiguration und zur internen Konsistenz genutzt werden. Die eigentlichen Daten werden dabei in einer neuen Datenbank

(der Name dieser kann konfiguriert werden) gespeichert. Eine solche Datenbank kann mehrere Collectionen beinhalten, das Äquivalent zu Tabellen in einer SQL Datenbank. Diese Collectionen beinhalten i.d.R ähnliche Documente im Format BSON (Binary JSON).

B. Datenformat

Es werden Datenformate genutzt, die jeweils in Collections gespeichert werden : **Sensoren**.

Listing 2 zeigt ein Sensor Objekt, wie es in der Datenbank gespeichert sein könnte.

```
{
    "_id": ObjectID(abc123def456),
    "sensornode": "SensorNode_1",
    "sensorname": "BME280",
    "sensortyp" : "AirPressure",
    "unit"      : "hPa,
    "value"     : 1040.6999999999991,
    "timestamp" : "2022-12-06T15:15:28.112Z"
}
```

Listing 2. Sensor Objekt

Das Feld **'_id'** ist ein Primärschlüssel, welcher einzigartig ist und automatisch von MongoDB vergeben wird. Mit diesem können Objekte eindeutig referenziert werden. Über **'sensornode'** wird die Bezeichnung der einzlenen Nodes gespeichert. Das Feld **'sensorname'** speichert den technische Namen eines Sensors als String und das Feld **'sensortyp'** die tatsächliche Sensorbezeichnung. Durch Kombination dieser drei Werte können die Sensoren eindeutig zugeordnet werden, sowie im Frontend passend der per Name angezeigt werden.

Die Datenbankobjekte werden im Code als Pydantic Dataclasses hinterlegt, was das Parsen dieser Objekte (z.B. als JSON Payload) erleichtert. Dabei werden die Klassen mit dem Decorator **@dataclass** verziert und erhalten **TypeHints** mit entsprechden Datentypen

```
@dataclass
class Sensor_value_dto():
    sensornode: str
    sensorname: str
    sensortyp: str
    value: Union[float,bool]
    timestamp: datetime.datetime
    unit: str = ""
```

Listing 3. Sensor Dataclass

```
@dataclass
class actuator_list_dto:
    actuator_node : str
    actuator_act  : str
    actuator_value: Union[str,float,bool,None]
    actuator_dtype: str
```

Listing 4. Actuator Dataclass

C. Backend

Mittels dem Python Framework **FastAPI** werden diverse Endpoints bereitgestellt, die das Erstellen und Ausgeben der Datenobjekte ermöglichen.

Die Sensoren können über folgende Routen ausgegeben werden:

- **GET /sensornames**: Gibt eine List aller zur verfügung stehenden Sensoren zurück.
- **GET /sensornodes**: Gibt eine List aller zur verfügung stehenden Knoten zurück.
- **GET /sensorvalues/current**: Gibt den aktuellen Wert jedes Sensors zurück.
- **GET /sensorvalues**: Gibt eine List von Sensoren zurück, welche sich nach ihren Attributen filtern lassen.

Die Aktoren können über folgende Routen ausgelesen und gesteuert werden:

- **GET /actuatornames**: Gibt eine List aller zur verfügung stehenden Aktorenbezeichnungen zurück.
- **GET /actuators**: Gibt eine Liste von JSON-Objekten zurück mit entsprechenden Informationen einzelner Aktoren.
- **GET /actuators/filter_by_actuatorname**: Gibt eine Liste von JSON-Objekten, gefiltert bei Aktorbezeichnung zurück.
- **PUT /actuators**: Ermöglicht per Übergabe von Aktorenknoten und Bezeichnung den Wert des Aktors zu ändern.

Die Endpunkte nutzen dabei ein automatisiertes Umwandeln zu entsprechenden Dataclasses. Dies sorgt dafür, dass Pflichtfelder übergeben werden und nicht benötigte Elemente verworfen werden. Über einen PyMongo Client werden diese Dataclasses also entweder von Route zur Datenbank weitergeleitet, oder aus der Datenbank zur Route. Die Verbindung erfolgt dabei über eine **MONGO_URI**, welche Hostnamen und Port beinhaltet. Dieser String sieht wie folgt aus: *mongodb://localhost:27017/*.
Als tatsächlicher Webserver wird **uvicorn** genutzt, welcher im **__main__.py** gestartet wird. Dies ermöglicht es, das Backend mittels des Befehlt *python -m backend* zu starten.

D. Tests

In einem separaten Testscript **test_backend.py** werden die Routen getestet. Hierbei wird jedoch eine Testtabelle in der Datenbank genutzt, um nicht mit anderen Daten zu interferieren. In einer Setup-Methode werden dabei die Referenzen auf die Tabelle ausgetauscht. Der Server wird ebenfalls durch einen **TestClient** ersetzt, welcher in dem FastAPI Framework integriert ist.

Die Tests können mittels des Befehls *python -m pytest -s* gestartet werden.

VI. FRONTEND

Das Frontend wurde mithilfe der Frameworks Angular entwickelt.

A. Architektur

B. Entwickelte Komponenten