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 OPCUA 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 so viel Lüftung und Heizung, anhand der Luftqualität/Temperatur gesteuert werden.

OPCUA steht hierbei für „Open Platform Communication – Unified Architecture“ und ist eine aktuelle Technologie aus dem Industrie-4.0-Umfeld. OPCUA 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 OPCUA 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. Um z.B. alle Sensoren zu finden, die die Temperatur messen können, wird nach nach allen Verbindungen mit Subjekt x, die nach dem Schema „x – measures – Temperature“ aufgebaut sind gesucht, 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

A. Wahl der Sensoren / Aktoren

Zentraler Punkt unseres Projektes stellte auch die Entwicklung echter Sensorknoten dar, um auf reale Daten von Sensoren zurückgreifen bzw. verschiedene Aktoren ansteuern zu können. Hierbei wurden im ersten Schritt Sensoren und Aktoren gesucht, welche für unser Projekt in Frage kommen und anschließend nach Priorität geordnet, um die Abarbeitungsreihenfolge zu bestimmen.

Sensoren:

- BME280: Temperatur, Luftfeuchtigkeit, Luftdruck
- HR-SR501: Anwesenheitserkennung/Bewegungsmelder
- KY-037: Mikrofon/Schallpegel zur Bestimmung der Lautstärkenbelastung

Aktoren:

- LEDs (Rot – Gelb – Grün – Blau) als Statusanzeige
- LCD-Display zur Anzeige verschiedener Informationen (erweitert mit 4 Eingabetasten für mögliche Benutzereingabe)
- Piezo-Buzzer zur Wiedergabe von Statustönen

B. Konzeptionierung

Bei der Entwicklung des Knotens wurde vor allem darauf geachtet, dass dieser eine generische Schnittstelle für die Sensoren/Aktoren bereitstellt, um diese zu einem späteren Zeitpunkt um weitere Senor-/Aktorelemente erweitern zu können. Hierfür wurden zunächst aus den gegebenen Elementen die benötigten Funktionen (Special Functions, kurz SF) extrahiert:

- Analog-Digital-Wandlung (z.B. MQ-135, KY-037)
- Digital Lesen (z.B. Hr-SR501, LCD-Display Taster)
- Digital Schreiben (z.B. LEDs, LCD-Display)

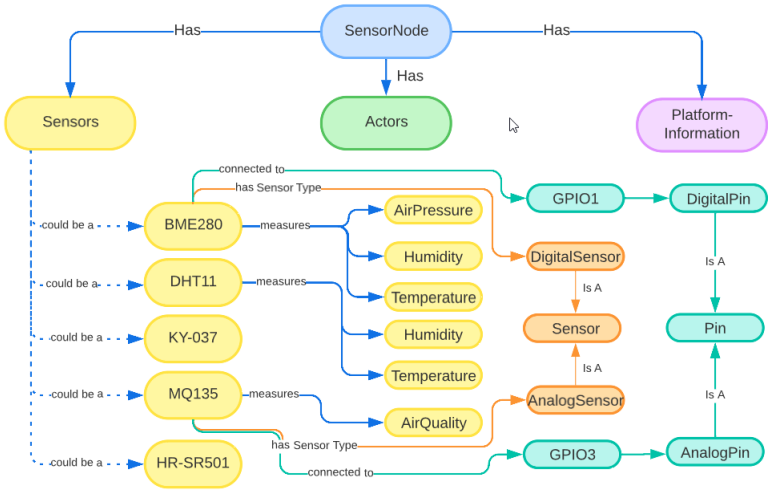

Fig. 1. Beispiel Semantisches Modell Sensornetzwerk

- Frequenz-Modulation (Piezo-Buzzer)

Zudem wurde ein genormter Daten-Bus implementiert, welcher es ermöglicht, digitale Sensoren/Aktoren und Erweiterungsplatinen anbinden zu können. Hier kommt der I²C-Bus zum Einsatz, welcher eine synchrone Kommunikation über eine Zweidrahtleitung ermöglicht. Der Raspberry-Pi übernimmt dabei die Rolle des Masters, welcher die Kommunikation leitet. Die Adressierungsfunktionalität erlaubt es dabei bis zu 126 Slaves einzubinden. Auch der eingesetzte BME280 implementiert diese Schnittstelle und wird in unserem Projekt darüber angesteuert.

Bei der Analyse benötigten Funktionen stellte sich schnell heraus, dass die GPIO-Pins des Raspberry-Pi's diese Anforderungen nicht erfüllen kann, da dieser beispielweise keine ADC-Funktionalität bereitstellt, sowie diverse Sensoren auf 5V Spannungslevel kommunizieren, während die IO's es Raspberry-Pi's nur 3,3V kompatibel sind. Deswegen wurde der I²C-Datenbus um einen Mikrocontroller erweitert. Dieser dient als Erweiterung der GPIOs des Pi's, stellt die 5V Kompatibilität bereit, um die restlichen Anforderungen zu erfüllen.

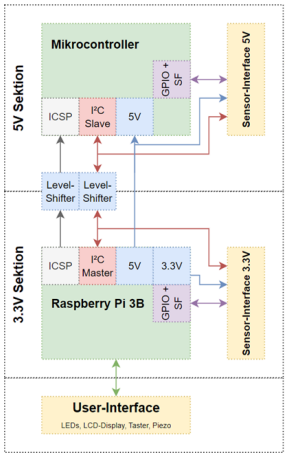

Fig. 2. Interne Verbindungen und Schnittstellen auf einem Sensorknoten

C. Hardwareentwurf

Anschließend wurde ein Schaltplan mit den benötigten Komponenten entworfen und mit einem Steckbrettaufbau validiert. Für eine einheitliche und saubere Verkabelungslösung wurde anschließend eine Aufsteckplatine passend zu dem Raspberry Pi entworfen, welche alle notwendigen Hardware-Komponenten enthält. Ebenfalls wurden alle Userinterface-Elemente wie Display, Taster, Piezo-Buzzer uvm. mit aufgebracht, welche auf jeden Knoten vorhanden sind. Seitlich befinden sich das 3.3V und 5V Sensorinterface, ausgeführt als Stiftleiste, an welches dynamisch die Sensoren angeschlossen werden kann.

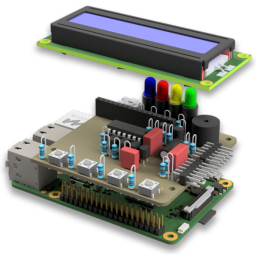

Fig. 3. Hardware Grundaufbau für einen Sensorknoten (ohne Sensoren)

D. Mikrocontroller

Als Mikrocontroller kommt ein Atmel ATTiny84 zum Einsatz. Auf diesem wurden die generischen Funktionen wie die ADC-Konversation, digitales Schalten und Lesen der Spannungspegel, sowie PWM-Funktionalitäten implementiert. Viele der Funktionalitäten lassen sich ebenfalls parametrisieren, wie die Referenzspannung der ADC-Wandlung, konfigurierbare Pullup-Widerstände beim Lesen der digitalen Eingänge uvm. Dies erlaubt eine generische Ansteuerung der Pins über den PI und erhöht somit die Flexibilität, da die Pins des Mikrocontrollers nicht mehr fest an bestimmte Aufgaben oder Funktionalitäten gebunden sind.

Alle Funktionen basieren auf dem Prinzip 0-N Eingabe-Parameter, Verarbeitung und Rückgabe von 0-N Ergebnissen. Anhand dieses Aufbaus wurde ein I²C Protokoll entworfen, welches einen Remote-Procedure Call durch den I²C Master erlaubt. Der Funktionsaufruf wird dabei gestartet, sobald der Master die benötigten Parameter übertragen hat und anschließend einen Restart (RST) an den Slave sendet. Sollte der Funktionsaufruf zum Zeitpunkt der Rückübertragung nicht vollständig beendet sein, wird die sogenannte „Clock-Hold“ Sonderfunktionalität des I²C Protokolls genutzt, welche es dem Client ermöglicht, die Übertragung zu pausieren. Nach Abschluss werden anschließend die Resultate an den Master zurückgesendet. Jedes übertrage Byte wird von dem empfangenden Teilnehmer bestätigt, andernfalls wird die Übertragung abgebrochen. Die letzten beiden Bytes entsprechen einem Status-Code, welcher über die erfolgreiche Ausführung der Prozedur bzw. dem ggf. entstandenen Fehler informiert, sowie eine Checksumme, welche eine Erkennung von Übertragungsfehler ermöglicht.

Eine erstellte Python-Bibliothek implementiert die korrespondierenden RPC-Funktionsaufrufe auf dem Raspberry-Pi und prüft die Datenübertragung/Fehlercodes. Alle RPC-Aufrufe sind mit Unit-Tests abgedeckt, welche zusätzlich ebenfalls einen Hardware-Check durchführen. Hierfür müssen in 2er Gruppen eingeteilt Pins mittels Jumperkabeln verbunden werden, welche sich durch den entsprechenden Unit-Test gegenseitig auf Funktionalität testen.

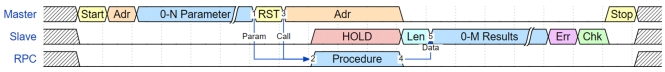

Fig. 4. Kommunikationsablauf zwischen Raspberry Pi und Mikrocontroller über I²C

IV. Steuerung Sensoren&Aktoren

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

B. OPCUA 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 OPCUA 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.

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

Im **„measurement_thread“** werden Temperatur, Luftfeuchtigkeit, Luftdruck und Luftqualität einmal pro Minute abgefragt und die entsprechenden Werte auf dem Display aktualisiert, sowie an den OPCUA 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 OPCUA 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 OPCUA 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 Nodes gespeichert. Das Feld **'sensorname'** speichert den technischen Namen eines Sensors als String und das Feld **'sensortyp'** die tatsächliche Sensorbezeichnung. Durch Kombination dieser drei Werte können die Sensoren eindeutig zugeordnet, 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 Liste aller zur Verfügung stehenden Sensoren zurück.
- **GET /sensornodes**: Gibt eine Liste aller zur Verfügung stehenden Knoten zurück.
- **GET /sensorvalues/current**: Gibt den aktuellen Wert jedes Sensors zurück.
- **GET /sensorvalues**: Gibt eine Liste von Sensoren zurück, welche sich nach ihren Attributen filtern lassen.

Die Aktoren können über folgende Routen ausgelesen und gesteuert werden:

- **GET /actuatornames**: Gibt eine Liste 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 und nicht benötigte Elemente verworfen werden. Über einen PyMongo Client werden diese Dataclasses also entweder von der Clientanfrage zur Datenbank weitergeleitet, oder umgekehrt aus der Datenbank zum Client. 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.

VI. FRONTEND

Das Frontend wurde mithilfe des Admin-Dashboard ngx-admin auf Basis von Angular 9+ und Nebular entwickelt. Mittels Angular wird eine Weboberfläche komponentenbasiert zur Verfügung gestellt. Für die serverseitige Anwendung benötigt Angular zudem Node.js, eine JavaScript-Laufzeitumgebung. Zudem wird ngx-admin mit dem Eva Design System unterstützt, um UI-Design zu vereinfachen.

A. Architektur

In der unteren Abbildung 5 ist die Architektur des Frontends zu sehen. Eine Komponente besteht aus einem Template (HTML-Datei) und einem Style (SCSS-Datei). Diese werden als Metadaten im Dekorator der Komponente festgelegt und stellen die Ansicht dar. Mithilfe der Datenbindung kann das Template mit der Komponente Daten austauschen und gegebenenfalls Events ausführen. Bei Bedarf können Komponenten durch die Dependency Injection Zugriff auf eine Serviceklasse erhalten. Somit können wiederverwendbare Funktionen oder Daten aus dem Backend zur Verfügung gestellt werden. In den Modulen werden Komponenten, Module, Services usw. gruppiert und verwaltet. In Angular werden zwei Modularten unterschieden. Jedes Angular Projekt besitzt ein Root-Modul, das app.module.ts heißt. Diese ist dazu da, um die gesamte Webanwendung zu verwalten und diese wird beim Start der Anwendung als erstes geladen

und initialisiert. Mit Feature-Modulen kann der Code, der sich auf eine bestimmte Funktionalität oder ein bestimmtes Feature bezieht, von anderem Code getrennt werden und bleibt somit organisiert. Das Frontend wird auf dem Raspberry Pi Server gehostet. Für das Hosting wird ein Apache 2 Webserver verwendet.

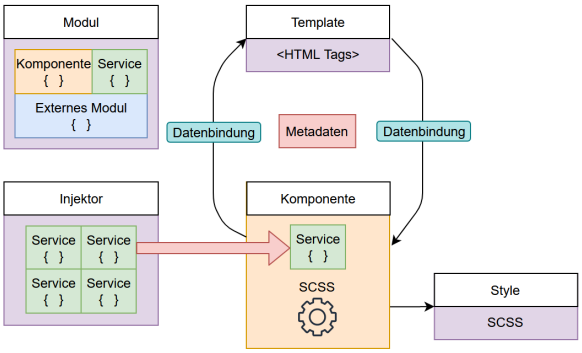

Fig. 5. Angular Architektur des Projektes

B. Entwickelte Komponenten

Für alle Diagramme wurde die Bibliothek ECharts verwendet. Zusätzlich zu den vorhandenen Komponenten vom ngx-admin wurden folgende weitere Komponenten entwickelt.

- **AirQualityChart** Bildet die Werte des Luftqualitätssensors in einem Liniendiagramm ab.
- **AirQuality** Visualisiert die Werte des Luftqualitätssensors unter Verwendung der AirQualityChart Komponente.
- **buzzer-frequenzy** Ein Eingabefeld zum Festsetzen der Frequenz des Piezoelements.
- **gant** Das Gantt-Diagramm zeigt an, von wann bis wann der Bewegungsmelder Anwesenheit, bzw. Abwesenheit erkannt hat.
- **lcd-input** Ein Eingabefeld zum Festsetzen des LCD-Displays auf dem Sensorknoten.
- **lineChartComponent** Ein universell verwendbares Liniendiagramm.
- **switch** Ein Knopf, der an- und ausgeschaltet werden kann.
- **temperatureGauge** Ein Tachometer zum Visualisieren von verschiedenen Werten.
- **tempSensorCard** Kombination aus dem LineChart-Component und dem temperatureGauge Komponente. Stellt eine zweiseitige Karte dar, die auf der einen Seite Luftdruck, Luftfeuchtigkeit und Temperatur in einem Liniendiagramm darstellt und auf der anderen Seite ein Tachometer mit den aktuellen Werten.
- **sensornode-dashbord** Stellt das eigentliche Dasboard für jeden Knoten dar. In dieser Komponente werden die oben genannten Komponenten des jeweiligen Sensorknotens dargestellt. Jede Komponente wird als einzelne Karte dargestellt. Diese können sich je nach Bildschirmgröße passend anordnen.
- **led-display** Zeigt die LED-Anzeige des aktuellen Knotens.
- **Header (modifiziert)** Der vorhandene Header wurde modifiziert, um eine Zeitspanne auswählen zu können.

C. Entwickelte Services

- **BackendDataService** Ist für die Kommunikation mit dem Backend zuständig. Zum Abfragen und Senden von Daten.
- **DashboardFunctionalityService** Auslagerung der Funktionen für die sensornode-dashboard Komponente.
- **SharedDataService** Stellt allen Komponenten ein Observable des ausgewählten Zeitraums und der aktuellen Sensorknoten-ID zur Verfügung.

D. Routing

Alle verfügbaren Seiten im Projekt repräsentieren ein Dashboard eines Sensorknotens. Zugänglich sind diese über die URL: /pages/Knoten-ID. Siehe Listing 3.

```
{
  path: ':id',
  component: SensorNodeDashboardComponent
}
```

Listing 5. Knoten-ID Routing

Alle verfügbaren Knoten werden am Anfang aus dem Backend abgerufen, und anschließend links im Navigator als Menüpunkt angezeigt. Auf dem Dashboard des einzelnen Knotens werden über die aktuelle Knoten-ID die Anfragen an das Backend gesendet.

VII. Architektur

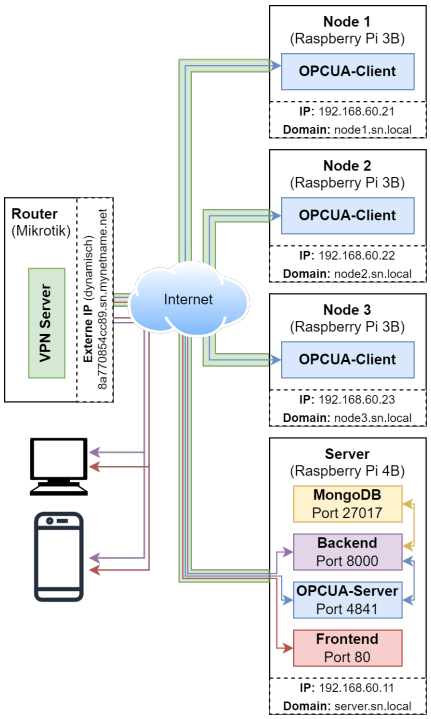

Fig. 6. Gesamtarchitektur

A. VPN-Netzwerk

Um Standortunabhängigkeit zu erreichen, wurde ein VPN Netzwerk auf einem Router aufgesetzt, in welches sich alle Sensorknoten und der Server beim Start automatisch einwählen können. Die Knoten und der Server erhalten dabei eine feste IP, sowie einen eigene interne Domain. Dies erlaubt eine direkte Kommunikation zwischen den Teilnehmern, über welches auch die OPCUA-Kommunikation abgewickelt wird. Der Server stellt zusätzlich eine Web-API, sowie das zugehörige Web-Frontend bereit, welches über diesen aufgerufen werden kann.

B. Gesamtarchitektur

Abbildung 6 zeigt eine Übersicht der Gesamtarchitektur. Zu sehen ist dabei, dass die Komponenten Frontend, OPCUA-Server, Backend auf dem selben Server laufen. Dieser Server, sowie die drei OPCUA-Clients, sind dabei über die das VPN verbunden.

Über die externe IP des Routers können Clients auf die Anwendung zugreifen.