Entwurf und Implementierung eines Multi-Computersystems zur Maschinen- und Sensor-Datenübertragung in untertätigen Bergwerken

---

Seminararbeit

von

Tim Wende

Matr.-Nr.: 3281514

Studiengang: Angewandte Mathematik und Informatik

Betreuer: Arne Köller, M.Sc, Maximilian Getz, M.Sc.

1. Prüfer: Prof. Dr. rer. nat. Alexander Voß

2. Prüfer: Univ.-Prof. Dr.-Ing. Elisabeth Clausen

Angefertigt am

Institute for Advanced Mining Technologies an der RWTH Aachen University

Aachen, den 15.12.2022

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Entwurf und Implementierung eines Multi-Computersystems zur

Maschinen- und Sensor-Datenübertragung in untertägigen Bergwerken

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Tim Wende, 3281514

Aachen, den 15.12.2022

Digital unterschrieben
von Tim Wende
Datum: 2022.12.15
10:00:05 +01'00'

Unterschrift der Studentin / des Studenten

Abstract

Eine koordinierte Kombination aus Digitalisierung und Elektrifizierung haben erhöhte Arbeitssicherheit, Produktivität und verminderte Umweltbelastung zum Ziel. Für beide Aspekte ist die Übertragung von Daten und Energie grundlegend notwendig und muss deswegen sorgfältig und strukturiert geplant werden.

Eine strikte Trennung der beiden Aspekte führt dazu, dass verschiedene Konzepte redundant umgesetzt werden müssen. Beispielsweise folgen LAN-Kabel (Digitalisierung) und elektrische Leiter (Elektrifizierung) dem selben Konzept einer direkten kabelgebunden Verbindung, sind jedoch schwierig kombinierbar und erfüllen ausschließlich ihren jeweiligen spezifischen Zweck.

Im Rahmen dieser Seminararbeit wird ein Beitrag zum öffentlichen Forschungsprojekt *HEET II* erbracht. Dieses Forschungsprojekt erprobt einen innovativen Ansatz, Probleme der Digitalisierung und Elektrifizierung als Gesamtproblem zu lösen.

Ziel dieser Seminararbeit ist es, einen methodischen, objektorientierten Softwareentwurf für den Datenfluss zwischen verschiedenen Hardwarekomponenten in diesem Gesamtproblem zu erstellen und anschließend zu implementieren. Der Datenfluss erstreckt sich dabei von der Erhebung verschiedener Maschinen- und Sensor-Daten über den drahtlosen Transport durch ein System von multiplen, Mesh-verbundenen Routern bis hin zur systematischen Speicherung der Daten.

Es sollen verschiedene Diagrammtypen der UML genutzt werden, um sich vorab einen detaillierten Überblick über die Kommunikation zu verschaffen und die erhaltenen Informationen zu protokollieren. Diese Diagramme dienen später dazu, eine sinnvoll strukturierte Kommunikationsschnittstelle implementieren zu können, welche problemlos für weitere Systeme beziehungsweise Hardwarekomponenten erweitert werden kann.

Die Speicherung der Daten erfolgt über eine Zeitreihendatenbank, welche ebenfalls an eine zu erstellende Monitoring-Software angeschlossen ist, um die erhaltenen Daten menschenlesbar zu visualisieren.

Vorab soll die entwickelte Softwarelösung jedoch erstmals an einem Versuchsaufbau am Institut getestet werden. In diesem Versuch befinden sich zum Großteil die selben Komponenten, wie im geplanten Projekt.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Für einen produktiveren, sicheren und umweltfreundlichen Bergbau sind drei zentrale Kernpunkte notwendig: Digitalisierung, Elektrifizierung und Automatisierung [3]. Viele Herausforderungen dieser Kernpunkte lassen sich im Bergbau auf die Übertragung von Energie und Daten zurückführen.

Digitalisierung soll für eine erhöhte Arbeitssicherheit und Produktivsteigerung sorgen, indem verschiedene Maschinen und Sensoren miteinander kommunizieren und etwaige Probleme frühzeitig erkennen, melden oder eigenständig versuchen zu beheben [6, 7].

Elektrifizierung soll nachhaltig für eine Verminderung von Umweltbelastungen sorgen, indem beispielsweise CO_2-arme Energiequellen die Energie für den Bergbau extern umwandeln und in ein internes Netz einspeisen.

Eine teure, kabelgebundene Installation von Infrastruktur ist häufig notwendig, um Energie und Daten in untertätigen Bergwerken zu übertragen, jedoch werden meist mehrere verschiedene Konzepte benötigt, um das Gesamtproblem der Energie- und Datenübertragung zu lösen. So benötigt man beispielsweise mehrere Hundert Meter verschiedener Kabel beziehungsweise elektrischer Leiter, welche, wenn sie nicht korrekt isoliert oder geschützt angebracht werden, potenziell als Brand- oder Explosionsquelle dienen. Des Weiteren muss diese Infrastruktur im laufenden Betrieb dauerhaft weiter ausgebaut werden, wenn der Abbau im Bergwerk fortschreitet.

Eine Umwandlung von Treibstoff in elektrische Energie vor Ort ist zwar möglich, jedoch verstoßen die dafür häufig eingesetzten Dieselmotoren gegen das Prinzip der Elektrifizierung, da sie innerhalb des Bergwerks gesundheitsschädliche Gase produzieren, welche durch größere, teurere Wettervolumenströme verringert werden müssen. Ebenfalls ist die externe Umwandlung von Energie und der darauf folgende Transport in Form von Batterien weder sonderlich prozesseffizient, noch einfach zu automatisieren, da der Austausch der Batterie einen Stillstand der Maschine voraussetzt.

In Kooperation mit unter anderem dem Projektkoordinator *INSTYTUT TECHNIKI GÓRNICZEJ KOMAG* (KOMAG) arbeitet das *Institute for Advanced Mining Technologies* (AMT) der *RWTH Aachen University* an dem öffentlichen, von der EU geförderten Forschungsprojekt *HEET II*[1]. Ziel dieses Projekts ist es, mögliche Lösungsvorschläge für die vorab geschilderten Herausforderungen im Steinkohlebergbau vorzustellen.

[1] Innovative high efficiency power sytem for machines and devices, increasing the level of work safety in underground mining excavations, http://heet2.komag.eu/

In diesem Projekt wird ein Schienennetz für eine Einschienenhängebahn aufgebaut[2]. Eine Batterie der Einschienenhängebahn wird von der Schiene dauerhaft kabellos geladen, wodurch kein Stillstand der Arbeit für einen Batteriewechsel zu Stande kommt, und dennoch Energie von externen Quellen zugeführt werden kann.

Zusätzlich befinden sich in diesem Netz verschiedene Access Points, welche ebenfalls über die Schiene mit Strom versorgt werden. Über das Mesh-basierte WLAN-Netzwerk, welches von den Access Points aufgebaut wird, werden die erhobenen Maschinen- und Sensor-Daten übertragen.

Die Access Points, Sensoren und als Steuereinheit genutzten *Revolution Pi's* (RevPi's) werden in Module zusammengefasst. Diese Module befinden sich in druckfest gekapselten Gehäusen, welche für einen geeigneten Explosionsschutz notwendig sind[3].

Ein großer Vorteil solch einer Einschienenhängebahn ist der mehrfache Nutzen des Schienennetzes. Jegliche Infrastruktur, welche für beschriebene Herausforderungen gebraucht wird, wird über dieses Schienennetz realisiert.

Im Rahmen dieser Seminararbeit wurden Teile aus den Aufgabenbereichen der Digitalisierung des AMTs übernommen und vollständig bearbeitet. Der Fokus wurde auf die Kommunikation verschiedener Hardwaremodule sowie die Steuerung des Datenflusses innerhalb der Projektgrenzen gelegt. Somit befasst sich die Seminararbeit ausschließlich mit der softwarenahen Digitalisierung in Bergwerken. Die hardwarenahe Elektrifizierung, welche bei *HEET II* im Fokus liegt, muss jedoch ebenfalls als Teil des ganzen Projekts betrachtet werden.

1.2 Ziel der Arbeit

Zum Ende der Arbeit soll ein vollständig implementierter methodisch und objektorientiert entwickelter Softwareentwurf in Form eines Prototypen zur Verfügung stehen, dessen modular erweiterter Nachfolger ohne übermäßige Einarbeitungszeit in einer realen Umgebung installiert und verwendet werden kann. Dieser Prototyp steuert den Datenfluss, welcher sich von der Erhebung verschiedener Maschinen und Sensor-Daten über den drahtlosen Transport durch ein System von multiplen, Mesh-verbundenen Routern bis hin zur systematischen Speicherung der Daten erstreckt.

[2] Einschienenhängebahnen sind besonders relevant für den Transport von Personal und Material im untertätigen Steinkohlebergbau.

[3] Im untertägigen Steinkohlebergbau kann in Folge einer Methanfreisetzung durch Abbauaktivitäten potentiell eine explosionsfähige Atmosphäre vorliegen. Betriebsmittel müssen daher nach Vorgaben des Explosionsschutzes vorliegen (z. B. ATEX).

Es sollen verschiedene Diagrammtypen der *Unified Modeling Language* (UML) genutzt werden, um sich vorab einen detaillierten Überblick über die Kommunikation zu verschaffen und die erhaltenen Informationen zu protokollieren. Diese Diagramme dienen später dazu, eine sinnvoll strukturierte Kommunikationsschnittstelle implementieren zu können, welche problemlos für weitere Systeme beziehungsweise Hardwarekomponenten erweitert werden kann.

Alle Entwicklungsschritte sollen durch eine vollständig ausgearbeitete Dokumentation nachvollziehbar dargestellt werden.

2 Stand der Technik

Konzepte zum Transport von Bergleuten und Ausrüstung sowie Gütern in Form einer Einschienenhängebahn existieren bereits seit über 50 Jahren [9, 8]. Ebenfalls sind Mesh-basierte WLAN-Netzwerke oder die Stromversorgung eines Transportmittels über dessen Fortbewegungsmittel keine neue Erfindung [14].

Eine Kombination von Teilen dieser Technologien ist jedoch neuartig beziehungsweise wurde bisher selten genutzt, wodurch das gesamte Potential der jeweiligen Konzepte nicht immer vollständig ausgenutzt wird. *HEET II* hingegen beschäftigt sich mit einer Kombination aller genannten Technologien.

Da die Entwicklungen, welche in *HEET II* realisiert werden, primär für den Einsatz in untertägigen Kohlebergwerken konzipiert wurden, konzentriert sich das Projekt primär auf die dort auftretenden Probleme[4], kann jedoch ebenfalls für weitere Arten von Bergwerken genutzt werden[5].

Im Rahmen der Digitalisierung werden verschiedenste Sensoren angesteuert, welche beispielsweise Gase messen und bei einer Grenzwertüberschreitung in das System eingreifen lassen, indem kritische Maschinen abgeschaltet werden. Alternativ wird das Bergwerk dauerhaft über ein Belüftungssystem mit Frischluft versorgt, welches zusätzlich gefährliche Gase absaugt.

Eine Grenzwertüberschreitung findet beispielsweise trotz funktionsfähiger Belüftungssysteme bei einem Gasausbruch statt. Dabei wird plötzlich eine große Grubengasmenge freigesetzt [10].

Um dieser Gefahr für Bergleute und Maschinen entgegenzuwirken, wurden bereits verschiedene Systeme entwickelt. Diese nutzen teilweise tragbare Sensoren, welche Schwefelwasserstoff und Kohlenstoffmonoxid messen und die Messwerte über Bluetooth angeschlossenen Smartphones mitteilen [1]. Andere Implementierungen bauten ein Sensornetzwerk auf, in welchem verschiedene Sensoren die Konzentrationen von Methan, Kohlenstoffmonoxid und Schwefelwasserstoff über das ZigBee-Protokoll an eine Basisstation übertragen [2]. Beide Alternativen nutzen Arduino Mikrocontroller. [5]

[4] z. B. potentiell explosionsfähige Atmosphäre
[5] insbesondere die Kommunikation über ein Mesh-basiertes WLAN-Netzwerk

In unserem Fall sind die relevanten zu beobachten Gase unter anderem *Methan* (CH_4), *Kohlenstoffmonoxid* (CO) und *Sauerstoff* (O_2). Methan kann eine potentiell explosionsfähige Atmosphäre bilden. Ein gemessener schneller Anstieg von Kohlenstoffmonoxid kann für eine Brandfrüherkennung genutzt werden. Zusätzlich kann sich eine zu geringe Konzentration von Sauerstoff oder zu hohe Konzentration von Kohlenstoffmonoxid negativ auf die Gesundheit der Bergleute auswirken.

Andere Messwerte, wie beispielsweise *Temperatur* oder *Luftfeuchtigkeit*, sind ebenfalls für den Gesundheitsschutz der Bergleute relevant. Sollten diese Messwerte festgelegte Grenzwerte über- beziehungsweise unterschreiten, ist ein längeres arbeiten im Bergwerk unter Umständen nicht möglich und muss gesetzlich geregelt verkürzt oder mit längeren Pausen versehen werden. Eine effiziente bedarfsgesteuerte Wetterführung, um unter anderem die Versorgung der Bergleute mit Frischluft zu garantieren, ist beispielsweise ebenfalls nicht ohne die Daten verschiedener Sensoren möglich [15].

Im Gegensatz zu den oben erwähnten Alternativen ist das Netz der Einschienenhängebahn erweiterbar und kann zusätzlich Maschinendaten messen und speichern. Ein alternatives allgemeingültiges Konzept oder kommerziell zu erwerbendes Produkt, welches in vergleichbaren Anwendungsfällen eingesetzt werden kann, ist dem Autor zum aktuellen Stand nicht bekannt.

3 Konzept

3.1 Bereits bestehende Systemkomponenten

3.1.1 Hardware

Da in dieser Seminararbeit größtenteils die Aufgabenbereiche der Digitalisierung bearbeitet werden, ist weder die Konzeptionierung des Schienensystem, noch die der Einschienenhängebahn oder der anzuschließenden Module ein Gegenstand dieser Seminararbeit. Diese Aufgaben werden der Elektrifizierung zugeordnet und wurden bereits vor Beginn dieses Projekts abgeschlossen.

Aus diesen existierenden Konzepten soll jedoch im Rahmen dieser Seminararbeit ein Prototyp entwickelt werden.

3.1.2 Software

Von vorherigen Prototypen existiert bereits ein Python-Skript, welches grundlegend und nicht erweiterbar einzelne Aufgaben der Kommunikation statisch implementiert hat.

Der grundlegende Aufbau einzelner Nachrichtentypen war ebenfalls bereits partiell vorhanden, wird jedoch im Rahmen dieser Seminararbeit stark überarbeitet und verändert.

3.2 Anforderungsanalyse

Die vorhandene Code-Basis ist in einen methodisch entwickelten, objektorientierten Softwareentwurf zu überführen. In diesem Entwurf soll primär der Datenfluss gesteuert und nachvollziehbar dargestellt werden.

Die reale Implementierung ist demnach ausschließlich ein spezifisches Beispiel, welches mit dem Entwurf umsetzbar ist. Einzelne Algorithmen zur Erhebung, Transport und Verarbeitung sind frei wählbar und zur Laufzeit austauschbar.

In der realen Implementierung erhalten wir über mehrere Kanäle, wie beispielsweise Bus-Systeme oder externe Schnittstellen, unterschiedliche Arten von Daten. Diese Kanäle werden in aktive und passive Datenerzeuger unterteilt. In einem Versuchsaufbau kann es beliebig viele Erzeuger geben. Das zu konfigurierende Testsystem erhält beispielsweise Daten von analogen Input-Karten und Nachrichten über ein CAN-Bus-System.

Die zu entwickelnde Code-Basis soll eine Möglichkeit bieten, die erhaltenen Daten einheitlich zu verarbeiten. Eine individuelle Verarbeitung von Teildaten[6] ist nicht vorgesehen und sollte bereits bei der Erhebung der Daten erfolgt sein.

Nach der Verarbeitung sollen die Daten an einen festgelegten Server übermittelt werden. Da das *MQTT*-Protokoll besonders für instabile Netze mit geringer Bandbreite und hoher Latenz geeignet ist [11] und teilweise bereits als Standard für IoT-Anwendungen bezeichnet wird [12], wurde es als zentrales Kommunikationsprotokoll gewählt.

Die erhaltenen Daten werden in eine Datenbank überführt. Anforderung an die Datenbank ist die Unterstützung einer SQL-ähnlichen, relationalen Abfragesprache.

Zum Schluss soll eine vollständige Dokumentation angefertigt worden sein, damit das entwickelte Produkt ebenfalls zukünftig in Projekten eingesetzt werden kann, ohne eine lange Einarbeitungszeit erwarten zu müssen.

[6] z. B. ein sensorabhängiges Skalieren des Eingangspegels

3.3 Entwurf

3.3.1 Hardwareprototyp

In dem realen Testumfeld sind fünf Hardwaremodule vorgesehen siehe Abb. 1.

Der zu entwickelnde Prototyp ist ein vereinfachtes Modell des realen Testumfelds. Alle relevanten Steuereinheiten wurden zwar übernommen, jedoch werden sie zukünftig durch Erweiterungskarten ergänzt, welche über ein internes Bus-System mit dem RevPi kommunizieren. So wird es beispielsweise möglich sein, nativ den RevPi an einen *CAN*-Bus anzuschließen oder über analoge Eingänge verschiedene Sensoren anzuschließen, welche anliegende Gase messen können.

Auf Grund fehlender Hardware und begrenzter Zeit werden alle datenerzeugenden Schnittstellen virtualisiert beziehungsweise zu erhaltene Daten durch externe Quellen simuliert.

Die Hardwaremodule `1` und `2` erzeugen Testdaten und verteilen diese an weitere Module. Als zentrale Steuereinheit befindet sich in beiden Modulen ein RevPi Core 3+. Dieser RevPi erhält Daten über verschiedene Erweiterungsmodule. Das Hardwaremodul `1` verwendet das Erweiterungsmodul RevPi AIO. Das Hardwaremodul `2` verwendet das Erweiterungsmodul RevPi Con CAN. Für beide Module ist eine WLAN-Antenne vorgesehen, welche sie mit dem Mesh-basierten WLAN-Netzwerk verbindet. In dem Prototypen werden jedoch noch LAN-Kabel verwendet.

Das Hardwaremodul `3` stellt einen MQTT-Broker zur Verfügung. Dementsprechend muss im Betrieb der gewählte MQTT-Broker-Dienst Mosquitto dauerhaft laufen. Als Steuereinheit fungiert ebenfalls ein RevPi Core 3+.

Das Hardwaremodul `4` erhält die Daten und speichert diese in einer Datenbank ab. Da im Betrieb festgestellt wurde, dass der vorgesehene RevPi Core 3+ zu wenig Speicherplatz bietet, läuft der Dienst für die InfluxDB-Datenbank auf einem angeschlossen Laptop. Auf diesem Laptop befindet sich ebenfalls das Hardwaremodul `5` in Form einer Chronograf-Anwendung zur Datenvisualisierung.

Die für den Testaufbau verwendeten Hardwaremodule werden wie in Abb. 1 dargestellt über LAN-Kabel verbunden. Ein gestrichelter Rahmen signalisiert die Modulgrenzen. Der Datenfluss erfolgt von links nach rechts.

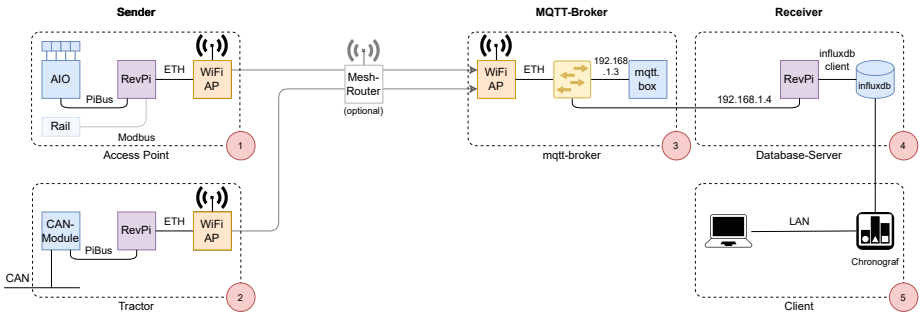

Abbildung 1: Skizze Hardwareaufbau.

3.3.2 Erhebung der Rohdaten

Alle aktiven und passiven datenerzeugenden Kanäle werden von einem *Client* verwaltet. Dieser läuft als ein `python3`-Skript jeweils auf der Steuereinheit eines Hardwaremoduls.

Aktive Datenerzeuger stellen eine Schnittstelle zur Verfügung, wodurch ein Client eigenständig das Erheben der Daten anstoßen kann. Diese Daten werden dem Client *synchron* mitgeteilt, wodurch dieser während der Erhebung der Daten blockiert.

Passive Datenerzeuger werden zu Beginn der Laufzeit gestartet und versenden eigenständig intervallgesteuert die zu erzeugenden Daten. Diese Daten werden dem Client *asynchron* mitgeteilt. Dadurch werden zwar keine weiteren Datenerzeuger blockiert, jedoch muss der Client Multithreading-fähig sein.

Die Kategorisierung von aktiven beziehungsweise passiven Datenerzeugern erfolgt meist durch die Schnittstelle der Sensoren. Falls die Sensoren ein asynchrones Abfragen unterstützen, werden die Datenerzeuger als passive Clients implementiert. Sollte das nicht der Fall sein, erhalten die Sensoren einen aktiven Datenerzeuger.

Ein Beispiel für aktive Datenerzeuger sind die analogen Input-Karten. An diese Karten werden analoge Sensoren angeschlossen, welche eine Spannung von 0 V bis 10 V liefern. Dieser Eingangspegel wird in einen repräsentativen numerischen Wert mit einer gedachten Einheit[7] umgerechnet und zur weiteren Verarbeitung freigegeben. Bisher ist ausschließlich ein Skalieren der Daten um einen festen Faktor gewünscht.

[7] z. B. Grad Celsius, ppm etc.

Ein Beispiel für einen passiven Datenerzeuger ist ein CAN-Bus-Knoten. Dieser empfängt ein fest definiertes Datenpaket von eigenständigen externen Sensoren zu einem beliebigen Zeitpunkt und zerteilt dieses in atomare Daten beziehungsweise interpretiert diese Daten.

Ein CAN-Datenpaket besteht in dem Versuchsumfeld beispielsweise aus Maschinendaten der Einschienenhängebahn. Das vollständige Datenpaket entspricht demnach Abb. 2.

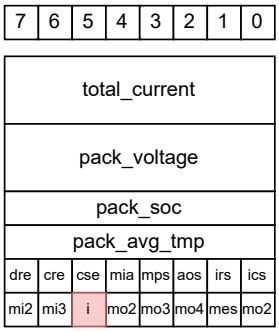

Abbildung 2: CAN-Nachricht.

Von außen betrachtet sieht die Kommunikation eines Hardwaremoduls zur Erhebung der Rohdaten wie in dem Sequenzdiagramm Abb. 3 zu sehen aus.

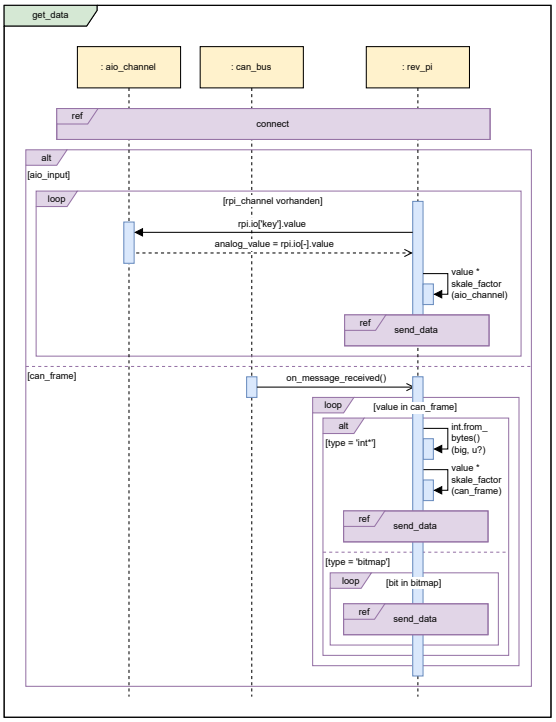

Abbildung 3: Sequenzdiagramm Erhebung der Rohdaten.

Da sich jeder Client vorab bei dem MQTT-Broker anmelden muss, wurde dieser Vorgang ebenfalls in einem referenzierten Sequenzdiagramm (Abb. 4) dargestellt.

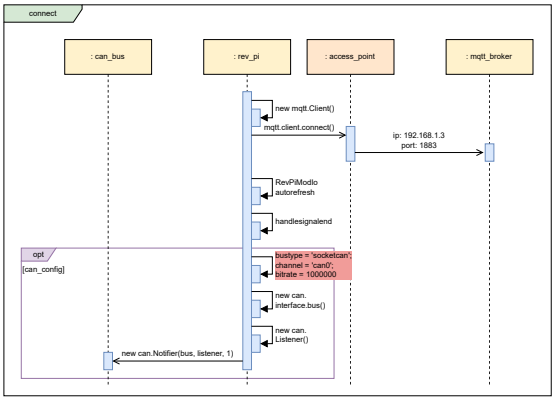

Abbildung 4: Sequenzdiagramm Erhebung der Rohdaten.

3.3.3 Verarbeitung der (Roh-) Daten

Die Verarbeiter befinden sich zwischen der Erzeugung der Rohdaten und dem Versenden. Für einen Erzeuger sehen die Verarbeiter wie „normale“ Sender aus. Für den Sender hingegen sehen die Verarbeiter wie „normale“ Erzeuger aus.

Es soll möglich sein eine beliebige Anzahl an nicht-reversiblen Verarbeitungsschritten vorzunehmen, welche einer festen Reihenfolge unterliegen.

Spätere Verarbeiter können weder erkennen, ob vorher eine Verarbeitung stattgefunden hat, noch diese Verarbeitung rückgängig machen. Von der technischen Machbarkeit abgesehen ist ein solcher Vorgang auch nicht erwünscht. Jeder Verarbeiter soll eine feste Aufgabe komplett abschließen und nicht von äußeren Gegebenheiten beeinflusst werden.

Wenn keine weitere Verarbeitung gewünscht ist, werden die Daten direkt von der Erzeugung zum Versenden freigegeben.

3.3.4 Transport der Daten

Da die Clients in unserem Anwendungsfall weder die Daten interpretieren, noch selber dauerhaft abspeichern sollen, müssen die Daten an einen zentralen Server versendet werden.

Die erhaltenen Daten werden beispielsweise erneut auf ein fest definiertes Intervall skaliert und an einen MQTT-Broker versendet, damit der Server die Daten dort abfragen kann. Eine MQTT-Nachricht besteht aus einem `topic` und einem UTF-8 codierten `payload`.

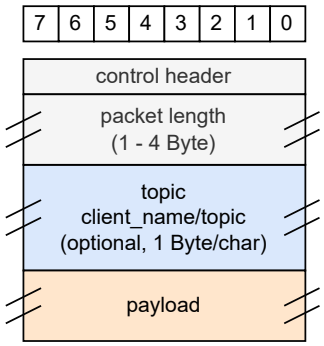

Abbildung 5: MQTT-Nachricht.

Als Namenskonvention wurde festgelegt, dass das `topic` den *Sender* und die *Art der Daten* durch einen `/` getrennt enthält. Sollte nun der Sender `ap1` eine `co`-Messung bereitstellen wollen, so wäre das topic `ap1/co` (Abb. 6).

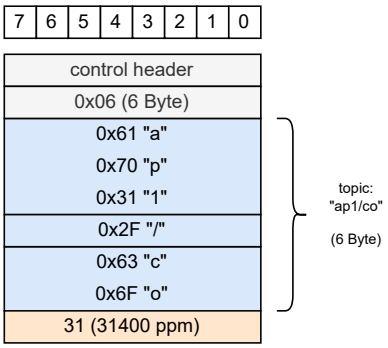

Abbildung 6: Beispiel-MQTT-Nachricht.

Um die Gesamtlänge einer MQTT-Nachricht zu errechnen (z. B. um das Netzwerk nicht übermäßig auszulasten), wird die Größe aller Pakete aufsummiert.

$$|\text{topic}| := |\text{sender}| + 1\,\text{B} + |\text{type}| + |\text{payload}|$$

$$|\text{mqtt}| := 1\,\text{B} + \left\lceil \frac{\ln\left(|\text{topic}|\right)}{8 \cdot \ln\left(2\right)} \right\rceil + |\text{topic}|$$

Die Kommunikation zwischen Sender und Server ist im Sequenzdiagramm Abb. 7 gegeben. Der MQTT-Broker wird in dem Diagramm nicht dargestellt, da das Datenpaket zwar aus Netzwerksicht an den MQTT-Broker gesendet wird, jedoch direkt von dem Server, der das `topic` abonniert hat, abgefangen wird. Somit befindet sich das Datenpaket aus Softwaresicht zu keinem Zeitpunkt bei dem MQTT-Broker.

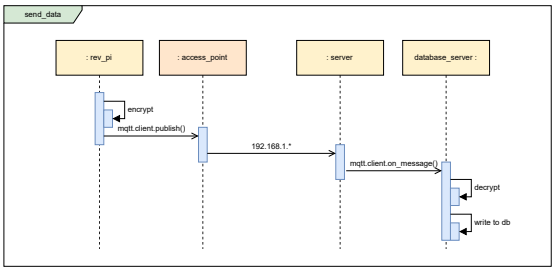

Abbildung 7: Sequenzdiagramm Versenden der Daten.

3.3.5 Speicherung der Daten

Die versendeten Daten wurden, wie beschrieben, einem Server über einen MQTT-Broker zur Verfügung gestellt. Zur weiteren Verarbeitung dieser Daten wird der Code, der für die folgend beschriebenen Aufgaben eingesetzt wird, als `python3`-Skript auf einem zentralen Server ausgeführt.

Aus Sicherheitsgründen werden für die meist verbreitetsten Abfragesprachen sogenannte *prepared Statements* implementiert, um die erhaltenen Daten in einer Datenbank abzuspeichern. Aufgrund der internen Verschlüsselung beziehungsweise Signierung des MQTT-Brokers werden zwar nur bekannte Clients akzeptiert und die Daten werden direkt von den Sensoren ohne User-Input weitergeleitet, jedoch ist es trotzdem notwendig, sich zusätzlich vor SQL-Injections zu schützen, da externe Geräte Daten besipielsweise über Bus-Systeme an Clients senden können und sich somit indirekt Zugriff zu der Datenbank verschaffen können.

Für den Prototypen wurde sich im Rahmen der Seminararbeit für eine *InfluxDB*-Zeitreihendatenbank entschieden. In dem auftraggebenden Forschungsprojekt *HEET II* werden dieser und alle folgenden Schritte von einem Projektpartner übernommen. Da jedoch für diese Seminararbeit und die Aufgabenbereiche des AMTs in *HEET II* eine Präsentation der Ergebnisse gewünscht ist, wurde temporär mit dieser Übergangslösung gearbeitet.

Eine Zeitreihendatenbank hat die Besonderheit, dass jede Entität als Primärschlüssel einen *Zeitstempel* hat. Dieser Zeitstempel ist beispielsweise der Zeitpunkt des Erhalts von Sensordaten. Zeitreihendatenbanken sind für Sensordaten besonders beliebt, da meist nur schwache Relationen zwischen Daten existieren und man ausschließlich einen Zeitpunkt und einen zugehörigen numerischen Wert abspeichern muss. Zum Unterscheiden verschiedener Messungen, kann man die Daten mit sogenannten *Metainformationen* versehen. [16]

Möchte der Server Informationen über den Anteil von `co`, welcher von `ap1` gemessen wurde, speichern, so legt InfluxDB eine neue Messreihe für CO-Messungen an. In diese Messreihe wird der gemessene Wert mit der Metainformation `ap=ap1` gespeichert. Wenn `ap2` ebenfalls eine CO-Messung durchführt, wird der gemessene Wert ebenfalls in diese Messreihe gespeichert. So können die Messwerte der verschiedenen Standorte besser miteinander verglichen werden.

Wenn einen ausschließlich die Messwerte eines bestimmten Clients[8] interessieren, kann man die Messreihen mit einem `SELECT`-Statement filtern, sodass nur Messwerte mit der Metainformation `ap=ap1` angezeigt werden. Anschließend ist es möglich über verschiedene Messreihen hinweg Tabellen mithilfe der Metainformationen durch `JOIN`-Statements zusammenzufügen.

[8] z. B. ap1

3.3.6 Visualisierung der Daten

Die erhaltenen Messreihen können einzeln abgefragt werden. Alternativ kann man die Spalte `value` von verschiedenen Messreihen gebündelt abfragen.

Daraus resultierende Tabellen können von einer Weboberfläche in einen Graphen umgewandelt werden. Sollten sich in der Tabelle mehrere Entitäten mit unterschiedlichen Metainformationen befinden, erstellt die Weboberfläche einen weiteren Graphen für jeden Client.

4 Implementierung

4.1 Verwendete Softwaremuster

Der zu erstellende Code orientiert sich an den Konzepten einer Daten-Pipeline mit dem Decorator-Pattern als konkretes Softwaremuster.

Pipelines ermöglichen es, einem Programmdaten zwischen Verarbeitungsschritten weiterzuleiten [13]. Da sich das Konzept dieses Projekts intuitiv in eindeutige Verarbeitungsschritte unterteilen lässt, wird dieses Muster verwendet.

Da es sich bei Pipelines jedoch um ein Architekturmuster handelt, entscheidet die Wahl Pipelines zu nutzen nicht über die Art der konkreten Implementierung, sondern eher über die Art, wie Daten und einzelne Operationen betrachtet werden.

Das *Decorator-Pattern* ist ein Strukturmuster, welches ebenfalls geeignet für diesen Anwendungsfall ist. Es dient dazu, flexibel Klassen zu erweitern, ohne für jede Kombination an Funktionalitäten eine weitere Unterklasse erstellen zu müssen. [4]

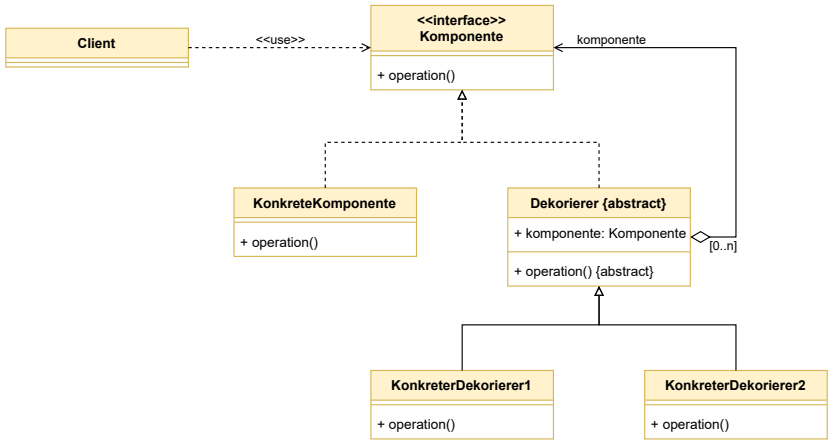

Abbildung 8: Dekorierer-Muster (Abstrakt).

Für den vorliegenden Anwendungsfall kann das vorhandene Konzept auf dieses abstrakte Klassendiagramm angewendet werden. Die *konkrete Komponente* wäre demnach die Klasse, in der Daten erzeugt werden. Ein *konkreter Dekorierer* verarbeitet die Daten, welche von der `operation` der konkreten Komponente erzeugt wurde, sieht jedoch für weitere Klassen ebenfalls wie eine Komponente aus.[9]

Jeder Client hat eine eigene Konfigurationsdatei hinterlegt. Diese ist unterteilt in einen allgemeingültigen Teil, welcher für jeden Client identisch ist, und einen spezifischen Teil.

In dem allgemeingültigen Teil befinden sich Informationen über beispielsweise die IP-Adresse und den TCP/UDP-Port des MQTT-Brokers.

Der spezifische Teil beinhaltet Informationen über beispielsweise analoge Input-Karten oder einen anliegenden CAN-Bus. Ebenfalls ist dort der Name des Clients zu finden, über den dieser am Server eindeutig identifizierbar ist.

4.2 Entwicklungsumgebung

Die Entwicklung erfolgt in einer „Visual Studio Code"-Umgebung (VSCode) mit verschiedenen Python-Erweiterung. Die Speicherung der Daten erfolgt in einem GitLab-Repository zur Versionsverwaltung und cloudbasierten Sicherung.

Zum Ausführen des Codes wird auf den RevPi's ein VSCode-Server aufgebaut, mit denen sich der Laptop, auf dem die Software entwickelt wird, verbindet. Eine Authentifizierung des Laptops erfolgt über OpenSSH-Keys, welche auf den jeweilig beteiligten Hardwarekomponenten gespeichert werden. Ein Python-Prozess führt demnach die Skripte auf den RevPi's aus und wird von fest definierten Cronjobs gestartet.

4.3 Erhebung der Rohdaten

Die Erhebung der Rohdaten erfolgt in der Klasse `Anwendung`. Diese ist ein spezifizierter Dekorierer, welcher ausschließlich die letzte abschließende Komponente sein kann.

Der Konstruktor wird genutzt, um Attribute der Klasse zu setzen. Wenn keine individuellen Algorithmen genutzt werden sollen, kann man die übergebenen Parameter an dem Konstruktor der `super`-Klasse weiterleiten.

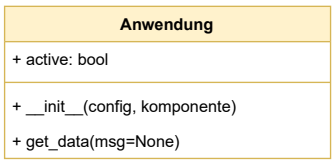

Abbildung 9: Klasse Anwendung (Client).

[9] Beispiele folgen in den jeweiligen Verarbeitungsschritten

Die Methode `get_data` wird für beliebige Clients zur Verfügung gestellt. Optional kann man ein beliebiges Objekt mitgeben, welches zur Erzeugung von Messungen genutzt wird.

Intern erstellt eine Anwendung beliebig viele Messwerte als Paar mit jeweils einem Namen beziehungsweise einem topic und einem Wert (value). Diese werden zur weiteren Verarbeitung an die private Hilfsmethode `operation` weitergegeben. Da diese Methode keine weiteren Veränderungen durchführt, werden die Nachrichten direkt an den nächsten Dekorierer, welcher als Attribut in dem Objekt gespeichert ist, weitergegeben.

Die Klasse `AIO` speichert sich ausschließlich den Teil der `config` zusätzlich im Konstruktor ab, welcher die analogen Eingänge beschreibt. Das Abfragen von Daten erfolgt von einem Client, da es sich um eine aktive Klasse hält.

Die Klasse `CAN` erstellt im Konstruktor einen `Notifier`, um über erhaltene Nachrichten informiert zu werden, und übergibt diesem die Methode `get_data` um erhaltene Nachrichten automatisch zu verarbeiten. Des Weiteren speichert er sich die Struktur eines *CAN-Frames* ab.

Ein CAN-Frame definiert, wie erhaltene Bytes interpretiert werden sollen. Um das Frame korrekt aufzuteilen, benötigt man die Länge der einzelnen Typen. Diese Teile werden dann in den angegeben Typen umgewandelt und mit dem angegeben Namen als topic versendet.

Eine Konfiguration eines CAN-Frames sieht beispielsweise wie in Abb. 10 verkürzt dargestellt aus:

```
"can_frame": [
    {
        "type": "int",
        "length": 2,
        "signed": true,
        "name": "total_current",
        "factor": 1
    },
    "_comment": "...",
    {
        "type": "bitmap",
        "length": 1,
        "bitmap": [
            "multi_purpose_input2_signal_status",
            "multi_purpose_input3_signal_status",
            "ignore",
            "multi_purpose_output2_signal_status",
            "multi_purpose_output3_signal_status",
            "multi_purpose_output4_signal_status",
            "multi_purpose_enable_signal_status",
            "multi_purpose_output2_signal_status"
        ]
    }
]
```

Abbildung 10: CAN-Frame-Konfiguration.

4.4 Verarbeitung der (Roh-) Daten

Die Verarbeitung der erzeugten Daten erfolgt in abgeleiteten Klassen der abstrakten Klasse `Dekorierer`.

Im Konstruktor wird größtenteils ausschließlich die folgende Komponente gesetzt. Da die meisten Dekorierer eher strukturelle Änderungen vornehmen, ist meist keine Konfiguration notwendig, da diese Dekorierer allgemeingültig sein sollen.

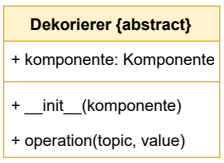

Abbildung 11: Abstrakte Klasse Dekorierer (Client).

Die Methode `operation` erhält die erzeugten Daten, verarbeitet diese und leitet die an die nächste Komponente weiter.

Die Klasse `AVG` puffert alle gemessenen Werte in einem `dictionary` und versendet alle Werte gebündelt nach Ablauf eines Timers. Dazu werden *numpy*-Arrays genutzt, um numpy funktionen, wie `np.average` nutzen zu können.

Die Klasse `DEBUG` logt alle erhaltenen Nachrichten über eine festgelegte Funktion. Die Auswahl der Funktion erfolgt über das *Strategy-Pattern*. Standardmäßig wird `print` verwendet.

4.5 Transport der Daten

Der Transport beziehungsweise die finale Verarbeitung der Daten erfolgt in abgeleiteten Klassen der abstrakten Klasse `Komponente`.

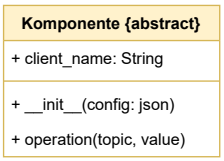

Abbildung 12: Abstrakte Klasse Komponente (Client).

Der Konstruktor wird erneut genutzt, um Instanzen der Klasse zu konfigurieren.

In der Methode `operation` werden die Daten ein letztes Mal auf dem RevPi verarbeitet. Da eine konkrete Komponente keine weiteren Komponenten speichert, wird die Methode `operation` genutzt, um die Daten in eine Datei zu schreiben oder über ein weiteres Protokoll an andere Netzwerkgeräte weiterzuleiten.

Die Klasse `MQTT` fügt dem `topic` den Namen des Access Points an und versendet die Werte mit dem neuen Namen über das MQTT-Protokoll.

Je „weiter entfernt“ die Komponente von einer Anwendung ist, desto generalisierter ist sie. Im Gegensatz dazu sind „nahe“ Klassen spezialisierter. Dies sieht man beispielsweise an der Vererbungshierarchie, wonach `Dekorierer` von `Komponente` abgeleitet wird. `Anwendung` ist eine Spezialisierung der Klasse `Dekorierer`. Durch das Verarbeiten der Rohdaten werden diese ebenfalls immer weiter abstrahiert, wodurch ein maßgeschneiderter Umgang mit den Daten von einem späteren Dekorierer unter Umständen nicht weiter möglich ist. Komponenten sind von dieser Regel teilweise ausgeschlossen, da die Daten meist insofern verändert werden, dass sie für die finale Komponente am besten geeignet sind.

4.6 Speicherung der Daten

Da wir uns nicht mehr auf dem RevPi (Client) befinden, sondern auf einem beliebigen Server, durchlaufen wir die soeben beschrieben Schritte in der umgekehrten Reihenfolge, um die Informationen der verarbeiteten Rohdaten bestmöglich zu extrahieren. Die Klassenstruktur ist analog zu der eben beschriebenen Klassenstruktur des Clients, deswegen wurde ein beispielhafter Durchlauf von Daten hier zusammengefasst.

Da Softwarekomponenten auf dem Server selten ausgetauscht werden müssen, erfolgt die Implementierung meist grundlegender, als die stark modulare Implementierung eines Clients.

Die Struktur eines Dekorierer-Patterns ist jedoch trotzdem sinnvoll, um vorhandene Funktionalitäten erneut nutzen zu können, und Funktionalitäten im Betrieb einfach zu ergänzen, beziehungsweise vorhandene Funktionalitäten anzupassen.

Die Klasse `Anwendung` entspricht der gleichnamigen Klasse des Clients. Die Unterteilung in aktive beziehungsweise passive Anwendungen entfällt jedoch, da wir nur asynchrone Schnittstellen zu weiteren Hardwaremodulen haben. Aus dem selbem Grund wird keine `get_data`-Methode mehr benötigt.

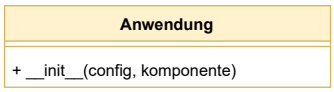

Abbildung 13: Klasse Anwendung (Server).

Die Klasse `MQTT` ist das exakte Gegenstück der Komponente `MQTT` des Clients. Es empfängt die dort versendeten Nachrichten und leitet sie zur nächsten Komponente weiter.

Die Klasse `Dekorierer` wird genutzt um erhaltene Daten serverseitig zu filtern, zu verarbeiten oder zu protokollieren.

Dazu wurde die Klase `DEBUG` von der Client-Seite übernommen. Durch die symmetrische Struktur von Server und Client, mussten keine Anpassungen an der Klasse vorgenommen werden.

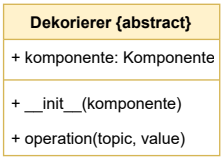

Abbildung 14: Klasse Dekorierer (Server).

Weiterhin können die Dekorierer genutzt werden, um vorab bei Messungen mit beispielsweise einer zu hohen Varianz oder bei fehlenden Messungen zu warnen. Meistens erfolgen diese Überprüfungen jedoch nach dem Speichern in eine Datenbank in einem späteren Schritt. Dies ist im Forschungsprojekt *HEET II* die Aufgabe von Projektpartnern.

Die Klasse `Komponente` ist ebenfalls auf dem Server der finale Verarbeitungsschritt. Jedoch werden die Daten nicht an weitere Hardwaremodule weitergeleitet, sondern auf dem Server dauerhaft gespeichert.

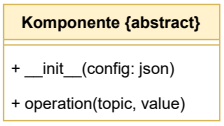

Abbildung 15: Klasse Komponente (Server).

Die Klasse `DB` bietet aus diesem Grund eine API, um Messwerte in eine InfluxDB-Datenbank zu speisen. Eine Abwandlung der InfluxQL-Befehle in SQL-Befehle ist ebenfalls möglich.

4.7 Visualisierung der Daten

Die Auswertung der Daten erfolgt manuell über Chronograf, die grafische Oberfläche von InfluxDB.

Tabellen sowie einfache Graphen lassen sich mit Hilfe von InfluxQL-Abfragen erstellen. InfluxQL ist eine *SQL*-ähnliche Abfragesprache der InfluxDB.

Beispielsweise lassen sich mit folgender Abfrage alle analogen Input-Werte der letzten Stunde anzeigen.

```
USE "heet_2";
SELECT
    "value"
FROM
    "co", "ch4", "o2", "t", "rf"
WHERE
    time > now()-1h;
```

Abbildung 16: Beispiel InfluxQL.

Chronograf zeigt standardmäßig die abgefragten Messwerte als Tabelle an. Ein Umstellen auf den Graphen ist jedoch möglich, wenn ein eindimensionaler Wert abgefragt wird.

Abbildung 17: Beispiel Chronograf.

Um die Seite nicht manuell aktualisieren zu müssen, bietet Chronograf die Möglichkeit, eigenständig den Graphen dauerhaft zu aktualisieren.

4.8 Gesamtbild der Implementierung

Wenn die einzeln beschriebenen Klassen in das abstrakte Klassendiagramm des Dekorierer-Musters überführt werden, erhält man eine grobe Übersicht über die Implementierung. Durch konkrete Besonderheiten in der Implementierung weicht das Diagramm zwar durch eine weitere abstrakte Klasse (`Anwendung`) von der ursprünglich Version des Musters ab, folgt jedoch der selben Idee einer dynamischen Unterklassenbildung (Abb. 18 und 19).

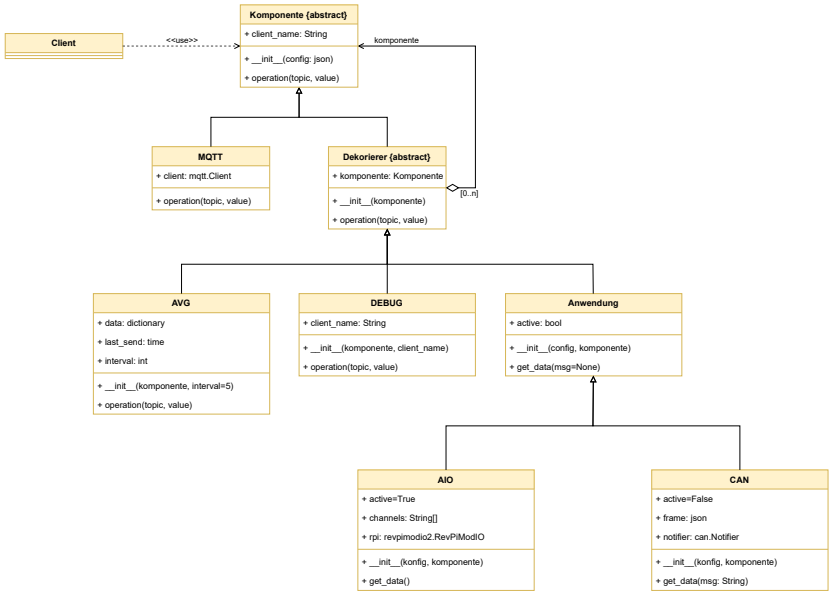

Abbildung 18: Klassendiagramm des Clients.

Die selbe Struktur ist auf der Seite des Servers zu finden. Jedoch ist dieses Klassendiagramm, auf Grund noch deutlich weniger benötigter Unterklassen, merklich dünner besetzt (Abb. 19).

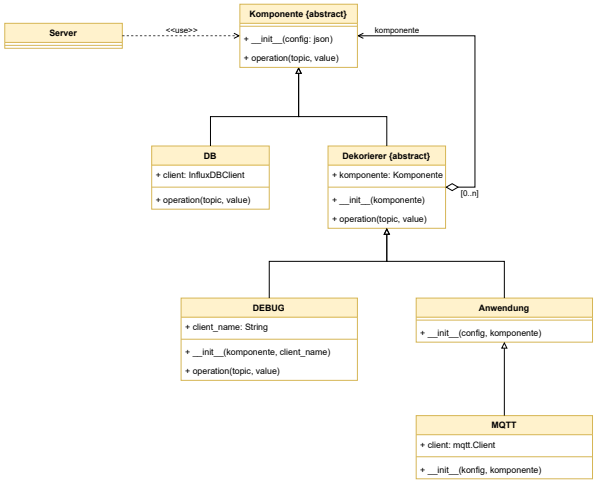

Abbildung 19: Klassendiagramm des Servers.

5 Fazit

5.1 Konklusion

Im Rahmen dieser Seminararbeit wurde die Grundlage für eine strukturierte Kommunikation zwischen verschiedenen Hardwaremodulen geschaffen. Ein schnelles Ausrollen und Konfigurieren neuer Systeme sowie ein dynamisches Anpassen der bestehenden Infrastruktur ist möglich.

Alle Kernanforderungen, welche zu Beginn dieses Projekts definiert wurden, wurden erfolgreich erfüllt. Alle weiteren optionalen Anforderungen wurden bei der Modellierung berücksichtigt und teilweise ebenfalls erfolgreich abgeschlossen.

5.2 Ausblick

Die bestehende Klassenstruktur soll um weitere Funktionalitäten ergänzt werden. Beispielsweise werden weitere Hardwaremodule über neue Bus-Systeme mit dem Client Informationen austauschen. Zusätzlich soll eine Kommunikation zwischen Clients stattfinden, wodurch alle Systeme autonom arbeiten können.

Zeitgleich ein realer Versuchsaufbau eingerichtet und die Installation der im Rahmen dieser Seminararbeit erstellten Software betreut. Eine Einarbeitung weiterer Mitarbeiter*innen wird ebenfalls stattfinden. Diese sollen teilweise den Code nutzen und bearbeiten können, sollten neue Anforderungen an das System gestellt werden.

Die reale Umgebung dient als Testumfeld für das gesamte Projekt *HEET II*. Sie soll genutzt werden, um verschiedene Testdaten zu generieren und zu zertifizieren, dass eine kabellose Energie- und Datenübertragung eine sicherere und prozesseffizientere Alternative zu bestehenden Konzepten ist.

Literatur

[1] *A Portable Environmental Data-Monitoring System for Air Hazard Evaluation in Deep Underground Mines*. MDPI. URL: https://www.mdpi.com/1996-1073/13/23/6331.

[2] *An Internet of Things System for Underground Mine Air Quality Pollutant Prediction Based on Azure Machine Learning*. MDPI. URL: https://www.mdpi.com/1424-8220/18/4/930.

[3] *Assessment of the Effects of Global Digitalization Trends on Sustainability in Mining*. Bundesanstalt für Geowissenschaften und Rohstoffe. URL: https://www.bgr.bund.de/EN/Themen/Min_rohstoffe/Downloads/digitalization_mining_dustainability_part_I_en.pdf?__blob=publicationFile (besucht am 13. 12. 2022).

[4] *Decorator Pattern: Das Muster für dynamische Klassenerweiterungen*. IONOS. URL: https://www.ionos.de/digitalguide/websites/web-entwicklung/was-ist-das-decorator-pattern/ (besucht am 12. 12. 2022).

[5] *Designing a Monitoring System to Observe the Innovative Single-Wire and Wireless Energy Transmitting Systems in Explosive Areas of Underground Mines*. MDPI. URL: https://publications.rwth-aachen.de/record/840187/files/840187.pdf.

[6] *Digitalisierung im Bergbau – Industrie 4.0*. Mining Report. URL: https://mining-report.de/digitalisierung-im-bergbau-industrie-4-0/ (besucht am 16. 11. 2022).

[7] *DMT – „Digital Mining Transformation" für die Rohstoffindustrie*. Mining Report. URL: https://mining-report.de/dmt-digital-mining-transformation-fuer-die-rohstoffindustrie/ (besucht am 09. 11. 2022).

[8] *Einschienenhängebahn - Schienensysteme*. Neuhäuser. URL: https://neuhaeuser.com/bergbau25/index.php/de/produkte/einschienhaengebahn (besucht am 12. 12. 2022).

[9] *Elektrohydraulische Einschienenhängebahn*. Neuhäuser. URL: https://neuhaeuser-gmbh.de/bergbau25/index.php/de/elektr-hydr-ehb (besucht am 12. 12. 2022).

[10] *Kategorien der Gasausbrüche im Steinkohlenbergbau aus praktischer Sicht*. Mining Report. URL: https://mining-report.de/kategorien-der-gasausbrueche-im-steinkohlenbergbau-aus-praktischer-sicht/ (besucht am 16. 11. 2022).

[11] *MQTT - Message Queue Telemetry Transport*. Elektronik Kompendium. URL: https://www.elektronik-kompendium.de/sites/net/2204051.htm (besucht am 09. 12. 2022).

[12] *MQTT: The Standard for IoT Messaging*. MQTT. URL: https://mqtt.org/ (besucht am 09. 12. 2022).

[13] *Pipeline Architecture*. cs.sjsu. URL: http://www.cs.sjsu.edu/~pearce/modules/patterns/distArch/pipeline.htm (besucht am 12. 12. 2022).

[14] *Stromsystem und Oberleitung*. Trackopedia. URL: https://www.trackopedia.com/lexikon/infrastruktur/stromsystem-und-oberleitung (besucht am 12. 12. 2022).

[15] *Ventilation on Demand – Bedarfsgerechte Wetterführung*. Mining Report. URL: https://mining-report.de/ventilation-on-demand-bedarfsgerechte-wetterfuehrung/ (besucht am 09. 11. 2022).

[16] *Zeitreihendatenbank (Time Series Database, TSDB)*. ComputerWeekly. URL: https://www.computerweekly.com/de/definition/Zeitreihendatenbank-Time-Series-Database-TSDB (besucht am 12. 12. 2022).