HAW HAMBURG

BACHELOR THESIS
Andreas Lindhorst

Plug and Produce Agents: Entwicklung eines verteilten Agent-Handling-System basierend auf Thread-Netzwerken

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Andreas Lindhorst

Plug and Produce Agents: Entwicklung eines verteilten Agent-Handling-System basierend auf Thread-Netzwerken

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Jan Sudeikat
Zweitgutachter: Prof. Dr. Michael Köhler-Bußmeier

Eingereicht am: 23. April 2024

Andreas Lindhorst

Thema der Arbeit

Plug and Produce Agents: Entwicklung eines verteilten Agent-Handling-System basierend auf Thread-Netzwerken

Stichworte

Multi-Agenten-Systeme, Plug and Produce, OPC UA, Adaptive Systeme, Thread-Netzwerkprotokoll

Kurzzusammenfassung

Die vorliegende Arbeit beschreibt die Entwicklung eines agentenbasierten Plug and Produce-Systems sowie des verteilten Agent-Handling-Systems, das zur Kommunikation das Thread-Netzwerkprotokoll verwendet. Es werden Motivation und Zielsetzung für die Systementwicklung erläutert und Grundlagen zu Plug and Produce, dem Thread-Netzwerkprotokoll, OPC UA und Multiagentensystemen unter Berücksichtigung des aktuellen Technologiestandes präsentiert. Eine Anforderungsanalyse wird durchgeführt, gefolgt von der Beschreibung des Designs der erforderlichen Komponenten und des OPC UA Datenmodells mit Diskussionen zu den getroffenen Entscheidungen. Die Umsetzung wird durch Implementierungsdetails eines Prototypen sowie der Benutzeroberfläche für die Interaktion mit dem System dargestellt. Zur Evaluation wird ein Szenario verwendet, bei dem eine nicht optimale Ausgangskonfiguration des Systems durch den Anschluss neuer Hardware in eine optimale Konfiguration überführt wird. Wobei alle Ausgaben und Zeiten dokumentiert werden. Performanztests werden definiert und durchgeführt. Potenzielle Weiterentwicklungsmöglichkeiten im Bereich der Thread-Netzwerkprotokoll Implementierung, OPC UA und des verwendeten Nebenläufigkeitsframeworks werden diskutiert, ebenso wie die Generalisierung der Plug and Produce Agents und weitere Anwendungen des Systems außerhalb des Evaluationsszenarios. Die Analyse der Evaluationsergebnisse bestätigt die erfolgreiche Implementierung und identifiziert Probleme im Nachrichtenaufkommen und mit G-Code-Dateien größer als 2 MB Dateigröße. Für die identifizierten Probleme werden Lösungsansätze definiert und diskutiert. Zusammenfassend zeigt sich, dass das entwickelte System den Anforderungen entspricht und ein funktionsfähiges Plug and Produce-System darstellt, das auf dem Thread-Netzwerkprotokoll basiert, keinen Single-Point-of-Failure aufweist und die strukturierte Informationsübermittlung der Hardwarekomponenten an den OPC UA Server ermöglicht.

Andreas Lindhorst

Title of Thesis

Plug and Produce Agents: Development of a distributed Agent-Handling-System based on Thread-Networks

Keywords

Multi-Agent-Systems, Plug and Produce, OPC UA, Adaptive Systems, Thread-Networkprotocol

Abstract

This paper describes the development of an agent-based Plug-and-Produce system with a distributed Agent Handling System that utilizes the Thread network protocol for communication. It outlines the motivation and objectives for system development and presents fundamentals of Plug-and-Produce, the Thread network protocol, OPC UA, and multi-agent systems in line with current technological advancements. An analysis of requirements is conducted, followed by a description of the design of necessary components and the OPC UA data model, including discussions on decision-making processes. Implementation details, such as those of a prototype and the user interface for system interaction, are provided. The implemented system is evaluated using a scenario wherein it transitions from a suboptimal initial configuration to an optimal one through the addition of new hardware, with all outputs and timings documented. Performance tests are defined and executed. Potential avenues for further development concerning the Thread network protocol implementation, OPC UA, and the concurrency framework used are discussed, as well as the generalization of Plug and Produce Agents and additional applications beyond the evaluation scenario. An analysis of evaluation results confirms successful implementation while identifying issues with message traffic and G-Code files larger than 2 MB, for which solutions are proposed. In conclusion, the paper demonstrates that the developed system meets requirements and constitutes a functional Plug-and-Produce system based on the Thread network protocol, free from single points of failure, and facilitating structured information exchange of hardware components with the OPC UA server.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

AHS Agent Handling System.

DAHS Distributed Agent Handling System.

FDM Fused Deposition Modeling.

GPIO General-purpose input/output.

IDE Integrated Development Environment.

MAS Multi Agenten System.

OPC Open Platform Communication.

OPC UA Open Platform Communication Unified Architecture.

PUB/SUB publish-subscribe.

PWM Pulsweitenmodulation.

SBC Single Board Computer.

SLA Stereolithografie.

SLS Selective Laser Sintering.

SoC System-on-a-Chip.

VOC Volatile Organic Compounds (Flüchtige organische Verbindungen).

WSL Windows Subsystem for Linux.

1 Einleitung

Im Kontext von Industrie 4.0 werden vermehrt Multi-Agenten-Systeme (MAS) eingesetzt [33], um eine dynamische Rekonfiguration von Industrieanlagen/Maschinenanlagen zu realisieren. Diese Systeme werden mit einer zentralen Architektur und einer festen Anzahl von angeschlossener Sensorik/Aktorik geplant. Die Agentensteuerung und Verteilung wird durch die zentrale Architektur übernommen und basiert vorwiegend auf der festen Sensorik und Aktorik. Dadurch ist ein schneller Aufbau und die Erweiterung um zusätzliche Sensoren und Aktoren eines MAS limitiert und mit vielen Vorkonfigurationsschritten verbunden. Die in dieser Arbeit realisierte MAS-Anwendung ist dezentral und dynamisch konfigurierbar aufgebaut, um ein MAS mit Plug and Produce Agents zu realisieren. Dadurch ist ein Aufbau eines MAS mit wenigen Vorkonfigurationen möglich. Es wird ermöglicht, ein MAS dynamisch in der Größe zu skalieren, indem neue dezentrale Knoten hinzugefügt werden. Diese Knoten können während der Laufzeit um weitere Sensoren und Aktoren erweitert werden, die automatisch durch das entwickelte DAHS in das laufende MAS eingliedert werden.

1.1 Motivation

Die Entwicklung der Plug and Produce Agents und dem Distributed-Agent-Handling-System (DAHS) ist maßgeblich durch das Paper "Identification of resources and parts in a Plug and Produce system using OPC UA" [10] motiviert. In diesem wurde eine Plug-and-Produce Anwendung konzipiert um eine dynamische Instanziierung von Agenten, basierend auf angeschlossenen Komponenten, mithilfe eines AHS (Agent-Handling-System) zu realisieren. Dabei wurde ein zentralisierter Ansatz gewählt, in dem eine Datenbank die zur Verfügung stehenden Agentenkonfigurationen gesammelt anbietet. Dadurch ist die Architektur und Entwicklung eines verteilten Agent-Handling-Systems bzw. DAHS motiviert. Es soll zeigen, dass es die Hauptschwäche des zentralisierten Ansatzes, den Single-Point-of-Failure, lösen kann, jedoch durch die Probleme eines verteilten Systems nicht

beeinträchtigt wird. Allgemein bietet die Idee des AHS Vorteile für (semi-)industrielle Anwendungen, die eine weitere Motivation für das DAHS darstellen. Sie ermöglichen die dynamische Systemneukonfiguration durch eine nahtlose Integration neuer Komponenten und verbessern dabei die Effizienz durch die Automatisierung und Optimierung der Agenteninstanziierung. Sie erhöhen die Widerstandsfähigkeit des Gesamtsystems durch Fehlererkennungs und -behandlungs-Mechanismen und fördern die Interoperabilität zwischen Systemen durch eine Vereinheitlichung beim Umgang mit Agenten. Die Einführung von AHS in MAS ermöglicht eine Abstraktion der vielfältigen Agentenverwaltungsmechanismen in eine Komponente, wodurch die Komplexität reduziert und die Wartbarkeit erhöht wird.

1.2 Zielsetzung

Die folgenden Fragestellungen bzw. Herausforderungen ergeben sich aus dem in der Motivation zitierten System und der Definition des Evaluationsszenarios im Abschnitt 5.3.2. Durch die Erstellung und Evaluation des Systems soll speziell folgendes beantwortet bzw. gelöst werden:

- Gestaltung des Datenmodells: Es soll ein OPC UA basiertes Datenmodell erstellt werden, mit einem hierarchischen OPC UA Organisationsmodell, nach [68], welches die zugehörigen Agenten eines System-Knotens in der Hierarchie gruppiert und einem übergeordneten Agenten zuweist. Dabei soll gezeigt werden, wie ein Datenmodell die Darstellung des Systems von außen, und die Verwaltung der Agenten im System, vereinfacht.
- Konfiguration der Agenten: Zur Konfiguration der verschiedenen Agententypen muss jeder DAHS-Knoten jede notwendige Konfiguration vorhalten bzw. hinzufügen können. Es muss ein Mechanismus entwickelt werden, der das Hinzufügen und Abrufen von Konfigurationen ermöglicht. Dabei soll gezeigt werden, dass bei der Instanziierung eines neuen Agententyps dynamisch diese Konfiguration abgerufen werden kann, falls diese noch nicht vorhanden ist.
- Erkennung neuer Hardware: Um den Aspekt des Plug and Produce zu realisieren, muss eine Methode zur Erkennung von neu angeschlossener Hardware entwickelt werden, welche anhand der vorhandenen Informationen aus den Agentenkonfigurationen den richtigen Agenten zur angeschlossenen Hardware instanziiert. Dabei

soll gezeigt werden, dass eine weitgehend Bus-unabhängige Hardwareerkennung und Agenteninstanziierung möglich ist und die Anforderungen an ein Plug and Produce-System erfüllt.

- Verteilte Arbeitsweise: Durch den Verzicht auf zentralisierte Speicher, muss eine Synchronisierung der System-Informationen erfolgen. Dies führt dazu, dass veraltete Informationen oder verlorene Nachrichten im System zu unvorhergesehenen Zuständen führen können. Durch die Konstruktion des DAHS mit Synchronisierungsmechanismen und Fehlerbehebungsmechanismen soll gezeigt werden, dass die verteilte Arbeitsweise kein Nachteil zu einem zentralisierten System darstellt.

Der Einsatz des Thread-Protokolls (siehe Definition in Abschnitt 2.2) wurde durch den technischen Fortschritt in der Netzwerktechnik motiviert, da es durch die Mesh-Architektur und die diversen automatischen Ausfall-Mitigations-Mechanismen die geplante Funktionsweise des DAHS unterstützt.

1.3 Inhaltlicher Aufbau der Arbeit

In dieser Arbeit werden der Stand der Technik, die notwendigen Komponenten zur Realisierung sowie die Softwareanwendung erläutert und anhand eines Szenarios aus dem semi-industriellen Umfeld evaluiert und analysiert. In Kapitel 1 wird eine Einleitung zur Thematik gegeben und es werden die Ziele des entwickelten Systems erläutert und was diese Ziele motiviert. In Kapitel 2 werden die notwendigen Grundlagen beschrieben, welche Informationen über die verwendeten Technologien und Architekturen geben. In Kapitel 3 werden die Anforderungen formal analysiert und die getroffenen Designentscheidungen erläutert. In Kapitel 4 wird die praktische Umsetzung beschrieben. Es wird auf genutzte Algorithmen, Python-Packages und die Entwicklungsumgebung sowie auf die verwendete Hardware eingegangen. In Kapitel 5 wird das Evaluationsszenario und die darauf basierenden Evaluationen beschrieben. Des Weiteren wird auf die Testumgebung und die Testdaten eingegangen. Zuletzt wird die Evaluationsdurchführung beschrieben und die Ergebnisse dargestellt. In Kapitel 6 wird zuerst auf die Weiterentwicklungsmöglichkeiten des DAHS und der Plug and Produce Agents eingegangen, sowie Verbesserungsmöglichkeiten aufgezeigt, welche durch die Entwicklung und Evaluation ersichtlich geworden sind. Zuletzt wird auf die Ergebnisse der Evaluation eingegangen und darauf basierend ein Fazit formuliert. In Kapitel A sind Informationen zu den einzelnen Agenten-Konfigurationen zu finden, sowie größere Grafiken.

2 Grundlagen

In diesem Kapitel werden die Begriffe und Technologien, welche in der Einleitung referenziert wurden, aufgegriffen und ein Überblick über diese dargestellt.

2.1 Plug and Produce

Unter Plug and Produce verstehen sich, im Kontext von Industrie 4.0, Systeme die eine automatische Selbstkonfiguration, -optimierung und -diagnose durchführen können. Dies findet basierend auf verbundenen, neu hinzukommenden und verschwindenden Komponenten statt. Die Hauptidee dabei ist, das erforderliche Expertenwissen zur Konfiguration, Optimierung und der Fehlerdiagnose von Industriesystemen aus dem menschlichen Bereich in die Automatisierung zu verlagern [17]. Darüber hinaus soll es auch intelligente Fabriken ermöglichen, welche die Konfiguration der Maschinen nach Bedarf der Produktion und wechselnder Produktionsspezifikationen anpassen. [35]

2.1.1 Stand der Technik

Schon im Jahr 2000 wurde ein System zum "Plug and Produce" an der University of Tokyo und Tsukuba University entwickelt [7]. In dem entwickelten System wurde eine Holon-basierte-Architektur genutzt, die es ermöglicht neue Produktionszellen ohne Vorkonfiguration hinzuzufügen. Das System nutzt ein Broadcast-Netzwerk zur Kommunikation der Komponenten und benötigt einen System Manager, der das zentrale Management wie z.B. DHCP übernimmt. Das System muss zum Entfernen von Komponenten heruntergefahren werden, benötigt danach aber keine weitere Konfiguration. In [61] wurde ein auf Agenten basiertes Framework vorgestellt, um Plug and Produce zu ermöglichen. Auch in diesem vorgestellten System wird ein zentralisierter Ansatz verfolgt. Die verschiedenen Konfigurationen der Komponenten werden in einer globalen Datenbank vorgehalten und beim Hinzukommen einer Komponente von dort abgerufen.

In [55] wurde ein generisches Plug and Produce System mit OPC UA Skills entwickelt und evaluiert. Dieses System implementiert eine gemischte Architektur. Die Komponenten wurden weitgehend als verteiltes System gesehen, die über einen Ethernet-Channel miteinander kommunizieren um unter anderem die Bekanntmachung zu realisieren. Die benötigten Informationen zur Produktion selbst werden jedoch in einer zentralen Datenbank (Knowledge Base) vorgehalten, auf welche die Komponenten zugreifen um z.B. die nächsten Produktionsschritte abzurufen.
In [25] werden Aspekte der Voraussetzungen, Parametrierung und Typen von Plug and Produce diskutiert und ein dreischichtiges Konzept für Plug and Produce Anwendungen hypothesiert.

2.2 Thread-Netzwerkprotokoll

Unter Thread versteht sich in diesem Kontext ein von der ThreadGroup [75] konzipiertes und entwickeltes Wirless-Personal-Area-Network (WPAN) Netzwerkprotokoll nach dem IEEE 802.15.4 Standard [1]. Es wurde speziell für IoT/Smarthome Anwendungen konzipiert und bietet seit der Version 1.3.0 auch eine native Matter [15] Integration an. Mit der OpenThread Implementierung [23] von Google, existiert eine Open Source (BSD-3-Clause-license) Implementierung des Protokolls nach der 1.3.0 Spezifikation.

2.2.1 Zusammenfassung des Protokolls und der Funktionsweise

Das Protokoll ist IPv6 basiert, es wird kein IPv4 unterstützt. Es ist applikationsagnostisch konzipiert und der Netzwerkverkehr ist verschlüsselt. Es werden vier Betriebsmodi bei den Netzwerkteilnehmern unterschieden:

- Border-Router: Stellt eine Brücke zwischen einem anderen Netzwerk (z.B. IPv4/IPv6 Ethernet/Wifi) und dem Thread-Netzwerk bereit und kann Pakete aus und in das Thread-Netzwerk routen.
- Leader: Trifft Entscheidungen für das Netzwerk. Vergibt die Router-Adressen und nimmt neue Router-Anfragen an. Wird automatisch durch einen Leader-Auswahlalgorithmus aus den verfügbaren Routern ausgewählt. Bei einem Ausfall des Leaders wird automatisch ein neuer Leader gewählt.

- Router: Routet Netzwerkpakete zu allen verfügbaren Nachbarn und erweitert damit das aufgebaute Thread-Netzwerk.
- Child: Kann nur Netzwerkverkehr empfangen/senden, aber nicht routen. Dieser Modus ist vor allem für batteriebetriebene Geräte gedacht.

Der Wechsel der verschiedenen Modi kann dynamisch geschehen, z.B. wenn ein Netzwerkteilnehmer im Router-Modus auch Border-Router fähig ist und ein neues Netzwerk angeschlossen wird, kann der Leader diesen Router zu einem Border-Router hochstufen. Ein neues Gerät beginnt immer im Child-Modus. Wenn dieses jedoch Router fähig ist, wird es vom Leader nach einer Router-Anfrage zu einem Router hochgestuft. Die Modi, mit Ausnahme des Leaders, der immer durch den Leader-Auswahlalgorithmus bestimmt wird, können auch statisch von der Netzwerkstack-Implementierung festgelegt werden. Thread integriert einen Mechanismus zur autonomen Selbstheilung, der in der Lage ist, auf dynamische Veränderungen der Netzwerktopologie zu reagieren. Durch die adaptive Anpassung der Routing-Struktur bei Geräteausfällen oder Netzwerkstörungen, gewährleistet das Protokoll kontinuierliche Konnektivität. Dadurch weist das es eine erhöhte Robustheit und Ausfallsicherheit auf und kreiert keinen Single-Point-of-Failure, solange mindestens ein Router im Netzwerk vorhanden ist. Das implementierte Routing-Protokoll optimiert die Übertragung von Daten innerhalb des Netzwerks, basierend auf der Netzwerktopologie und die sich daraus ergebenden Metriken. Hierbei agieren Geräte, welche im Router- oder Border-Router-Modus sind, als Schlüsselkomponenten, indem sie den besten Übertragungsweg bestimmen und die Daten entsprechend weiterleiten. Dieser Mechanismus trägt zur Optimierung der Netzwerkübertragungen und zur Vermeidung von Engpässen bei. Eine weitere wesentliche Eigenschaft des Protokolls liegt in der Fähigkeit, Daten aus externen Netzwerken in das Thread-Netzwerk zu routen. Dies wird durch die Geräte im Border-Router-Modus realisiert, die als Schnittstelle zwischen dem Thread-Netzwerk und externen Netzwerken agieren. Die Border-Router übernehmen die Protokollübersetzung, wodurch eine nahtlose Kommunikation zwischen Thread-Geräten und externen Umgebungen ermöglicht wird. [74]

2.2.2 Verwendung im Projekt

In diesem Projekt wird Thread eingesetzt um den modularen Ansatz des DAHS und der Plug and Produce Agents zu unterstützen. Aufgrund des Fokus auf ein Netzwerkprotokoll

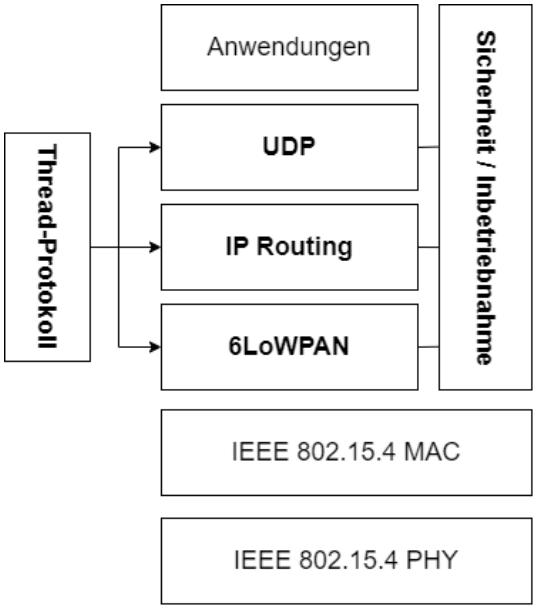

Abbildung 2.1: Thread-Netzwerkprotokoll Stack, Darstellung nach Definition aus ThreadNetwork Fundamentals [74]

ohne Single-Point-of-Failure, mit Selbstheilungsfeatures und der einfachen Kommissionierung von neuen Netzwerkteilnehmern [76] unterstützt Thread das Plug and Produce der geplanten Anwendung auf der Infrastrukturebene über mehrere Netzwerkteilnehmer. Es wird die Opensource Implementierung OpenThread von Google [23] in der BorderRouter Version als Netzwerkstack verwendet. Des Weiteren wird zur Kommissionierung von neuen Teilnehmern im Netzwerk die von der Thread Group veröffentliche Android Applikation "Thread 1.1 Commissioning App" verwendet.

2.2.3 Hardware

Für die Thread-Konnektivität werden SoCs inklusive Antennen benötigt, welche das Thread-Netzwerkprotokoll auf Hardwareebene unterstützen. Für dieses Projekt werden dazu die Development Kits und USB-Dongles basierend auf dem nRF52840 SoC [37] von Nordic Semiconductor genutzt. Diese bieten sich aufgrund der guten Dokumentation seitens Nordic Semiconductors [36] und der Verwendung in diversen Beiträgen von OpenThread [22] an.

2.3 OPC Unified Architecture

Bei OPC UA handelt es sich um einen Standard für den plattformunabhängigen Datenaustausch, der es ermöglicht Maschinendaten zu transportieren und maschinenlesbar semantisch zu beschreiben. Der Standard wird von der OPC Foundation [42] entwickelt und gepflegt. Seit 1996 steht mit OPC Classic, ein Windows COM/DCOM basierter Standard zur Verfügung. 2008 wurde mit OPC UA [45] der plattformunabhängige Standard veröffentlicht, der auf der Basis von OPC Classic entwickelt wurde. Die Spezifikation von OPC Classic kann unter [43] eingesehen werden (bezahlte Mitgliedschaft vorausgesetzt).

2.3.1 Organisationsmodelle in OPC UA

OPC UA definiert keinen strukturellen Aufbau der Objekte und Variablen in einem OPC UA Server. Entsprechend können beliebige anwendungsbezogene Organisationsmodelle zur Strukturierung in OPC UA entworfen werden. Zum Entwurf eines Modells wird die OPC UA Informationsmodellnotation [40] genutzt.

In [68] wird ein hierarchisches Organisationsmodell für OPC UA definiert. Es basiert auf dem Agent/Group/Role (AGR) Modell. Die Konzeption des Modells wird durch die verbesserte Transparenz und Adaptivität durch die organisatorische Struktur motiviert. Das AGR-Modell wird in [19] wie folgt definiert, übersetzt und paraphrasiert aus dem Englischen:

> *Das AGR-Modell basiert auf drei grundlegenden Konzepten: Agent, Gruppe und Rolle, die strukturell miteinander verbunden sind und nicht durch andere Grundelemente definiert werden können. Diese Konzepte erfüllen eine Reihe von Axiomen, die sie miteinander verbinden. Ein Agent ist in diesem Modell eine aktive und kommunikative Entität, die Rollen innerhalb von Gruppen übernimmt. Ein Agent kann mehrere Rollen innehaben und Mitglied mehrerer Gruppen sein. Das AGR-Modell legt dabei keine Einschränkungen auf die Architektur der Agenten fest. Eine Gruppe wird als eine Sammlung von Agenten definiert, die gemeinsame Merkmale teilen. Sie dient als Kontext für ein Muster von Aktivitäten und unterstützt die Aufteilung von Organisationen. Zwei Agenten können nur kommunizieren, wenn sie derselben Gruppe angehören. Ein Agent kann jedoch Mitglied mehrerer Gruppen sein, was die*

Definition von Organisationsstrukturen ermöglicht. Die Rolle ist die abstrakte Darstellung der funktionalen Position eines Agenten in einer Gruppe. Ein Agent muss in einer Gruppe eine Rolle spielen, kann aber mehrere Rollen übernehmen. Rollen sind gruppenbezogen und müssen von einem Agenten angefordert werden. Es besteht die Möglichkeit, dass eine Rolle von mehreren Agenten gespielt wird.

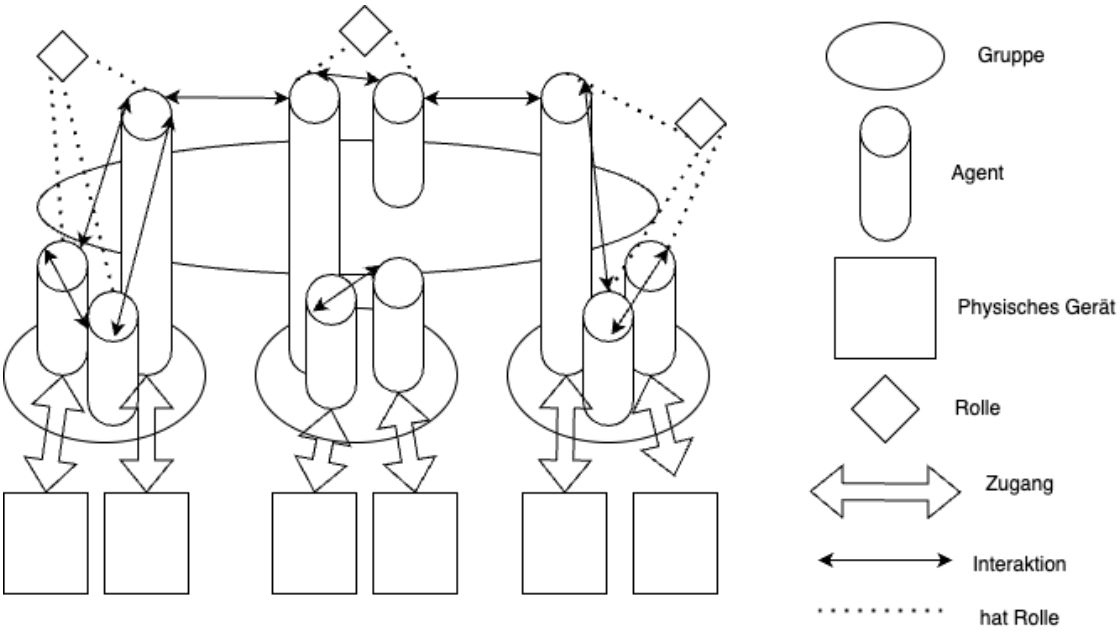

Abbildung 2.2: AGR basiertes Organisationsmodell, das die agentenbasierten Kontrolle von physischen Geräten zeigt, nach Grafik 1 aus [68]

In diesem Projekt wird als Beispiel ein OPC UA Organisationsmodell nach Vorbild aus [68] mit einer definierten Hierarchie umgesetzt um die Konfiguration, Parameter und Sensor- bzw. Aktorwerte der Plug and Produce Agents und das DAHS-MAS strukturiert nach einem entwickelten Informationsmodell zu verwalten und zu überwachen.

2.4 Multiagentensystem

Multiagentensysteme (MAS) sind ein Teilgebiet der verteilten künstlichen Intelligenz. In [53] werden MAS wie folgt definiert:

> *A Multi-Agent System (MAS) is an extension of the agent technology where a group of loosely connected autonomous agents act in an environment to achieve a common goal. This is done either by cooperating or competing, sharing or not sharing knowledge with each other. Multi-agent systems have been widely adopted in many application domains because of the beneficial advantages offered. Some of the benefits available by using MAS technology in large systems [78] are*
>
> 1. *An increase in the speed and efficiency of the operation due to parallel computation and asynchronous operation*
> 2. *A graceful degradation of the system when one or more of the agent fail. It thereby increases the reliability and robustness of the system*
> 3. *Scalability and flexibility- Agents can be added as and when necessary*
> 4. *Reduced cost- This is because individual agents cost much less than a centralized architecture*
> 5. *Reusability-Agents have a modular structure and they can be easily replaced in other systems or be upgraded more easily than a monolithic system*

Weitergehend werden in [53] auch die Probleme und Herausforderungen von MAS diskutiert. Obwohl Multiagentensysteme Vorteile gegenüber Einzelagentensystemen bzw. Architekturen nach anderen Programmierparadigmen bieten, besitzen sie auch signifikante Herausforderungen. Diese Herausforderungen umfassen Probleme mit der Umgebungsinteraktion, die limitierte Sicht auf verteilte Agenten und Limitierungen durch die Sensorik. Es sind auch Probleme durch Abstraktionsschwierigkeiten, der Konfliktlösung und Interferenzen zwischen den Agenten möglich. Agenten müssen häufig die Aktionen anderer vorhersehen und auf diese reagieren, beziehungsweise die Adaptivität besitzen um mit diesen Aktionen umgehen zu können, während sie sich in dynamischen Umgebungen bewegen. Dies führt zu potenziellen Instabilitäten. Des Weiteren können die Entscheidungen der Agenten zu sub-optimalen Ergebnissen führen, aufgrund der limitierten Wahrnehmung ihrer Umgebung. Dies ist besonders problematisch, wenn versucht wird globale Lösungen zu finden. Abstraktion und Konfliktlösung werden komplexer, wenn Agenten verschiedene Ziele verfolgen und um Ressourcen konkurrieren. Mechanismen zur Schlussfolgerung müssen heterogene Agenten und sich verändernde Umgebungen berücksichtigen. Entsprechend hängt die Eignung von Multiagentensystemen von der spe-

zifischen Anwendungsdomäne ab, speziell solcher die eine Interaktion von Entitäten mit unterschiedlichen oder konkurrierenden Zielen involvieren.

Im Vergleich von MAS mit anderen Programmierparadigmen, wie z.B. der Objektorientieren Programmierung (OOP), der funktionalen Programmierung oder der deklarativen Programmierung, ergeben sich Unterschiede besonders im Fokus auf Autonomie und Kommunikation bei der Entwicklung von Komponenten. Es wird jedoch auch gezeigt, wie diese Paradigmen in MAS genutzt werden. Agenten gehen in MAS über reine objektorientierte Modelle hinaus, indem sie nicht nur passive Entitäten sind, sondern dynamisch handeln und miteinander kommunizieren können. In der funktionalen Programmierung stehen Funktionen im Vordergrund. Agenten in MAS erweitern dieses Modell, indem sie nicht nur Funktionen ausführen, sondern auch aktiv miteinander interagieren und ihre Zustände dynamisch ändern können. Die deklarative Programmierung, insbesondere unter Verwendung von logik- und regelbasierten Ansätzen, findet auch in MAS Anwendung. Die Deklarativität ermöglicht eine Abstraktion von Ausführungsdetails und erleichtert die Modellierung komplexer, dynamischer Interaktionen zwischen den autonomen Agenten. Zusammengefasst konzentrieren sich Multi-Agentensysteme auf die Autonomie, Kommunikation und Kooperation/Konkurrenz ihrer Agenten, was sich vom Ansatz anderer Programmierparadigmen signifikant unterscheidet. Diese Konzentration ermöglicht es, komplexe, dynamische Umgebungen mit heterogenen, autonomen Agenten zu modellieren.

2.4.1 Anwendungsgebiete und Stand der Technik

MAS haben vor allem im (semi-)industriellen Sektor viele Anwendungsgebiete. Sie werden für die Kontrolle und Überwachung von vielfältigen Systemen eingesetzt, wie intelligenten Stromnetzen. Außerhalb des industriellen Sektor finden sich auch Anwendungen wie z.B. marktwirtschaftliche Simulationsmodelle [79]. Eine kürzliche Entwicklung in MAS ist die Implementation von Block-Chain-Technologie (BCT) [13]. Dabei werden benötigte Eigenschaften eines MAS, wie z.B. Konsensus von Agenten durch BCT realisiert und validiert. Diese Entwicklungen haben besonders für MAS welche mit sensiblen Daten arbeiten (z.B. Anwendungen aus dem Gesundheitsbereich) Relevanz, da Mechanismen zur Gewährleistung einer erhöhten Manipulationssicherheit eine Grundeigenschaft der BCT sind.

2.4.2 Beispiele von Multiagentensystemen

Es haben sich MAS für die verschiedensten Anwendungsgebiete etabliert. In [34] wird eine Auswahl auf verschiedene Aspekte wie u.a. der Benutzerfreundlichkeit, den Plattformeigenschaften und den Sicherheitsaspekten untersucht und verglichen. Es werden Jadex, eine in Java geschriebene Middleware und MASON, eine ebenfalls in Java realisierte agentenbasierte Simulationssoftware aus den vorgestellten MAS in [34] und osBrain und VOLTRON, welche in der Entwicklungsphase (siehe 4) des Projekts getestet wurden und in Python realisiert sind, mit ihren Anwendungsgebieten angeführt um eine kurze Übersicht zu verschaffen. Die Auswahl der vorgestellten MAS ist durch die Erfahrung des Autors mit diesen und der Relevanz für die Erstellung des in dieser Arbeit beschriebenen Systems getroffen und durch die Diversität der Anwendungsdomänen motiviert.

- MASON ist ein seit 2003 in Java entwickeltes MAS das auf Multiagentensimulationen spezialisiert ist. Es wird an der George Mason University in Fairfax entwickelt. Es bietet in der 21. Version vielfältige Möglichkeiten Agenten basierte Simulationen zu kreiieren, mit 2D und 3D Oberflächen [63].
- osBrain ist ein von OpenSistemas in Python3 entwickeltes General-Purpose-MAS. Es nutzt als Kommunikationsbasis zeroMQ und Pyro4 für die Systembereitstellung. Die neuste vorliegende Version ist 0.6.5. [50]
- VOLTRON ist ein Open Source entwickeltes MAS, mit dem Fokus auf die Steuerung und Verwaltung von Stromnetzen. In späteren Versionen wird ein General-Purpose-Ansatz verfolgt und vom Fokus mehr abgewichen. Es bietet als Kommunikationsplattformen zeroMQ und RabbitMQ an. Es liegt seit Oktober 2022 in der Version 8.2 vor, welche u.a. eine RESTful API hinzufügte. [72]
- Jadex ist ein von Active Components in Java entwickeltes serviceorientiertes MAS. Es wurde ähnlich zur Service Component Architecture (SCA) entworfen und um ein agentenorientiertes Konzept erweitert. [3]

Die Vorstellung dieser Auswahl zeigt, wie weit die Anwendungsdomäne für MAS ist. Es ist auch ersichtlich, wie unterschiedliche Kommunikationsbibliotheken zur Kommunikation der Agenten eingesetzt werden oder es für die Anwendung offen gelassen wird, diese auszuwählen.

2.4.3 Messaging

Unter Messaging versteht sich im Kontext der MAS die Kommunikation zwischen den Agenten untereinander, sowie zwischen den MAS-Komponenten und den Agenten. In den vorigen Abschnitten ist bereits erwähnt, dass die Kommunikation zwischen den Agenten eine signifikante Eigenschaft von MAS ist. Dabei haben sich verschiedene Kommunikationsmuster etabliert. Besonders die asynchrone Kommunikation ist für die lose gekoppelte Architektur von MAS vorteilhaft. Folgende Kommunikationsmuster werden unter anderem für MAS eingesetzt:

- **Publish-Subscribe [70]**: Agenten können Nachrichten zu spezifischen Themen oder Ereignissen veröffentlichen, die andere Agenten abonnieren.
- **Request-Reply [27]**: Ein Agent sendet eine Anfrage an einen anderen Agenten, der daraufhin mit den angeforderten Informationen oder Aktionen antwortet.
- **Message Broadcasting [?]**: Eine Nachricht wird von einem Agenten an mehrere Agenten gleichzeitig gesendet, um Informationen zu verbreiten oder Aktionen zu koordinieren.
- **Peer-to-Peer (P2P) [32]**: Agenten kommunizieren direkt miteinander, ohne einen zentralen Server oder Broker zu benötigen.
- **Mediated Communication [2]**: Die Kommunikation zwischen Agenten wird durch Proxies oder Mediatoren abgewickelt. Dies sind Komponenten, welche Nachrichten annehmen und weitergeben.
- **Agent Coordination [60]**: Agenten koordinieren ihre Aktionen durch Verhandlungen, Konsensbildung oder Vereinbarungsprotokolle.
- **Message Queues [59]**: Nachrichten werden in eine Warteschlange gestellt und können zu einem späteren Zeitpunkt von einem anderen Agenten abgerufen werden.

Des Weiteren ist es auch möglich Mischformen von Kommunikationsmustern zu bilden.

3 Design

In diesem Kapitel wird beschrieben, wie das DAHS und die Plug and Produce Agents entworfen werden. Es wird darauf eingegangen welche Technologien genutzt werden und kurze Diskussionen angeführt, die zur Entscheidung beigetragen haben.

3.1 Anforderungsanalyse

Die Anforderungen an das System wurden basierend auf der Zielsetzung aus dem Abschnitt 1 und dem Evaluationsszenario aus Abschnitt 5 in der folgenden Aufzählung definiert. Im Kapitel Design werden diese analysiert und berücksichtigt. Im Kapitel Umsetzung werden die Implementierungsdetails dieser Punkte aufgegriffen.

1. Erkennung neu angeschlossener Hardware und darauf basierende Agenteninstanziierung: Um der Anforderung an ein Plug and Produce System zu genügen, muss neu angeschlossene Hardware automatisch erkannt und basierend auf der Art der Hardware ein entsprechender Agent zur Verwendung dieser Hardware gestartet werden. Im Abschnitt 3.5.2 wird beschrieben wie diese Funktionalität im DAHS entworfen und umgesetzt ist.

2. Dynamisch konfigurierbare Agenten: Die Agenten des DAHS-MAS müssen bei der Instanziierung konfigurierbar sein, damit neue Hardware durch das Verteilen einer neuen Konfiguration unterstützt wird. Im Abschnitt 3.6.1 wird das Design erläutert, nach dem diese Funktionalität implementiert wird.

3. Dezentralisierte Verteilung von Agentenkonfigurationen: Damit Agentenkonfigurationen für neue Hardware vom Benutzer nicht manuell verteilt werden müssen und immer allen DAHS im Netzwerk alle Agentenkonfigurationen vorliegen, müssen diese von DAHS zu DAHS verteilt werden. Das Design nach dem diese Funktionalität implementiert wird, ist im Abschnitt 3.5.3 beschrieben.

4. Kompatibilität zum Thread-Protokoll: Zur Realisierung auf der Anwendungsebene, müssen alle Komponenten, welche Zugriff auf das Netzwerk haben, IPv6 unterstützen. Im Abschnitt 4.1 wird darauf eingegangen, dass diese Anforderung zur Entwicklung des minimalen DAHS-MAS geführt hat.

5. Kompatibilität mit OPC UA: Alle aktiven Komponenten müssen die Funktionalität besitzen, ihre Informationen im lokalen OPC UA Server des Thread-Knotens zu organisieren. Der Abschnitt 4.6 erläutert, wie diese Funktionalität umgesetzt wird.

6. OPC UA Datenmodell mit Organisationsstruktur: Zur strukturierten Darstellung der Informationen im OPC UA Server, muss ein OPC UA Datenmodell entwickelt werden, das die Anforderungen an das im Abschnitt 2.3.1 vorgestellten und ausgewählten Organisationsmodell erfüllt. Im Abschnitt 3.7 wird das entwickelte Datenmodell gezeigt und weiter erläutert.

7. Unabhängige Komponenten, ohne Single-Point-of-Failure: Die zu entwickelnden Komponenten des DAHS und der Plug and Produce Agents müssen robust gegen den Ausfall von einzelnen Komponenten sein.

8. Adaptive Systemperformanz, basierend auf angeschlossener Hardware: Das System soll basierend auf der verfügbaren Hardware die Performanz adaptiv anpassen können. Zur Erfüllung dieser Anforderung wird im Management Agenten eine Performanzstufe bestimmt, welche die Abklingzeit zwischen angenommenen Aufträgen anpasst und somit die Erfüllungsgeschwindigkeit mitbestimmt. Im Abschnitt 4.3 werden die Implementierung und die einzelnen Stufen beschrieben.

3.2 Messaging

Im Entscheidungsprozess des Designs wurde zwischen RabbitMQ [12] mit einem MQTT Plugin und ZMQ [73] entschieden. Diese Message Queues wurden aufgrund der Spezifikationen und der Verbreitung in bereits etablierten MAS Anwendungen ausgewählt. Die verteilte Architektur von RabbitMQ, bei der für ein komplett verteilten Ansatz entweder ein Cluster von Message Brokern verwendet oder eine Föderation zwischen den Brokern hergestellt werden muss [11], stellt eine kompliziertere Architektur dar. Deshalb wird als Grundlage für die Messaging-Architektur ZMQ genutzt, da bei dieser flexiblere Broker-Architekturen [69] möglich sind. Des Weiteren ist ZMQ eine leichtgewichtigere

Implementierung [54] und besitzt im Vergleich zu RabbitMQ bessere Performanzeigenschaften [18]. Die asynchrone Nachrichtenaustauschbibliothek bietet sich zur Realisierung der Anforderungen an das DAHS an, da eine Message Queue genutzt werden kann, ohne einen dedizierten (zentralen) Message Broker. ZMQ unterstützt zudem in der Version 4.x durchgehend IPv6. Um keine Abhängigkeiten zwischen den Agenten, Modulen und Netzwerkknoten durch das Messaging einzuführen wird das Publish-subscribe pattern (PUB/SUB) genutzt. Damit weiterhin kein zentraler Message Broker für die Verwaltung der Subscriptions und das Forwarding der Nachrichten notwendig ist, wird ein XPUB/XSUB Proxy entwickelt (das X in XPUB/XSUB steht im Kontext von ZMQ für "multiple", da diese Art des Sockets von mehreren PUB/SUB-Clients verwendet werden kann), der auf jedem Netzwerkknoten verfügbar ist. Um das Messaging über alle Netzwerkknoten hinweg zu ermöglichen, verbinden sich die XPUB/XSUB Proxies untereinander und bilden eine verteilte Message Queue. Die Netzwerkknotenentdeckung, damit die Proxies sich verbinden können, wird im nachfolgenden Abschnitt beschrieben.

3.3 Netzwerkknotenentdeckung

Um die Anforderungen erfüllen zu können, dass Nachrichten zwischen DAHS und Agenten verschickt werden können, bzw. das Publishment und die Subscriptions auch über einen Netzwerkknoten hinweg funktionieren, müssen den Proxies auf Anwendungsebene die Adressen aller Thread-Netzwerkknoten bekannt sein, auf welchen auch ein DAHS läuft. Dazu wird ein Modul entwickelt, das die Netzwerkknoten auffindet und in einer Liste für das DAHS aufbereitet. Da auf Netzwerkebene, im Thread-Netzwerk, bereits alle Netzwerkknoten in einem Mesh-Netzwerk miteinander verbunden sind, kann eine Spezifikation von OpenThread bei der Entwicklung genutzt werden. Nach IPv6-Adressierungs-Spezifikation [23] gehört jedes FTD (Full-Thread-Device) und MED (Minimal-End-Device) der Multicastgruppe ff03::1 an. Das Modul sendet bei der Initialisierung und in einem konfigurierbaren Zeitabstand eine Nachricht an die Multicastgruppe und empfängt danach die Antworten aller Netzwerkteilnehmer. Daraus wird die Liste erstellt, welche alle aktiven DAHS im Thread-Netzwerk mit ihren Netzwerkadressen enthält.

3.4 Aufbau der Infrastruktur

Damit die Infrastruktur des DAHS und der Plug and Produce Agents gestartet werden kann, wird ein Modul entwickelt das die grundlegende Infrastruktur-Komponenten startet und die notwendigen Informationen für diese ermittelt bzw. aus Konfigurationsdateien ausliest. Dabei wird die IP-Adresse des konfigurierten Netzwerkinterfaces ermittelt und die zu verwendenden Ports für die Komponenten, sowie die Einstellungen für den OPC UA Server (Namespace, Port, Binding-Interface) aus der Konfigurationsdatei ausgelesen. Des Weiteren soll auch ein vereinfachtes Debugging der Agenten und des OPC Servers möglich sein, indem im Infrastruktur-Setup direkt Agenten instanziiert und OPC Objekte mithilfe der OPC Server-Referenz erstellt und manipuliert werden. Dabei ist das Infrastruktur-Setup vorbereitet um auf unterschiedlicher Hardware genutzt zu werden (Raspberry Pi, Beaglebone Black, Windows WSL2, MacOS).

3.5 DAHS

Das DAHS wird nach dem Vorbild und den vorhandenen Informationen der Implementierung des AHS (Agent-Handling-System) aus "Identification of resources and parts in a Plug and Produce system using OPC UA" [10] entwickelt. Wobei der primäre Faktor die Realisierung der Verteilung und Unabhängigkeit zu anderen Komponenten im System ist.

Aufgrund der Entscheidung, ein eigenes minimales Multi-Agent-System zu entwickeln, um die Voraussetzungen wie im Abschnitt 4.1 beschrieben zu erfüllen, wird das DAHS auch die Komponenten für die Steuerung und Verwaltung der Agenten enthalten.

3.5.1 Erstellung/Aktualisierung/Verteilung der Agenten-Dictionaries

Für die Verwaltung der internen und externen Agenten, werden in jedem DAHS zwei Agenten-Dictionaries geführt. Beide Agenten-Dictionaries werden bei der Instanziierung eines neuen Agenten befüllt bzw. aktualisiert. Das interne Agent-Dictionary enthält alle notwendigen Informationen über den Agenten, um diesen steuern und überwachen zu können. Das Netzwerk Agenten-Dictionary enthält die notwendigen Informationen um über das Netzwerk direkt mit einem Agenten interagieren zu können. Wenn ein neuer

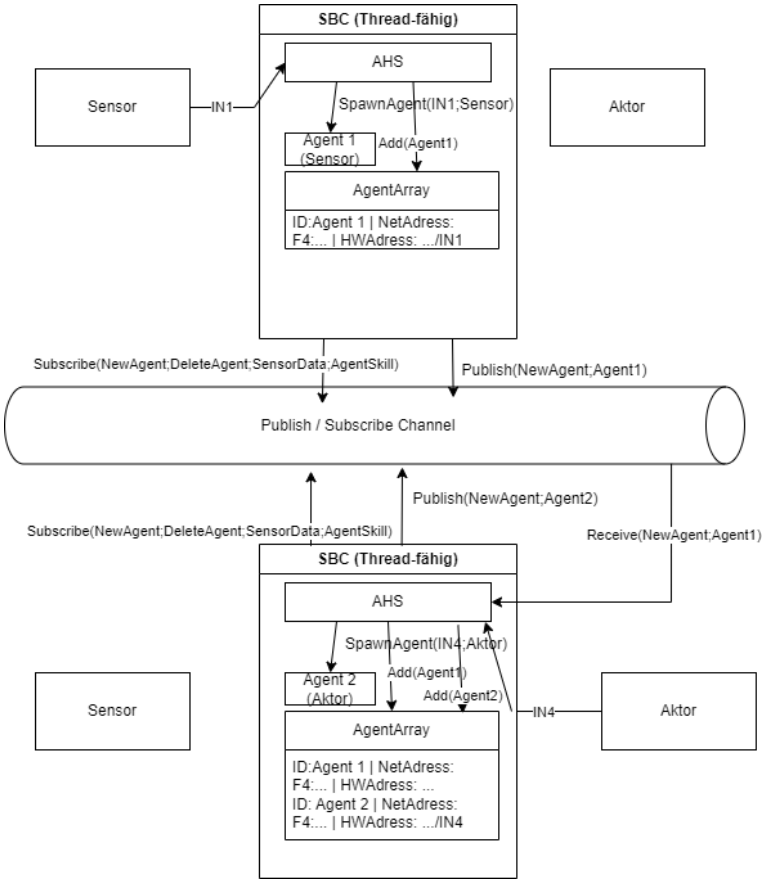

Abbildung 3.1: Erster Entwurf des DAHS und der Basisfunktionen

Agent instanziiert wurde, wird das aktualisierte Netzwerk Agenten-Dictionary publiziert und die anderen DAHS tragen die neuen Einträge in ihr Netzwerk Agenten-Dictionary ein. Dabei enthält das Netzwerk Agenten-Dictionary keine Thread-Referenzen auf die Agenten aus dem internen Agenten-Dictionary, da diese nicht serialisierbar sind und keinen Nutzen für die externen DAHS bereitstellen.

3.5.2 Erkennung neuer Hardware

Um einen Agenten basierend auf der angeschlossenen Hardware/Peripherie instanziieren zu können, muss das DAHS diese neu angeschlossenen Komponenten erkennen. In [10] wird die Erkennung neuer Hardware mithilfe des Dynamic Host Configuration Protocol (DHCP) über einen zentralen DHCP-Server realisiert und setzt entsprechend für

jede Hardwareeinheit einen IP-fähigen Host vor, damit diese vom DHCP-Server gefunden werden können. Das DAHS soll direkt angeschlossene Hardware an verschiedenen Interfaces erkennen können, ohne dass ein DAHS für jede Hardwareeinheit notwendig ist. Entsprechend ist die Erkennungsmethode aus [10] nicht einsetzbar und es wird eine neue Funktionalität zur Erkennung entwickelt. Die Erkennung basiert vorrangig auf den Agenten-Konfigurationen, welche dem DAHS vorliegen. In diesen ist beschrieben an welchen Ports oder Adressbereichen die Hardware für diesen Agenten zu finden ist. Das DAHS durchläuft vier Methoden zur Erkennung: I²C-Scan, OneWire-Protokoll-Scan, USB-Scan und GPIO-Scan. Dabei vergleicht das DAHS die bereits instanziierten Agenten Adressen und Ports mit den Gescannten und bestimmt anhand dessen, ob neue Hardware angeschlossen wurde. Die Hardwareerkennung wird periodisch in jedem Durchlauf der Processingloop des DAHS ausgeführt, siehe Abbildung A.2.

3.5.3 Verteilung von Konfigurationen

Eine der Hauptfunktionalitäten des DAHS, das es vom Vorbild vom AHS wie in [10] unterscheidet ist die Entwicklung eines Verteilungsmechanismus der Agentenkonfigurationen ohne einer zentralen Komponente. Bei der Initialisierung des DAHS werden alle Agentenkonfigurationen in das DAHS eingelesen und ein Observer gestartet, der das Verzeichnis der Agentenkonfigurationen überwacht. Wird über die UI eine neue Konfiguration hochgeladen, veranlasst das Observer-Event, dass diese Konfiguration vom DAHS eingelesen und danach per ZMQ an das NewAgentConfig-Topic publiziert wird. Dieses Event wird auch ausgelöst, wenn sich ein neues DAHS im Netzwerk registriert und bei der Initialisierung die Konfigurationen aus dem Netzwerk empfängt. Beim Empfang einer neuen Konfiguration wird diese vom DAHS dekodiert und wie nativ hochgeladene Konfigurationen im Verzeichnis der Agentenkonfigurationen als YAML-Datei abgelegt.

3.6 Plug and Produce Agents - Konfigurierbare Agenten

Die Agenten des geplanten MAS, die Plug and Produce Agents, werden nach dem Vorbild typischer Multiagentensysteme entworfen, siehe Abschnitt 2.4. Entsprechend muss jeder Agent die Möglichkeit haben mit anderen Agenten und dem System zu kommunizieren, ein gegebenes Verhalten ausführen können und vom DAHS verwaltbar sein. Dabei sollen zwei verschiedene Möglichkeiten zur Kreierung von neuen Agenten zur Verfügung

stehen. Ein statischer Ansatz, bei dem sich ein neu geplanter Agent von der Basisagentenklasse ableitet und diese um Funktionalitäten erweitert und ein dynamischer Ansatz bei dem eine neue Agentenkonfiguration geschrieben wird, welche bei der Instanziierung vom DAHS geladen wird und der Code zur Laufzeit im Agenten ausführbar gemacht wird. Der statische Ansatz soll dabei für Agenten genutzt werden, welche einen größeren Funktionsumfang haben und deshalb u.U. auch Änderungen im DAHS erforderlich machen. Die nächsten Abschnitte beleuchten die Umsetzung von dynamischen Agenten und zwei geplante statische Agenten.

3.6.1 Konfiguration

Die Plug and Produce Agents müssen dynamisch konfigurierbar sein, je nach hinzugefügter Hardware. Diese Konfigurationen müssen auch in einer Form dem DAHS vorliegen, welche sich zur Verteilung im Thread-Netzwerk eignet. Entsprechend wird eine Template-Konfiguration basierend auf dem YAML-Dateiformat erstellt. Die Entscheidung ist auf YAML gefallen, aufgrund der Python ähnlichen Formatierung. Entsprechend kann eine neue Konfiguration mit Python-Code geschrieben und formatiert werden, ohne auf weitere Formatierungsschwierigkeiten zu stoßen. Das Konfigurationsdesign besteht aus vier Teilen. Der erste Teil beschreibt die Informationen welche das DAHS braucht, um angeschlossene Hardware der entsprechenden Konfiguration zuzuordnen. Dazu gehört der Konfigurationsname, der Agententyp, der Port, die Hardwareadresse und der Anschlusstyp. Im zweiten Teil wird eine Methode definiert, welche benötigte Python-Packages nachinstallieren kann. Dazu wird beim Aufruf dieser Methode ein Subprocess mit pip [71] gestartet, alle angegebenen Packages heruntergeladen und installiert, falls diese nicht lokal verfügbar sind. Im dritten Teil wird das Verhalten des Agenten bestimmt, indem eine Agenten-Methode und eventuelle weitere Methoden definiert werden. Diese Agenten-Methode wird in der Processing-Loop des Agenten repetitiv aufgerufen. Im vierten Teil werden die Attribute für das OPC UA Datenmodell 3.7 definiert, damit diese beim instanziieren eines neuen Agenten direkt in den OPC UA Server geschrieben werden können. Eine detaillierte Beschreibung der einzelnen Agentenkonfigurationen und wie diese umgesetzt werden, ist im Anhang im Abschnitt A.1 Agentenkonfigurationen enthalten. Es werden die genutzten Python-Packages für die Konfigurationen benannt und die Funktionsweise des Agenten beschrieben. Des Weiteren sind auch die Agentenkonfigurationen im YAML-Format enthalten. Im Kapitel Umsetzung, im Abschnitt Hardware wird beschrie-

ben für welche Hardware 4.8 eine Konfiguration angelegt wurde und wie diese realisiert wurden.

3.6.2 ManagementAgent

Um die Verwaltung der Plug and Produce Agenten zu vereinfachen und eine Komponente zu entwickeln, welche die Performance des Gesamtsystems überwacht und dynamisch anpassen kann, wird ein Agent zum Management der Agenten in einer Gruppe realisiert. Der ManagementAgent wird vom DAHS nach dem Aufsetzen der Infrastruktur instanziiert und kreiert dabei die OPC UA Gruppe für die Verwaltung der Plug and Produce Agents. Der ManagementAgent prüft im OPC UA Server welche Agenten in der Gruppe angemeldet sind und kann anhand der vorhandenen Daten die Performance des Systems kontrollieren. Des Weiteren wird über diesen Mechanismus auch eine Plausibilitätsprüfung der Werte in OPC UA durchgeführt, wenn mehrere Agenten einer Gerätegruppe zur Verfügung stehen. Eine weitere Hauptfunktionalität des Management Agenten besteht in der Annahme und Weiterleitung von Aufträgen für die Agenten in der Gruppe. Dazu nimmt dieser Aufträge, welche z.B. über das UI per ZMQ an die Gruppe geschickt werden, entgegen und teilt diese basierend auf der Datenlage einem passenden Agenten zu oder lehnt den Auftrag ab.

3.6.3 PrintAgent

Um das Evaluationsszenario wie im Abschnitt 5.3.2 beschrieben umsetzen zu können, wird ein Agent realisiert, der die Octoprint-REST-API [28] abstrahiert. Dadurch kann dieser jeglichen Octoprint-kompatiblen 3D-Drucker [29] steuern und dessen Status in OPC UA abbilden. Dieser Agent soll auch eine weitere Methode zeigen, wie Plug and Produce Agenten erstellt werden können. Im Gegensatz zu den beschriebenen Agenten aus 3.6, leitet sich dieser Agent direkt vom generischen Agenten ab und nutzt keine Konfigurationsdatei.

Das Klassendiagramm des Entwurfs ist im Anhang in den Grafiken A.1 zu finden.

3.7 OPC UA Datenmodell

Zur Überwachung des DAHS und der Plug and Produce Agents wird OPC UA eingesetzt. Damit der Zugriff auf die Daten strukturiert und für die Agenten angepasst erfolgt, wird ein OPC UA Datenmodell erstellt, angelehnt an die OP UA Modelling Best Practice [41]. In dem Datenmodell wird eine Hierarchie genutzt, angelehnt an das vorgestellte Modell in [68], welches allgemein auf dem AGR-Modell basiert, siehe Abschnitt 2.3.1.

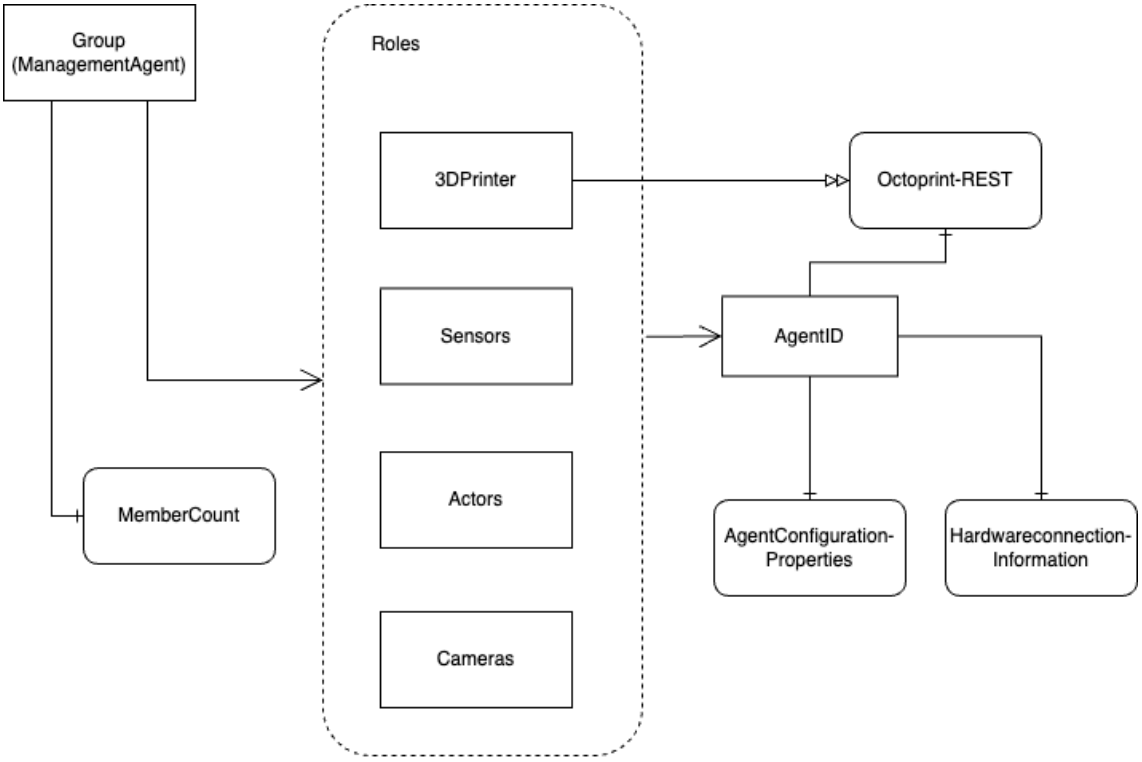

Abbildung 3.2: Informationsmodell nach OPC UA Notation [40]

In Abbildung 3.2 ist das entwickelte OPC UA Informationsmodell nach der OPC UA Informationsmodellnotation [40] zu sehen. Die Rollen sind einzelne Komponenten und sind zur besseren Visualisierung mit einer gestrichelten Linie umrahmt. Dies entspricht nicht der OPC UA Notation und hat entsprechend keine funktionalen Eigenschaften. Dieses Design mit der AgentID und den Variablen AgentConfiguration-Properties und der Hardwareconnection-Information wurde gewählt, um eine dynamische Struktur zu ermöglichen. Basierend auf der Agentenkonfiguration und wie die Hardware für diesen angeschlossen wird, stellt ein instanziierter Agent diese Informationen in OPC UA zur

Verfügung und kann auf diesen Variablen bei Bedarf arbeiten. Ein Agent mit der Rolle 3DPrinter abstrahiert die Informationen der Octoprint-REST-Schnittstelle und stellt diese in OPC UA dar. Entsprechend ändert sich das Informationsmodell dynamisch basierend auf dem angeschlossen 3D-Drucker Modell. Der ManagementAgent stellt die Gruppe für die Hierarchie bereit und besitzt eine Variable, welche angibt wie viele Agenten in der Gruppe aktiv sind.

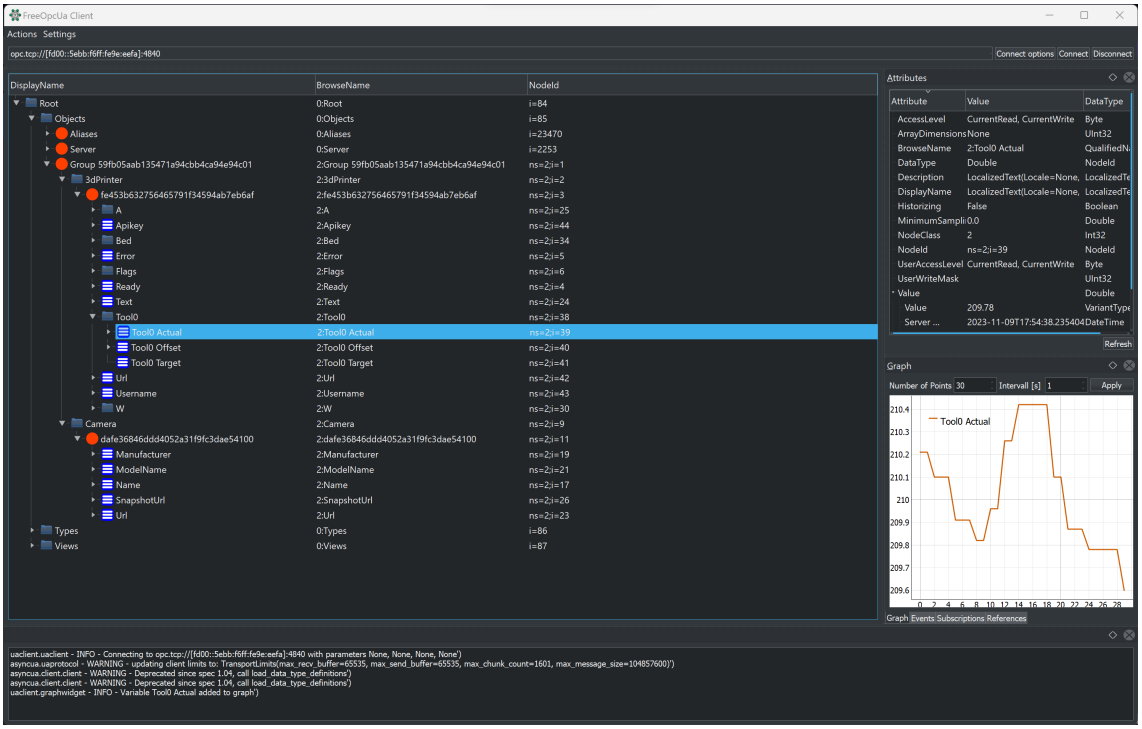

Abbildung 3.3: Gruppe mit 3D-Drucker und Kamera im OPC UA Server, Anzeige mit dem FreeOPCUA-Client

4 Umsetzung

In diesem Kapitel wird beschrieben, wie das DAHS und die Plug and Produce Agents umgesetzt werden. Es wird darauf eingegangen wie die Designentscheidungen und Technologien aus dem Design-Kapitel 3 implementiert und umgesetzt werden. Des Weiteren wird auf die verwendete Hardware eingegangen, welche für die Evaluierung ausgewählt wurde.

4.1 Grundlegende Implementierungsentscheidungen und Entwicklungsumgebung

Es wurden zwei MAS als Grundlage für das DAHS und die Plug and Produce Agents getestet, osBrain [50] und Voltron [72], die aber beide nicht alle grundlegenden Aspekte der Netzwerkfähigkeit mit IPv6 und die dynamische Konfigurierung von Agenten erfüllen konnten. osBrain weist Probleme mit der IPv6-Kompatibilität auf. In der in der genutzten Remote Object Bibliothek Pyro4 [31] schlägt das Binden an eine IPv6-Adresse fehl, da diese von osBrain fehlerhaft übergeben wird. Da das Projekt zur Zeit der Erstellung dieser Arbeit nicht aktiv weiterentwickelt wird und das Problem nicht behoben werden konnte, musste von der Nutzung von osBrain abgesehen werden. Bei Voltron lag die Problematik bei der dynamischen Konfiguration der Agenten, welche dort in zwei Schritten stattfindet. Im Vorfeld wird der Agentencode geschrieben, der das Verhalten des Agenten beschreibt. Danach wird eine Agentenkonfiguration erstellt, die den beschriebenen Agenten der Plattform bekannt macht. Dieses zweiphasige Prinzip hat sich als ungeeignet für das Design des DAHS herausgestellt, da keine simple Möglichkeit gefunden wurde das Agentenverhalten und die Agentenkonfiguration zu kombinieren und die Bekanntmachung zu automatisieren. Deshalb wird ein minimales MAS basierend auf den Anforderungen entworfen und entwickelt. Dabei sind die ausschlaggebenden Anforderungen an das minimale MAS, dass alle Komponenten IPv6 kompatibel sind, eine OPC UA Kompatibilität für alle adaptiven Komponenten gegeben ist und das Agenten dynamisch

bei der Laufzeit mit neuen Konfigurationen instanziiert werden können, ohne dass diese Konfigurationen dem MAS beim Start bekannt sein müssen. Die grundlegenden umzusetzenden MAS-Bestandteile für das minimale MAS sind dabei die Agenteninstanziierung, ein Directory-Service für lokale und entfernte (remote) Agenten, die Agentenverwaltung und ein Kommunikationsschema. Entsprechend wird das MAS nicht die vollen Funktionalitäten bieten, wie ein MAS nach der Definition aus Abschnitt 2.4, sondern speziell zur Erfüllung der Anforderungen an diese Arbeit und das Evaluationsszenario entworfen. Das MAS wird wie die bereits getesteten MAS in Python [56] umgesetzt um das Deployment und Testen auf den einzelnen Netzwerkknoten zu vereinfachen und um die vielfältigen existierenden Python-Libraries zu verschiedensten Sensoren/Aktoren nutzen zu können. Bei der Entwicklung wird eine Kompatibilität mit Python 3.9-3.11 sichergestellt. Für die OPC UA Anbindung wird die Server und Client Python-Implementierung opcua-asyncio [20] von FreeOpcUa mit dem Syncwrapper genutzt. Alle Komponenten werden auf Basis der threading Library von Python entwickelt, da dies bei den I/O intensiven Tasks, wie den Plug and Produce Agents, einen Performanzgewinn im Gegensatz zur multiprocessing Library darstellt und simpler zu implementieren ist, sowie eine höhere Kompatibilität gewährleistet als die Nutzung von asyncio [67]. Die Entwicklung wird mit der PyCharm IDE 2023.2.4 (Professional Edition) [30] von JetBrains durchgeführt.

4.2 Prototyp

Für die erste Iteration wurde der OPC UA Server anhand der Beispielimplementierung des minimalen OPC UA Servers von FreeOpcUa des opcua-asyncio Packages [21] implementiert, sowie ein rudimentäres DAHS, das stub Agenten instanziieren kann und mit diesen über ZMQ kommunizieren kann. In der nächsten Iteration wurde die Kommunikation über mehrere DAHS hinweg realisiert, dazu wurde nach dem Vorbild der beschriebenen Advanced Pub-Sub-Pattern in [26] der PubSub-Proxy implementiert. In den nächsten Iterationen wurden sukzessive die Basis-Agentenklasse und deren Konfigurationen implementiert. Auf weitere wichtige Implementierungsdetails wird in den folgenden Abschnitten eingegangen.

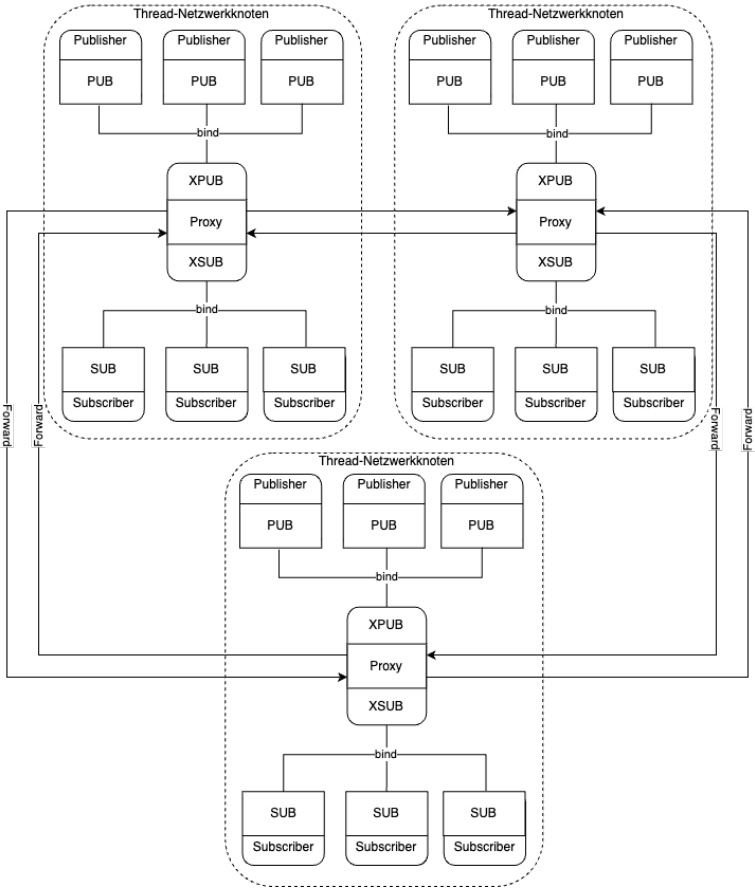

Abbildung 4.1: Konzeptionelle Ansicht der Funktionsweise des PubSub-Proxies mit drei Thread-Netzwerkknoten

4.3 Performanzstufen

Um zu zeigen, wie das Hinzukommen und Entfernen von Hardware, und somit die Agenten im System, einen Einfluss auf die Performanz auf das Gesamtsystem hat, wird im Management Agenten eine sechsstufige Bewertung des Systems implementiert. Die Stufen bestimmen, die Abklingzeit zwischen dem Abschluss eines 3D-Drucks und dem nächsten in der Warteschlange bzw. dem nächsten angenommenen. Die Stufen werden wie in der Tabelle 4.1 gezeigt definiert.

Die Präzision der angeschlossenen Hardware, speziell der Sensoren, hat einen direkten Einfluss auf die Performanz des Systems. Die Präzision wird als gleitende Skala zwischen

Performanzstufe	Abklingzeit (min)	Abklingzeit (s)	Notwendige Hardware
0	30	1800	Keine
1	20	1200	Filamentwaage
2	15	900	Gassensor
3	10	600	Filamentwaage + Gassensor
4	7,5	450	Filamentwaage + Gassensor + Temperatursensor
5	5	300	Filamentwaage + Gassensor + Temperatursensor + Partikelsensor

Tabelle 4.1: Performanzstufen mit Angabe der notwendigen Hardware und zugehöriger Abklingzeit in Minuten und Sekunden

den Stufen genutzt und passt die Abklingzeit entsprechend der prozentualen Annäherung zur nächsten Performanzstufe an.

4.4 Plausibilitätsprüfung

Um die Plausibilität von Sensorwerten festzustellen, wird in einer Gruppe in der drei oder mehr Sensoren mit der gleichen gemessenen Einheit registriert sind, vom ManagementAgent eine Plausibilitätsprüfung der Sensorwerte durchgeführt. Für die Plausibilitätsprüfung wird eine Studentisierung der Werte durchgeführt und unter Berücksichtigung der Präzision der Sensoren nach Ausreißern gesucht. In der Standardkonfiguration des Management Agenten werden Ausreißer mit einer Abweichung von mehr als 10% zur Fehlerbehandlung an das DAHS gemeldet und bei mehrmaligem Vorkommen in den Berechnungen ausgeschlossen.

4.5 Fehlererkennung und -behandlung

Die Agenten implementieren ihre eigene Fehlererkennung und -behandlung für etwaige Hardware und können bei Bedarf das DAHS im Falle eines nicht behebbaren Fehlers informieren. Der ManagementAgent einer Gruppe kontrolliert periodisch, ob Agenten

weiterhin ihre zugehörigen Werte im OPC UA Server aktualisieren und ob weiterhin Nachrichten per zeroMQ empfangen werden. Falls ein Agent seit 180 Sekunden keinen Wert aktualisiert hat, oder keine Nachricht von diesem empfangen wurde, wird dieser als überfällig markiert und das DAHS informiert. Es wird auch versucht den betreffenden Agenten zu kontaktieren. Das DAHS prüft in jedem Arbeitszyklus ob einer der Threads der Agenten im lokalen Agenten-Dictionary beendet wurde oder ob einer dieser Agenten als überfällig markiert wurde und beendet diese korrekt und löscht die Referenzen dieser. Falls die Hardware noch angeschlossen ist, wird durch die Hardwareerkennung ein neuer Agent an Stelle des fehlerhaften Agenten instanziiert.

4.6 OPC UA Implementierung

Die Agenten implementieren Methoden zum Anlegen und Aktualisieren von OPC UA Objekten und Variablen. Der ManagementAgent implementiert zusätzlich Methoden zum Abfragen aller verfügbaren Knoten im OPC UA Server und stellt die initiale Struktur im OPC UA Server her, wie sie durch das Datenmodell in Abschnitt 3.7 beschrieben ist. Das DAHS implementiert einen OPC UA Client, um im Falle des Ausfalls eines Agenten zu versuchen die verwaisten OPC UA Referenzen zu entfernen.

4.7 Benutzeroberfläche / UI

Zur Analyse und Debugging der OPC UA Server werden die OPC UA Clients, FreeOpcUa des opcua-asyncio Projekts und UaExpert von Unified Automation [77] eingesetzt. Für die Benutzerinteraktion wird eine rudimentäre Python-Anwendung auf Basis vom tkinter-Python-Package [57] implementiert, siehe Abbildung 4.2, das rudimentär die OPC UA Struktur anzeigen kann und über ZMQ mit den DAHS und Agenten kommunizieren kann. Über diese Oberfläche ist auch das Hochladen von G-Code-Dateien für die Druckaufträge, sowie YAML-Dateien für die Agentenkonfigurationen, über ZMQ möglich.

4.8 Hardware

Das DAHS mit den verschiedenen Möglichkeiten der Hardwarescans wird primär für Raspberry Pi SBCs der Version 2-4 entwickelt. Es wird weiterhin eine Basiskompatibi-

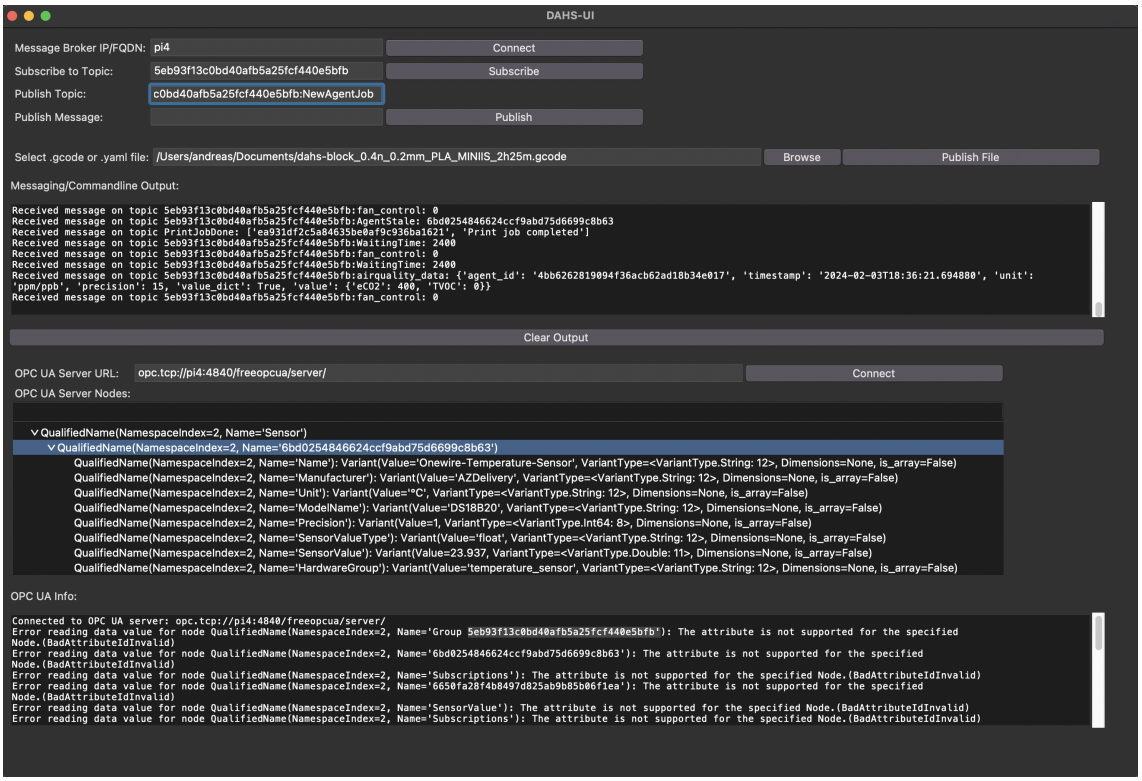

Abbildung 4.2: DAHS-UI, verbunden mit einem der Thread-Nodes zum Mitschneiden der Messages in einer Gruppe und zum Hochladen eines neuen Druckauftrages. In der unteren Hälfte ist in einer TreeView die zugehörige Gruppe in OPC UA zu sehen

lität mit BeagleBone Black SBCs gewährleistet, wobei die Hardwarescans durch emulierte Hardware ersetzt werden. Das Debugging ohne die Notwendigkeit eines Thread-Netzwerkes ist auch auf x86 Hardware mit Windows unter der WSL2 mit Debian oder nativem Debian und Mac OS möglich. Für die Thread-Konnektivität werden wie bereits im Abschnitt 2.2.3 beschrieben Entwicklungsboards und USB-Dongles von Nordic Semiconductor mit dem nRF52840 SoC eingesetzt.

Auswahl der Sensoren:

- Wägezelle mit HX711 Amplifier zum Bau einer Filamentwaage
- SDS011 Laser PM2.5/PM10 Partikelsensor für allgemeine Luftqualität
- SGP30 Gassensor zur Messung von TVOC

- SGP40 Gassensor zum Bestimmen des VOC-Index
- DS18B20 OneWire-Protokoll Temperatursensor, zur Temperaturmessung
- Logitech C170 als Webcam zur Überwachung
- Noctua 5V PWM Lüfter als Ersatz zur Lüftersteuerung

Der Aufbau und die genutzten Python-Packages für die Agentenkonfigurationen der Sensoren und Aktoren ist im Anhang im Abschnitt A.1 beschrieben.

4.9 Klassendiagramm

Das Klassendiagramm des Systems ist im Anhang zu finden. Siehe Abbildung A.1. Das Diagramm zeigt die Abhängigkeiten der Klassen zueinander. Das DAHS, der OPC-Server, PubSubProxy und ThreadNetworkNodeFinder werden vom infrastructureSetup instanziiert und erhalten von diesem die notwendigen Informationen zum Netzwerk und den konfigurierten Einstellungen. Das DAHS instanziiert die Agenten basierend auf der Hardwareerkennung. Die Agentenklasse wird zur Laufzeit modular durch die Agentenkonfigurationen (AgentConfig) erweitert. Der ManagementAgent und PrintAgent leiten sich von der Agentenbasisklasse ab und präsentieren dadurch eine weitere Möglichkeit spezielle Agenten zu entwickeln.

5 Evaluation

In diesem Kapitel wird beschrieben, wie das DAHS und die Plug and Produce Agents evaluiert werden. Dazu wird die Testumgebung, sowie Szenarien zur Evaluation beschrieben. Es werden einfache Funktionstest der einzelnen umgesetzten Komponenten definiert und danach auf ein großes Evaluationsszenario eingegangen, das einen umfänglicheren Einblick in die praktische Nutzung der Anwendung gibt.
Die Evaluation zeigt dass die Anforderungen aus dem Abschnitt 3.1 erfolgreich erfüllt sind. Die Ergebnisse werden im Kapitel 6 diskutiert.

5.1 Testumgebung

Die Evaluation wird in einem Raum mit den Ausmaßen von 4m x 2,5m x 2,5m durchgeführt. In diesem befinden sich drei jeweils 1,5m voneinander entfernte Single-Board-Computer (SBC) welche mit Thread-RPCs augestattet sind (SBC-Thread-Knoten). Diese drei SBC-Thread-Knoten werden jeweils als eine Produktionszelle angesehen und besitzen ihr eigenes DAHS. Eine dieser Zellen ist komplett mit Hardware ausgestattet und die anderen nutzen Hardware-Sensoren mit virtuellen Aktoren. Die genutzte Hardware in der vollausgestatteten Zelle wird im Abschnitt 4.8 beschrieben.

5.2 Testmodell

Zur Evaluation wurde ein 10cm x 5cm x 2cm großes Blockmodell mit einer DAHS-Schriftzug Extrusion in Fusion 360 entworfen und in PrusaSlicer 2.7 gesliced, siehe Abbildung 5.3. Das Modell benötigt 69,53g Filament. Die Druckzeit auf einem Prusa MINI+ mit InputShaping beträgt ca. 2 Stunden und 25 Minuten. Die Größe des G-Codes nach dem Slicen des Modells beträgt 3.055.373 Byte.

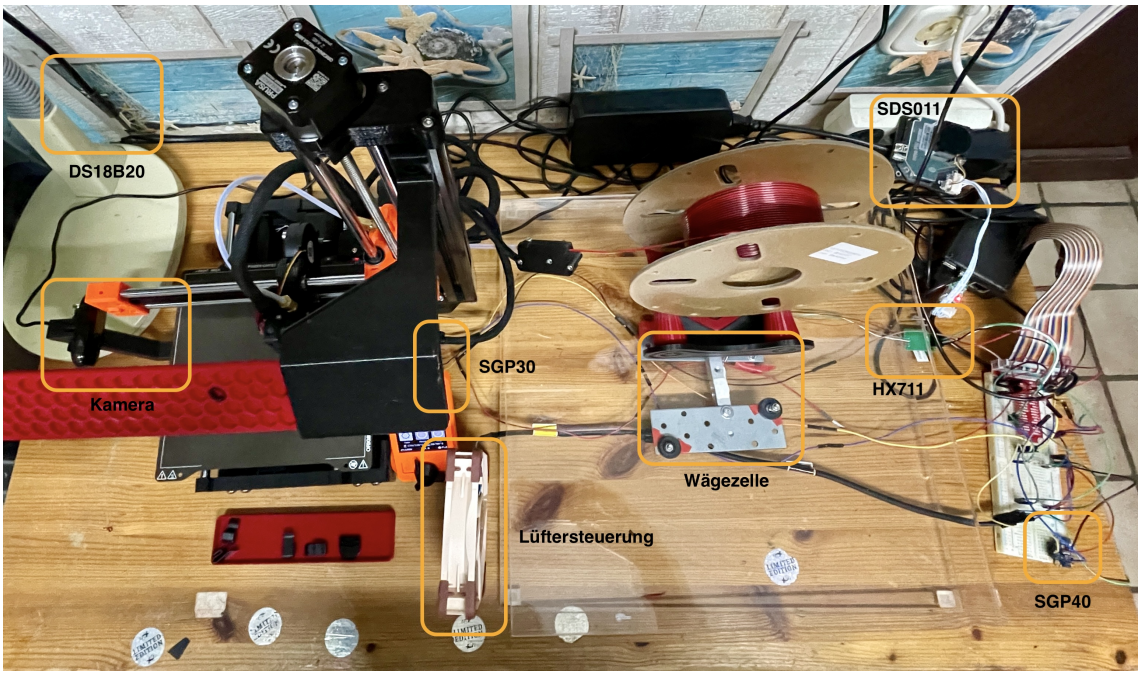

Abbildung 5.1: Vollausgestattete Produktionszelle mit Prusa MINI+ FDM-Drucker, Filament auf Filamentwaage, Sensorik und Aktorik an Raspberry Pi 4 über USB und GPIO-Verlängerung inkl. Breadboard

5.3 Testszenarien

In den folgenden Abschnitten werden die Szenarien zur Evaluation dargelegt. Es wird die Art des Szenarios beschrieben und welche Schritte für dieses durchgeführt werden. Nach der Beschreibung des Szenarios wird die Durchführung mit den Ausgaben des Systems, sowie ggf. der Sensor- und Aktorwerten, dokumentiert und ausgewertet.

Eine Zuordnung der Erfüllung der Anforderungen aus Abschnitt 3.1 zu den zugehörigen Abschnitten in der Evaluation ist in der Tabelle 5.1 zu finden.

5.3.1 Funktions- und Performanztests

In diesen Tests werden Funktionalitäten getestet und die Performanz dieser eingeordnet.

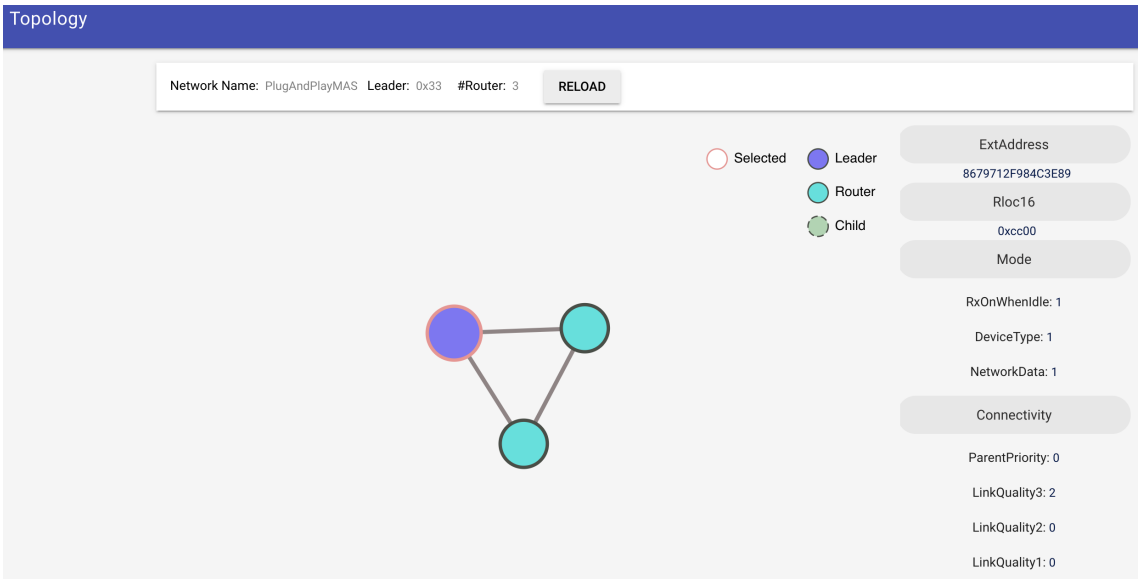

Abbildung 5.2: Visualisierung des aufgebauten Thread-Netzwerkes auf der OpenThread-Borderrouter Oberfläche für die Evaluation, mit drei Thread-Knoten

Verteilung von Agentenkonfigurationen

Zur Bestimmung der durchschnittlichen Verteilungszeit und zur Validierung der fehlerfreien Übertragung, werden In einem Netzwerk bestehend aus 3 DAHS Agentenkonfigurationen hochgeladen. Es werden die Eingangs- und Ausgangskonfigurationen verglichen um die korrekte Übertragung zu validieren und die Übertragungszeit protokolliert.

In 10 Versuchen wurden die Agentenkonfiguration aus Abschnitt A.1 und 2 Testkonfigurationen (Konfigurationen nach dem selben Schema, aber ohne beschriebenes Verhalten, je 232 Bytes Dateigröße) in ein Netzwerk aus 3 DAHS hochgeladen. Es wird die Zeit gemessen, ab dem das erste DAHS die Konfiguration gespeichert hat und mit der Verteilung beginnt, bis die restlichen DAHS die Konfiguration gespeichert haben. In Abbildung 5.4 sind die gemittelten Verteilzeiten über alle 3 DAHS in Millisekunden zu sehen. Die mittlere Verteilungszeit beträgt zwischen 7466ms und 8166ms. Die Agentenkonfigurationen variieren in der Größe von 232 Bytes zu 3148 Bytes. Die reine Netzwerkübertragungszeit dieser Konfigurationen beträgt zwischen 68ms und 2467ms im Thread-Netzwerk. Die höheren Zeiten bei den Mitteln über die DAHS ist durch den Overhead der Serialisierung und Deserialisierung, sowie die Weiterleitung durch die Proxies (siehe Abschnitt3.2) und das Wiederherstellen der Dateien im YAML-Format, zu erklären. Bei der Überprüfung

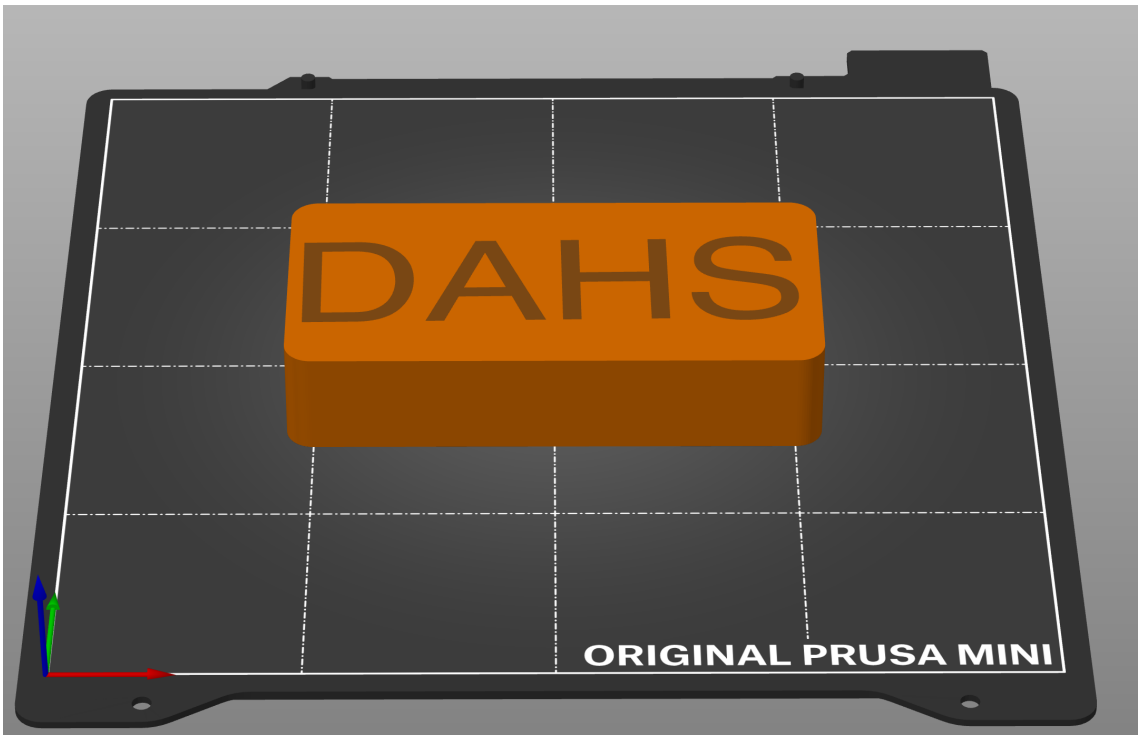

Abbildung 5.3: DAHS-Schriftzug auf Blockmodell im PrusaSlicer 2.7.1

der übertragenen Konfigurationen, wurde festgestellt, dass diese korrekt und ohne Fehler innerhalb der YAML-Syntax wiederhergestellt wurden. Es wurde festgestellt, dass die Konfigurationsgröße im Durchschnitt um 4,27% gestiegen ist, was durch die Wiederherstellung der YAML-Syntax zu erklären ist. Alle Multiline-Strings in den Konfigurationen, welche vor allem zum Schreiben der Agenten-Methoden verwendet werden, sind in Anführungszeichen gesetzt und um die notwendigen Steuerzeichen (Newlines, Carriage Return, etc.) erweitert worden. Durch die Evaluation der Verteilung der Agentenkonfiguration und die positiven Befunde bei der Netzwerkperformanz sowie der korrekten Übertragung, kann die **3. Anforderung** aus Abschnitt 3.1 als erfüllt angesehen werden.

Hardwareerkennung und Agentenbereitstellung

Für jede Agentenkonfiguration, ausgenommen der Webcam, aus dem Abschnitt A.1 wird bei einem laufenden DAHS die Hardware zehn mal angeschlossen und die Zeit zwischen der Erkennung und der Agentenbereitstellung gemessen. Dies soll zeigen, dass die Erken-

Anforderung	Abschnitt in der Evaluation
1	Funktionstest Hardwareerkennung und Agentenbereitstellung: Abschnitt 5.3.1
2	Vorbereitung der Phase 1 der Evaluation: Abschnitt 5.3.3
3	Funktionstest Verteilung von Agentenkonfigurationen: Abschnitt 5.3.1
4	Vorbereitung der Phase 1 der Evaluation: Abschnitt 5.3.3
5	Phase 2 der Evaluation: Abschnitt 5.3.3
6	Phase 2 der Evaluation: Abschnitt 5.3.3
7	Optionale Phase 1 der Evaluation: Abschnitt 5.3.3
8	Phase 1 der Evaluation: Abschnitt 5.3.3

Tabelle 5.1: Zuordnung der Anforderungserfüllung zu Evaluationsabschnitten

nung zuverlässig funktioniert und eine mit zu erwartender Abweichung nahe konstante Agentenbereitstellungszeit erreicht wird. Die Webcam wird ausgenommen, da diese als gekapselte Flask-Applikation die Bereitstellungszeit bei einem Abruf des Webcam-Streams oder Snapshots liefert, welche eine Abweichung zur tatsächlichen Bereitstellungszeit bedeutet. Die notwendigen Python-Packages für die Agentenkonfigurationen sind bei der Durchführung des Tests bereits auf dem System vorhanden. Entsprechend prüft das System mit pip [71] ob diese bereits installiert sind, lädt diese jedoch nicht erneut herunter.

In Abbildung 5.5 ist das Diagramm zu den 10 Versuchen je Agentenkonfiguration zu sehen. Die Bereitstellungszeit variiert zwischen 7271ms und 85451ms. Die Bereitstellungszeit der Gassensor-Agenten, welche I^2C Hardware nutzen, ist signifikant höher, was durch die Abhängigkeit von komplexeren Python-Packages begründet ist. Die Abweichung der Bereitstellungszeit über alle zehn Versuche ist über alle Agentenkonfiguration innerhalb einer Varianz von 0,86% und 11,66%. Aufgrund dieser Ergebnisse, kann die **1. Anforderung** aus dem Abschnitt 3.1 als erfüllt angesehen werden.

Thread-Netzwerkbandbreite

Um die Übertragungszeiten der Agentenkonfigurationen und des G-Codes, für die 3D-Druckaufträge, in Kontext setzen zu können, wird mithilfe des Netzwerkbandbreitenmesswerkzeugs iPerf [16] die durchschnittlich minimale und maximale Netzwerkbandbreite in der Testumgebung gemessen. Zur Durchführung der Messung wird auf den drei Raspberry Pi, die für das Evaluationsszenario genutzt werden, die neuste Version von iPerf3 installiert. Die Raspberry Pi werden fünf mal als Client und fünf mal als Server genutzt und in jeder Kombination zueinander. Die Netzwerkbandbreite wird vom Client 120 Sekunden

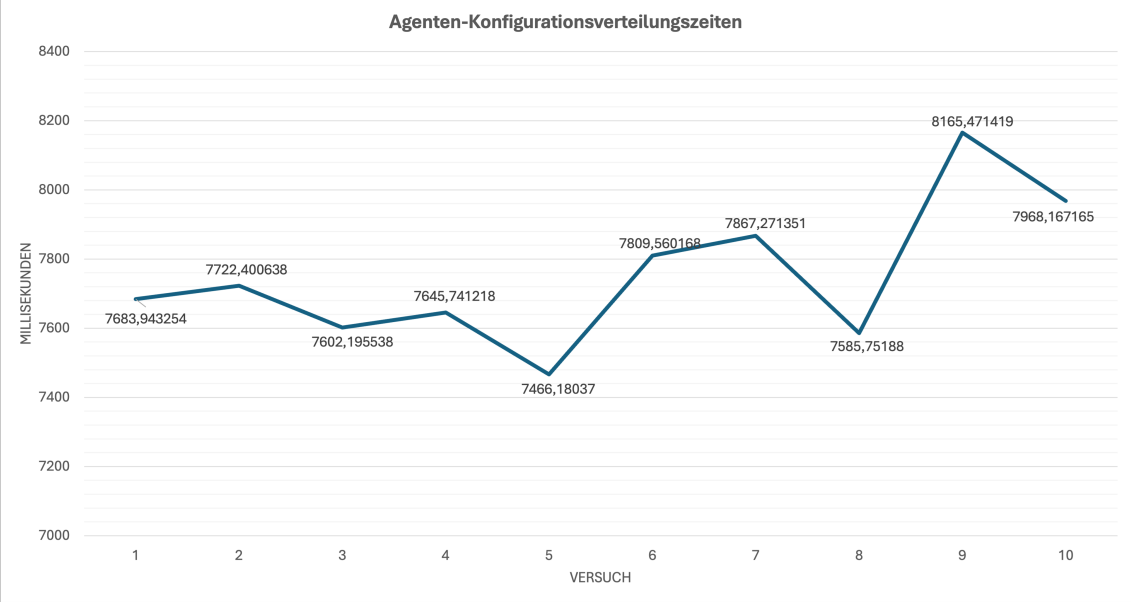

Abbildung 5.4: Durchschnittswerte der Verteilungszeiten von Agentenkonfigurationen (siehe Abschnitt A.1) an 3 DAHS über 10 Versuche

lang getestet und dann dokumentiert. Der Server mit dem Befehl und Schaltern iperf -s -V -B [fd65:f8ec:b989:1:9ac:dd2e:3073:5520] gestartet, respektive mit der IPv6-Adresse des wpan0 Interfaces, und der Client verbindet sich mit dem Server mit folgenden Befehl und Schaltern iperf -t 120 -V -c fd65:f8ec:b989:1:9ac:dd2e:3073:5520. Die Durchführung der Tests hat eine minimale durchschnittliche Netzwerkbandbreite von 65,9 KBit/s, respektive 8,125KB/s und eine maximale von 109,3 KBit/s, respektive 13,6625KB/s ergeben.

5.3.2 Tests nach Evaluationsszenario

In diesen Tests wird das System nach einem beschriebenen Szenario evaluiert und die Ergebnisse für die Diskussion im Abschnitt 6.2 aufbereitet.

Beschreibung des Evaluationsszenarios

Folgendes Szenario wurde entwickelt, um das System nach den beschriebenen Anforderungen aus Abschnitt 3.1 zu evaluieren. Dabei zeigt das Szenario eine Anwendung für das DAHS durch die einzelnen Arbeitszellen, welche jeweils verschiedene Hardware besitzen

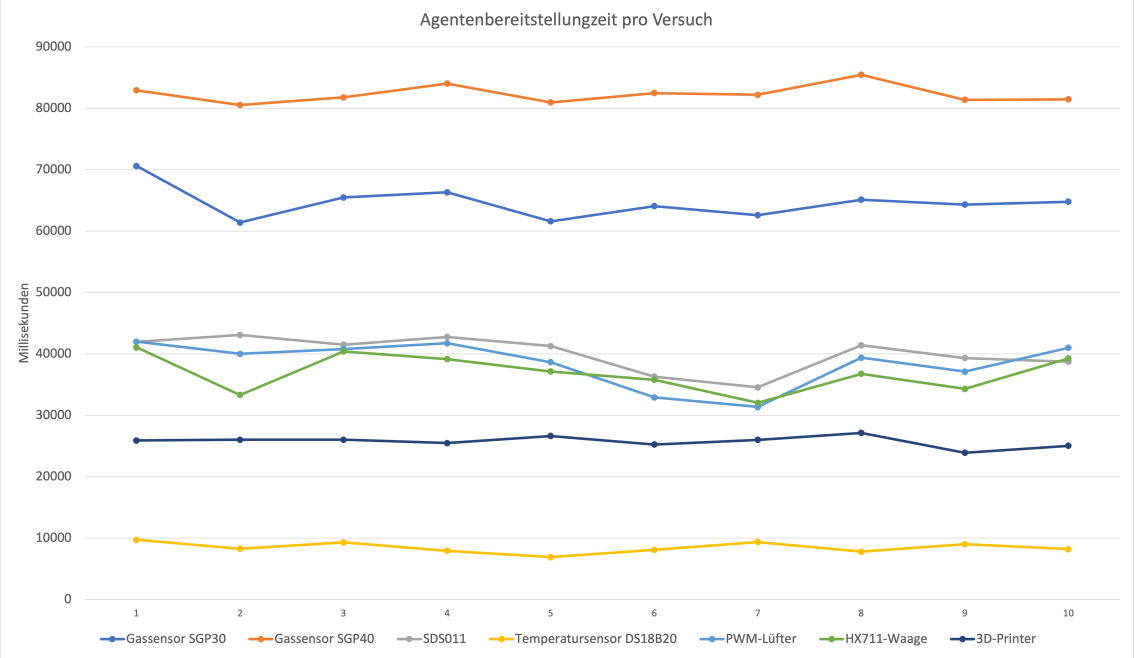

Abbildung 5.5: Bereitstellungszeiten von Agenten in Millisekunden aller Agentenkonfigurationen (siehe Abschnitt A.1), ausgenommen der Webcam, von der Hardwareerkennung bis zur ersten Bereitstellung von Daten des zugehörigen Agenten, über zehn Versuche

können, aber bei Bedarf auf Hardwaredaten einer anderen Zelle zugreifen könnten. Durch die konfigurierbaren Plug and Produce Agents sind die Zellen dynamisch um Hardware erweiterbar und erhalten für komplett neue Hardware die aktuellste Agentenkonfigurationen, ohne das eine zentrale Komponente notwendig ist. Des Weiteren zeigt es auch, durch den modularen Aufbau dieser Zellen, wie das System es ermöglicht schnell und unkompliziert eine neue Zelle hinzuzufügen.

In einem Maker-Space mit neusten Technologien kommen FDM und SLS/SLA 3D-Drucker zum Einsatz, welche höhere Sicherheitsanforderungen für die Beschäftigten haben, aufgrund der entstehenden VOCs beim Druckprozess und durch entweichende Dämpfe der genutzten Chemikalien [80]. Für dieses Szenario wird das DAHS mit dem minimalen MAS und den Plug and Produce Agents eingesetzt um die Sicherheit der Beschäftigten zu gewährleisten, eine optimale Auslastung der Maschinen zu realisieren, eine Überlastung zu verhindern und das ganze System unkompliziert erweiterbar und wartbar zu machen.

Dazu wird ein Maker-Space in einem Raster eingeteilt. In einem Abteil des Rasters können sich eine limitierte Anzahl von Maschinen befinden, welche zu einer Gruppe zusammengefasst werden (auf der Anwendungsebene und im OPC UA Organisationsmodell, siehe Abbildung 3.7). Die Maschinen werden pro Raster an mindestens einem SBC-Thread-Knoten angeschlossen.

Ausgangskonfiguration

In der Ausgangskonfiguration sind 3 Zellen vorhanden. 2 Zellen sind offline. Die verfügbare Zelle hat einen 3D-Drucker angeschlossen und der entsprechende PrintAgent bereits 4 Aufträge angenommen, wovon einer in Bearbeitung ist und 3 sich in der Warteschlange befinden. Die Zelle hat keine angeschlossenen Sensoren und keine angeschlossene Lüftersteuerung. Die Temperatur steigt in der Zelle auf 41-42$°C$. Der VOC-Index in der Zelle erreicht einen Wert von > 100. Von außerhalb des Thread-Netzwerkes werden neue Aufträge an die Zelle gestellt, welche aber aufgrund der bereits vollen Warteschlange abgelehnt werden.

Optimale Konfiguration

Aus der Ausgangskonfiguration wird in mehreren Phasen neue Sensorik/Aktorik und Zellen hin zugeschaltet, um Messdaten aufzunehmen, während das MAS in einen optimalen Zustand übergeht.

1. Phase: In der verfügbaren Zelle werden ein Temperatur- und Gassensor angeschlossen. Danach kann in den Pub/Sub-Channeln festgestellt werden, dass die zugehörigen Agenten bekanntgemacht werden und in den Debug-Logs des DAHS Temperatur- und Luftanalysedaten zu sehen sind.

2. Phase: Die Lüftersteuerung wird als Aktor angeschlossen. Es kann wieder im Pub/Sub-Channel festgestellt werden, dass ein neuer Agent instanziiert wurde und dass die Lüftersteuerung mit der Klimatisierung der Zelle beginnt. In den Debug-Logs der Zelle ist zu sehen, wie die Temperatur und die VOC-Konzentration abnehmen. Nachdem die Temperatur wieder im Bereich von 30$°C$ bis 40$°C$ ist und die VOC-Konzentration in einem nicht messbaren Bereich ist, schaltet die Lüftersteuerung die Klimatisierung runter/aus.

3. Phase: Die 2 zusätzlichen Zellen werden dazugeschaltet, diese werden mit bereits angeschlossenen Sensoren/Aktoren aktiviert. In den Netzwerk-Logs ist zu sehen, wie sich die SBC der Zellen in das Thread-Netzwerk eingliedern. Im Pub/Sub-Channel ist zu sehen, wie die Agenten der hinzugekommenen Zellen bekanntgemacht werden.

4. Phase: Es werden weitere Aufträge an das MAS gesendet, bis alle Auftrags- und Warteschlangenplätze voll sind. Während des senden von Aufträgen ist in den Logs der Management-Agenten ersichtlich wie das MAS gleichmäßig die Aufträge verteilt.

Optionale Phasen: Zur Überprüfung der optimalen Konfiguration können zusätzlich der Ausfall von Zellen, Sensoren, Aktoren und Agenten evaluiert werden. Dabei verhält sich das System ähnlich zu den bereits beschriebenen Phasen, jedoch mit dem Unterschied das einige Korrektur Methodiken in Gang gesetzt werden um einen Ausfall zu detektieren und danach zu mitigieren. Des Weiteren gibt es auch Plausibilitätsprüfungen der Sensorwerte, welche evaluiert werden können, wenn mehrere Sensoren der gleichen Art in einer Zelle zur Verfügung stehen.

1. optionale Phase: Ein Sensor/Aktor fällt oder löst einen Fehler aus. Dies wird vom DAHS detektiert und daraufhin an den entsprechenden Management-Agenten weitergegeben. Das DAHS versucht eine Fehlerbehandlung durchzuführen oder beendet die verwaisten Agenten daraufhin.

2. optionale Phase: Es werden zwei oder mehr Sensoren der gleichen Art in einer Gruppe angeschlossen. Der Management-Agent beginnt mit der Plausibilitätsprüfung. Falls der Sensorwert eines Agenten mehr als die Genauigkeit + 10% (in der Standardkonfiguration) abweicht, wird dies vom Management-Agenten erkannt. Der Management-Agent gibt bekannt welche Agenten die Prüfung nicht bestanden haben und entfernt diese aus der Liste zu berücksichtigender Werte für die weitere Verarbeitung.

5.3.3 Evaluationsdurchführung

Zur Evaluation wird das System auf Single-Board-Computern mit Thread fähigen Radio-Co-Processors [23] wie in 5.1 beschrieben aufgebaut. Die Sensorwerte werden in den OPC UA Servern protokolliert und der Systemaufbau wird mit einem OPC UA Client außerhalb des Thread-Netzwerkes visualisiert. Die Auslastungskontrolle wird durch die

Protokolle der MangementAgenten und PrintAgenten evaluiert. Für die Testdrucke wird das in 5.3 gezeigte Modell genutzt und in ABS gedruckt.

Vorbereitung

Zur Vorbereitung der Ausgangskonfiguration wurden fünf 3D-Drucke mit ABS manuell durchgeführt und der Raum (siehe 5.1) aufgeheizt. Es wurde ein DAHS auf einem Raspberry Pi 4 (4GB RAM) gestartet, der nur die Agentenkonfiguration für den 3D-Drucker (Prusa MINI+) enthält. Während des Startens durch das infrastructureSetup, war im Log zu sehen, dass die Komponenten korrekt auf dem wpan0 Interface mit der IPv6-Adresse fd65:f8ec:b989:1:a421:a9bc:9e04:ac04 gebunden wurden, was die **4. Anforderung** aus dem Abschnitt 3.1 erfüllt. Nach dem Start des DAHS wurde eine Verbindung mit der DAHS-UI zum System aufgebaut und das leere Topic abonniert um alle Nachrichten zu empfangen. Mit dem Unified Automation UaExpert wurde eine Verbindung zum OPC UA Server des Systems aufgebaut um dieses parallel zu überwachen. Im DAHS-UI wurden nacheinander die Agentenkonfigurationen zum NewAgentConfig-Topic publiziert. Im Debug-Log des DAHS war zu sehen, wie die Agentenkonfigurationen empfangen und als YAML abgespeichert wurden. Der Ausgabe der DAHS-UI war zu entnehmen, dass der ManagementAgent eine Abklingzeit (WaitingTime-Topic) von 1800 Sekunden publiziert. Im nächsten Schritt wurde die Filamentwaage angeschlossen, was dazu führte dass im Log des DAHS die Meldungen zum aktualisierten AgentenDictionary zu sehen waren, dies ist bei jedem neuen Agenten der Fall und wird nicht wiederholt aufgeführt, und darauf die Meldungen, dass die Installation der benötigten Packages abgeschlossen ist. In der Ausgabe des DAHS-UI war zu sehen dass der ManagementAgent eine Abklingzeit von 1200 Sekunden publiziert und das der neue Agent, der die Filamentwaage repräsentiert, ein Filamentgewicht von 966.52 Gramm publiziert. Dies zeigt, dass die **2. Anforderung** erfüllt ist, da der Agent basierend auf der vorher hochgeladenen Konfiguration erstellt wurde. Im Log des PrintAgents wurden Fehler angezeigt, dass die Variablen im OPC UA Server nicht aktualisiert werden konnten, da keine Daten von Octoprint empfangen wurden. Da diese Fehler aber abgefangen wurden, lief der PrintAgent weiter und aktualisierte die Daten in OPC UA in den nächsten Iterationen.

Phase 1

Als nächstes wurden die SGP30 und SGP40 Gassensoren, sowie ein DS18B20 OneWire Temperatursensor angeschlossen. Im Log des DAHS waren wieder die Instanziierungen zu beobachten, jedoch dauerten diese, wie erwartet, für die Gassensoren über eine Minute. Im DAHS-UI war zu sehen, dass der SGP30-Agent eine eC02 Belastung von 400ppm und einen TVOC-Wert von 5.5ppb publizierte. Der SGP40-Agent publizierte einen VOC-Index von 330 und der DS18B20-Agent eine Temperatur von 38.77°C. In der Ausgabe des DAHS-UI war nun zu sehen, dass der ManagementAgent eine Abklingzeit von 348 Sekunden publizierte. Im nächsten Schritt wurde der SDS011 Laser PM2.5/PM10 Sensor angeschlossen. Im DAHS Log war wieder die Instanziierung und im DAHS-UI waren die publizierten Werte des SDS011 zu sehen. Der PM2.5 Wert betrug 36 $\mu g/m^3$ und der PM10 Wert 68 $\mu g/m^3$. Der ManagementAgent publizierte eine Abklingzeit von 324 Sekunden. Die weitere Anpassung der Abklingzeit, zeigt auch, dass die **8. Anforderung** erfüllt ist, da die Systemperformanz adaptiv angepasst wurde.

Phase 2

Als nächstes wurde der Noctua NF-A12x25 5V PWM Lüfter angeschlossen. Im DAHS Log war diese Instanziierung auch wieder zu sehen und im DAHS-UI war zu sehen, dass der Lüfter direkt mit 1566 RPM gestartet wurde. Als letztes wurde die Logitech C170 Webcam angeschlossen. Im Log war die Instanziierung zu sehen und danach die Ausgabe des Flask-Servers, dass dieser nun auf allen Netzwerkinterfaces läuft. Auf der Oberfläche von Octoprint war nun auch der Stream der Webcam zu sehen. Wenn eine der Interface-IPs mit dem Port 7777 und /snapshot aufgerufen wurde, konnte ein Snaptshot des Streams abgerufen werden. Nachdem die erste Zelle mit aller Hardware ausgestattet wurde und diese stabil lief, wurden mithilfe des DAHS-UI 3D-Druckaufträge an die Zelle gesandt. Dafür wurde der in Abschnitt 5.2 beschriebene G-Code verwendet. Nach dem ersten Senden des G-Codes wurde beobachtet, wie der Prusa MINI+ direkt anfing diesen Auftrag zu bearbeiten. Im Log des ManagementAgenten war auch zu sehen, dass der Auftrag direkt weitergegeben wurde, da ausreichend Filament auf der Filamentwaage zur Verfügung stand. Im DAHS-UI war auch zu beobachten, dass auf dem UI-Interaction Topic die Nachricht zur Annahme des Auftrages empfangen wurde. Mit dem DAHS-UI wurde der gleiche G-Code weitere fünf Male an das DAHS gesendet. Vier weitere

Male wurde die Annahme erfolgreich bestätigt und nach dem fünften Senden wurde eine Benachrichtigung über die bereits volle Warteschlange über das UI-Interaction-Topic empfangen. Während aller Schritte konnte im UaExpert beobachtet werden wie die Gruppe des DAHS neue Rollen und darin enthaltene Agenten erhält. Der MemberCount der Gruppe stieg auch mit jedem neuen bereitgestellten Agenten an. Die Werte der Agenten waren in der Struktur auffindbar und konnten zur Überwachung abonniert werden. Dies bestätigt auch die Erfüllung der **5. und 6. Anforderung** aus dem Abschnitt 3.1.

Phase 3

Um die Umverteilung der 3D-Druckaufträge beginnen zu können, wurden zwei weitere Zellen bereitgestellt. Dazu wurde ein weiterer Raspberry Pi 4 (4GB) und ein Raspberry Pi 2 (1GB) gestartet und gewartet, bis diese sich im Thread-Netzwerk eingegliedert haben (überprüft wurde dies mit der OpenThread-Übersichtsseite, die jeder Borderrouter enthält). Beide dieser SBCs hatten bereits einen DS18B20 OneWire Temperatursensor und SGP30 Gassensor angeschlossen, enthielten aber noch keine Agentenkonfigurationen. Nach dem Starten der DAHS auf den SBCs ist in allen Logs der DAHS zu sehen gewesen, dass diese sich über das Thread-Netzwerk gefunden haben und ein ZMQ Socket zum Forwarding der Nachrichten erstellt wurde. Danach waren auch bereits Nachrichten aus den beiden neuen DAHS im DAHS-UI zu sehen. Zu sehen waren auch die Nachrichten an das NewDAHSarrived-Topic, was in Folge das erste DAHS veranlasste alle verfügbaren Agentenkonfigurationen zu publizieren. in den Logs der beiden neuen DAHS war zu beobachten wie diese die neuen Agentenkonfigurationen empfingen und diese abspeicherten. Danach war direkt die Initialisierung der Agenten zu der bereits angeschlossenen Hardware zu sehen.

Phase 4

Mit dem DAHS-UI wurde an das Topic bestehend aus der Gruppen-ID kombiniert mit ToggleDebug eine Nachricht publiziert um den Debug-Modus in beiden neuen DAHS ManagementAgenten einzuschalten. Dies war notwendig um Drucke mit den virtuellen 3D-Druckern in Octoprint vorzunehmen und die Notwendigkeit der Filamentwaage auszuschalten. Im nächsten Schritt wurden weitere Druckaufträge an das erste DAHS gesendet, das weiterhin eine volle Druckwarteschlange besaß, da der Druck ca. 2,5 Stunden dauert. Im Log des ManagementAgenten war zu sehen, dass die volle Warteschlange vom

PrintAgent publiziert wurde und der ManagementAgent begann nach anderen Drucker im Netzwerk zu suchen. Der erste weitere Druckauftrag wurde vom PrintAgent des zweiten Raspberry Pi 4 angenommen und der darauf folgende vom Raspberry Pi 2. Damit wurde auch gezeigt, dass die Gleichverteilung der Druckaufträge funktioniert, da immer der PrintAgent mit der kleinsten Warteschlange den Auftrag zugeteilt bekam. Die Übertragung der Druckaufträge an die entfernten PrintAgents hat zwischen 198 und 314 Sekunden gedauert.

Optionale Phase 1

Zur Evaluation der ersten optionalen Phase wurde die Verbindung des SGP30-Sensors zum Raspberry Pi 2 einige Male getrennt und wiederhergestellt, was einen CRC-Fehler in einem der vom Agenten verwendeten Packages verursacht und diesen Agenten zum Absturz bringt. Im Log des DAHS ist der Absturz erkennbar und der Versuch den Agenten wiederherzustellen. Da der Agent durch den CRC-Fehler in einem nicht rekonstruierbaren Zustand war, ist zu sehen, wie das DAHS einen neuen Agenten mit der SGP30 Konfiguration erstellt. Der ausgelöste Ausfall und die beobachtete Fehlerbehebung zeigt, dass das System robust gegen den Ausfall einzelner Komponenten ist und damit die **7. Anforderung** aus dem Abschnitt 3.1 erfüllt ist.

Optionale Phase 2

Zur Evaluation der zweiten optionalen Phase wurden im vollständig bestückten DAHS zwei weitere DS18B20 OneWire Temperatursensoren angeschlossen. In den Logs des DAHS waren wieder die Instanziierung der zugehörigen Agenten zu sehen und im UaExpert die in OPC UA hinzugefügten Sensoren mit ihren Werten. Daraufhin wurde einer der Sensoren mit einem Heißluftföhn erhitzt. Im DAHS-UI war zu sehen, dass der Agent dieses Sensors eine Temepratur von 76,33°C publiziert hat. Der ManagementAgent publizierte auf dem Gruppen-eigenen plausibilityFailed-Topic eine Liste, mit der Agenten-ID des Agenten als Inhalt, der vorher 76,33°C publizierte.

6 Fazit und Ausblick

In diesem Kapitel werden die Ergebnisse aus der Evaluation diskutiert und eingeordnet. Basierend darauf wird ein Fazit gezogen und Weiterentwicklungsmöglichkeiten des DAHS und der Plug and Produce Agents diskutiert und ein Ausblick auf weitere mögliche Anwendungsmöglichkeiten gegeben.

6.1 Fazit

In dieser Arbeit wurde das DAHS und die Plug and Produce Agents in einer Proof-of-Concept Version fertiggestellt und evaluiert. Die in der Zielsetzung definierten Herausforderungen sind umgesetzt durch die Evaluation des Systems belegt. In der Diskussion der Evaluation ist dargelegt, wie das definierte OPC UA Datenmodell sich erwiesen hat, das die Verteilung der Agentenkonfigurationen ohne eine zentrale Komponenten funktioniert und neue Konfigurationen zuverlässig verteilt werden und dass die Erkennung neuangeschlossener Hardware, und eine darauf basierende Agenteninstanziierung, funktional ist. Der Aufbau des Systems auf einem Thread-Netzwerk hat sich als funktional bewiesen, und gezeigt, dass auch Anwendungen aus dem semi-industriellen Sektor Vorteile aus diesem Protokoll ziehen können. In der Analyse des Nachrichtenaufkommens und die Auswirkung von diesem im Kontext der limitieren Bandbreite eines Thread-Netzwerkes, zeigt aber auch, dass bei der Übertragung von größeren Dateien über Kompression und weitere Mitigationsmechanismen nachgedacht werden muss um eine Überlastung des Netzwerkes vorzubeugen. Die in der Weiterentwicklung diskutierten Möglichkeiten zeigen, dass das entwickelte System auch Potenzial über das definierte Evaluationsszenario hinaus besitzt. Besonders die Weiterentwicklungsmöglichkeiten im Kontext von OPC UA bietet Potenzial. Das DAHS hat sich als Komponente zur Verwaltung von Agenten bewährt und bietet aufgrund der Implementierung auf dem minimalen MAS das Potenzial auf weitere MAS adaptiert zu werden. Die entwickelte und in der Evaluation benutzte Benutzeroberfläche zur Interaktion mit dem System hat sich bewährt und kann, über die Grenzen

dieser Arbeit hinweg, als simples Werkzeug zur Abfrage eines OPC UA Servers und zur Interaktion über ZMQ mit anderen Systemen verwendet werden.

6.2 Diskussion der Resultate der Evaluation

Die Evaluation zeigt, dass der realisierte Proof-of-Concept-Build, die definierten Anforderungen aus Abschnitt 3.1 erfüllt. Die Erkennung von neuer Hardware und die darauf basierende Agenteninstanziierung erweist sich als robust und zuverlässig. Es sind noch Nacharbeiten in der Erkennung von GPIO-basierter Hardware notwendig. Konfigurationen welche nur GPIO-Pins im "Low"-Status erwarten, können zu einer Agenteninstanziierung führen, ohne das Hardware angeschlossen wird, da die GPIO-Pins in der Initialisierung des DAHS alle in den "Low"-Status gebracht werden. Der Austausch von Nachrichten über das Thread-Netzwerk funktioniert stabil. Die darauf basierende Synchronisierung der DAHS ist bis auf den initialen Konfigurationsaustausch fehlerfrei und alle DAHS haben stets eine aktuelle Übersicht der verfügbaren Agenten im Netzwerk. Der Konfigurationsaustausch kann durch einen intervallbasierten Ansatz erweitert werden, damit sichergestellt ist, dass jedes DAHS immer die aktuellste Konfigurationsversion erhält. Die Umverteilung von Druckaufträgen mit einer Gleichverteilung über alle verfügbaren DAHS mit aktiven PrintAgents ist funktional. Lediglich die Übertragung des G-Codes dauert aufgrund der geringen Bandbreite des Thread-Netzwerkes mit 198-314 Sekunden länger als initial erwartet. Zur Mitigation sollten die hochgeladenen G-Code Dateien vor der Übertragung komprimiert werden. Das entwickelte und implementierte OPC UA Organisations- bzw. Informationsmodell hat sich als gut strukturierte Basis erwiesen, das wie in Abschnitt 6.3.3 beschrieben weiterentwickelt und verbessert werden kann. Die Plausibilitätsprüfung bei mehr als zwei Sensoren einer Art in einer Gruppe ist funktional. Damit diese auch bei zwei Sensoren einer Art korrekt angewendet werden kann, sollte diese um weitere Validierungsmechanismen erweitert werden, wie einer Erkennung eines unregelmäßigen Anstiegs oder Abfalls der Sensorwerte.

Die durchgeführten Funktions- und Performanztests haben gezeigt, dass die Instanziierung der verschiedenen Agenten in einem kontrollierten Umfeld in eine invariante Bereitstellungszeit aufweist und die Übertragung der Agentenkonfigurationen in einem erwartbaren Zeitrahmen in einem Thread-Netzwerk verteilt werden. Die Überprüfung der Netzwerkbandbreite des Thread-Netzwerkes hat gezeigt, dass die Übertragungszeiten der Druckaufträge nicht durch einen Fehler verursacht sind.

6.2.1 Nachrichtenaufkommen

In der Evaluation hat sich gezeigt, dass bei einer vollen DAHS Produktionszelle im Durchschnitt 3,48 ZMQ-Nachrichten per Sekunde erzeugt werden. Durch die hinzukommenden DAHS mit weniger Hardware ist ein Anstieg um 1,7 ZMQ-Nachrichten pro Sekunde auf 5,18 und respektive auf 6,92 Nachrichten pro Sekunde zu beobachten. Diese Werte sind für einen produktiven Betrieb im Thread-Netzwerk als unproblematisch einzustufen, aufgrund der zur Verfügung stehenden Bandbreite von bis zu 13KB/s (siehe Abschnitt 5.3.1). Zu beachten ist jedoch, dass durch eine höhere Anzahl von DAHS und somit einem höheren Nachrichtenaufkommen, auch die beobachteten Übertragungszeiten von größeren Dateien, wie den G-Code-Dateien für die Druckaufträge, weiter anwachsen werden und damit die Gleichverteilung der Druckaufträge verlangsamen.

6.3 Weiterentwicklungsmöglichkeiten

6.3.1 Tiefere Thread-Implementierung

Das DAHS basiert in der momentanen Implementierung auf einem Netzwerkstack-agnostischem Design. Das Thread-Protokoll wird zur Erfüllung der Robustheitsanforderungen genutzt. Das DAHS könnte auch mit anderen Netzwerkprotokollen verwendet werden, wobei jedoch einige Komponenten ihre Robustheit verlieren würden. In einer weiteren Entwicklungsiteration sollte dieser Ansatz aufgegeben werden und das Thread-Protokoll tiefer in der Anwendungsebene integriert werden. Somit könnte das DAHS auch beeinflussen, welcher Thread-Modus eingenommen wird. Als Beispiel könnte bei einem akkubetriebenen SBC, auf dem das DAHS eingesetzt werden soll, das DAHS den Child-Modus konfigurieren. Des Weiteren könnten auch die verfügbaren Metriken aus dem Thread-Netzwerk im DAHS verwendet werden, um z.B. den Management-Agenten mitzuteilen welche Gruppe sich näher an der eigenen befindet um Informationen von nahen Agenten bei einem Hardware-Ausfall als Ersatz nutzen zu können.

6.3.2 Agentenkonfigurationen

Die Agentenkonfigurationen werden weder nach dem Hochladen noch beim Verteilen auf Vollständigkeit oder Fehler überprüft. Des Weiteren ist auch das Einschleusen von schadhaften Code möglich, da die YAML in die Agenten eingelesen werden und die Methoden

zur Laufzeit kompiliert und ausgeführt werden. Es sollte ein Validierungsmechanismus und eine Verschlüsselung für die Agentenkonfigurationen implementiert werden.

6.3.3 OPC UA

Sicherheitsaspekte

In der Realisierung des Systems wird eine simple Implementierung eines OPC UA Servers genutzt, welche keine Zertifikate nutzt. Des Weiteren nutzen die implementierten OPC UA Clients, dies umfasst die Agenten und das DAHS, den Standard Admin-Benutzer des OPC UA Servers um mit diesem zu interagieren. Es sollten eigene Benutzer für die verschiedenen Rollen der Agenten und des DAHS angelegt werden und ein Berechtigungskonzept entwickelt werden, basierend auf den definierten OPC UA Security Standards [48]. Für einen produktiven Betrieb sollte auch über die Verwendung eines Zertifikats für jeden OPC UA Server nachgedacht werden.

OPC UA FileDirectory und OPC UA FileType

Um die Implementierung weiter zu standardisieren, können für die Übertragung der Agentenkonfigurationen die OPC UA ObjectTypes, FileDirectoryType [46] und FileType [47] genutzt werden. Dies würde die Methoden der Verteilung der Agentenkonfigurationen vereinfachen und wartbarer gestalten.

OPC UA Machinevision

Die realisierte Agentenkonfiguration für die Webcam stellt momentan einen Stream und eine Snapshot-Funktionalität bereit. Im OPC UA Server wird die URL zum Stream und zum Snapshot vom Agenten abgelegt und in Octoprint wird der Stream zur Verfügung zu gestellt und mit den Snapshots automatische Zeitrafferaufnahmen der 3D-Drucke angefertigt. Zur Weiterentwicklung, sollten die Möglichkeiten der OPC UA Machinevision [44] Definition genutzt werden um die Streams und Snapshots dort bereitzustellen und die Möglichkeit zu schaffen, diese weiterzuverarbeiten, wie z.B. eine Erkennung ob ein 3D-Druck gescheitert ist.

Erkennung neuer Hardware - OPC UA Funktionalität

OPC UA bietet mit den ”Discovery and Global Services” [39] eine Funktionalität zur Suche nach OPC UA Servern und Clients. Zusätzlich zu der im Abschnitt 3.5.2 beschriebenen und implementierten Methode zur Erkennung von angeschlossener Hardware, kann mit diesen Funktionalitäten eine Methode entwickelt werden, um neue Hardware auf IP-fähigen Hosts zu finden. Es könnten beispielsweise batteriebetriebene Thread-Knoten ohne ein eigenes DAHS, mit Sensorik ausgestattet werden und lokal ihre Werte in einem OPC UA Server bereitstellen. Diese Knoten könnten über die neue Funktionalität an das DAHS angebunden und basierend auf den OPC UA Informationen Agenten instanziiert werden. Dies würde durch die Implementierung von OPC UA Standards eine weitere Generalisierung des DAHS bedeuten und neue Anwendungsmöglichkeiten eröffnen.

Organisationsmodell

Das im Abschnitt 3.7 entwickelte Informationsmodell, basierend auf dem in [68] vorgestellten Modell und allgemein auf dem AGR-Modell [19] sollte funktional mit den Agentengruppen verbunden werden, damit die ZMQ Kommunikation auch nach diesem Schemata funktioniert.

Verbesserung der OPC UA Node Abfrage

Die OPC UA Clients nutzen in der jetzigen Implementierung eine simple Iteration über die Parent- oder Child-Nodes vom eigenen OPC UA Node aus, um einen gesuchten OPC UA Node zu finden. Dies sollte durch eine performantere Suchmethode ersetzt werden, wie der Abfrage über die NodeID, einer Kombination aus dem NodeIdentifier und dem NamespaceIndex.

6.3.4 Reimplementierung der Agenten mit Multiprocessing

Im Abschnitt 4.1 wird diskutiert, warum die Plug and Produce Agents und die weiteren Komponenten als Threads implementiert wurden. Die bessere I/O-Performanz bleibt auch nach der Evaluation ein bestehender Vorteil, jedoch hat diese Implementierung Nachteile in der Fehlerbehebung bei einem Absturz eines Agenten. Dies hat sich in der Evaluation beim Testen der Fehlerbehebung herausgestellt, da zwar ein Agent entfernt

und basierend auf der vorigen Konfiguration neu instanziiert werden kann, aber die bestehenden OPC UA Referenzen des Agenten nicht mehr vorhanden sind und entsprechend das Entfernen dieser aus dem OPC UA Server kompliziert macht. Deshalb sollte in einer Weiterentwicklung des Systems eine Reimplementierung der Plug and Produce Agents mit Multiprocessing anstatt von Threading vorgenommen werden. Dies verspricht eine einfachere Fehlerbehebung, da bei einem Fehler, eine Fehlerbehebung im übergeordneten Prozess vorgenommen werden kann. Des Weiteren kann auch eine höhere Performanz aufgrund der besseren Nebenläufigkeit erreicht werden, da die Prozesse nicht durch das Global Interpreter Lock (GIL) beschränkt werden, sondern jeweils einen eigenen Interpreter nutzen.

6.4 Generische Plug and Produce Agents

In der vorgestellten Implementierung sind der ManagementAgent und der PrintAgent Ableitungen von der Agenten-Basisklasse. Zum Aufzeigen der verschiedenen Möglichkeiten, wie ein Agent erstellt werden kann, haben diese sich als gute Beispiele herausgestellt. Diese sollten aber zur Vereinheitlichung und weiteren Generalisierung auch als Agentenkonfigurationen realisiert werden, damit das DAHS und die Plug and Produce Agents auch für weitere Anwendungen simpel adaptiert werden können. Insbesondere der ManagementAgent würde von dieser Maßnahme profitieren, da dieser auch bei anderen Anwendungen weiter als Gruppenleiter eingesetzt werden kann und die benötigten Management-Methoden leicht in eine eigene Konfiguration eingefügt werden können.

6.5 Weitere Anwendungen des DAHS und der Plug and Produce Agents

Die Plug and Produce Agents und das DAHS wurden für diese Arbeit speziell für das Evaluationsszenario entworfen und umgesetzt. Aufgrund der Grundlage auf einem für das DAHS entworfenen minimalen MAS, bietet sich das System nach der vorig beschriebenen Generalisierung auch für weitere Anwendungen an. Mit einer Erweiterung des minimalen MAS um weitere Features, wie dem Übertragen von Agenten von DAHS zu einem anderen und der Möglichkeit einer entfernten Instanziierung von Agenten, könnte ein vollwertiges MAS realisiert werden. Des Weiteren könnte ein eigenes Hardwareprotokoll entwickelt

werden um eine für das DAHS standardisierte Hardwareerkennung zu ermöglichen. Auf dieser Basis könnten bestehende Plug and Produce Anwendungen auf dem DAHS-MAS adaptiert und evaluiert werden.

Eine weitere Möglichkeit ist die Adaption des DAHS auf weitere MAS. Das DAHS hat aufgrund des Aufbaus auf einem minimalen MAS nur limitierte Abhängigkeiten. Es benötigt eine Messaging-Implementierung zur Interaktion mit Agenten, eine Logik die zur Instanziierung für Agenten genutzt wird, Informationen über die Laufzeitumgebung und die Möglichkeit instanziierte Agenten zu kontrollieren. Bei Erfüllung dieser Voraussetzungen, kann das DAHS adaptiert werden. Da ein MAS nach Definition einen Großteil dieser Voraussetzungen erfüllt, wie im Abschnitt 2.4 beschrieben, kann eine Adaption auf einer Vielzahl von MAS erfolgen. Dabei kann das DAHS wie in dieser Arbeit, mit mehreren Instanzen in einem Netzwerk verteilt genutzt werden oder als zentralisierte Systemkomponente.

Literaturverzeichnis

[1] IEEE Standard for Low-Rate Wireless Networks. In: *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)* (2020), July, S. 1–800

[2] Abdalla, Jamal: Elements of an Agent-based Mediative Communication Protocol for Design Objects. (2004)

[3] Actoron GmbH: *Jadex Active Components Framework*. 2021. – URL https://www.activecomponents.org/index.html#/docs/overview. – Zugriffsdatum: 2023-12-26

[4] Adafruit: *Add CircuitPython hardware API and libraries to MicroPython and CPython devices*. – URL https://github.com/adafruit/Adafruit_Blinka. – Zugriffsdatum: 2024-03-12

[5] Adafruit: *CircuitPython driver for SGP30 VoC sensor*. – URL https://github.com/adafruit/Adafruit_CircuitPython_SGP30. – Zugriffsdatum: 2024-03-12

[6] Adafruit: *CircuitPython library for the Adafruit SGP40 Air Quality Sensor*. – URL https://github.com/adafruit/Adafruit_CircuitPython_SGP40. – Zugriffsdatum: 2024-03-12

[7] Arai, T. ; Aiyama, Y. ; Maeda, Y. ; Sugi, M. ; Ota, J.: Agile Assembly System by "Plug and Produce". In: *CIRP Annals* 49 (2000), Nr. 1, S. 1–4. – URL https://www.sciencedirect.com/science/article/pii/S0007850607628832. – ISSN 0007-8506

[8] Avia Semiconductor: *HX711 Analog-Digital-Konverter für Wägezellen Datenblatt*. 2016. – URL https://cdn.sparkfun.com/datasheets/Sensors/ForceFlex/hx711_english.pdf. – Zugriffsdatum: 2024-01-27

[9] AZ-Delivery: *DS18B20 Sensor Datenblatt*. 2017. – URL https://cdn.shopify.com/s/files/1/1509/1638/files/DS18B20_3mCable_datasheet.pdf?v=1644320674. – Zugriffsdatum: 2024-01-27

[10] Bennulf, Mattias ; Danielsson, Fredrik ; Svensson, Bo: Identification of resources and parts in a Plug and Produce system using OPC UA. In: *Procedia Manufacturing* 38 (2019), S. 858–865. – URL https://www.sciencedirect.com/science/article/pii/S2351978920301682. – 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM 2019), June 24-28, 2019, Limerick, Ireland, Beyond Industry 4.0: Industrial Advances, Engineering Education and Intelligent Manufacturing. – ISSN 2351-9789

[11] Broadcom: *Distributed RabbitMQ*. 2023. – URL https://www.rabbitmq.com/distributed.html. – Zugriffsdatum: 2024-02-18

[12] Broadcom Inc.: *RabbitMQ: One broker to queue them all*. – URL https://www.rabbitmq.com/. – Zugriffsdatum: 2024-04-08

[13] Calvaresi, Davide ; Dubovitskaya, Alevtina ; Calbimonte, Jean-Paul ; Taveter, Kuldar ; Schumacher, Michael: *Multi-Agent Systems and Blockchain: Results from a Systematic Literature Review*. S. 110–126, 06 2018. – ISBN 978-3-319-94579-8

[14] Campbell, Scott: *DS18B20 TEMPERATURE SENSOR TUTORIAL*. – URL https://www.circuitbasics.com/raspberry-pi-ds18b20-temperature-sensor-tutorial/. – Zugriffsdatum: 2024-03-12

[15] Connectivity Standard Alliance: *matter: The Foundation for Connected Things*. – URL https://csa-iot.org/all-solutions/matter/

[16] Dugan, Jon ; Elliott, Seth ; Mah, Bruce A. ; Poskanzer, Jeff ; Prabhu, Kaustubh ; Ashley, Mark ; Brown, Aaron ; Jaissle, Aeneas ; Sahani, Susant ; Simpson, Bruce ; Tierney, Brian: *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. – URL https://iperf.fr/. – Zugriffsdatum: 2024-03-12

[17] Dürkop, Lars ; Jasperneite, Jürgen: *„Plug & Produce“ als Anwendungsfall von Industrie 4.0*. S. 59–71. In: Vogel-Heuser, Birgit (Hrsg.) ; Bauernhansl, Thomas (Hrsg.) ; Hompel, Michael ten (Hrsg.): *Handbuch Industrie 4.0 Bd.2: Automatisierung*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2017. – URL https://doi.org/10.1007/978-3-662-53248-5_50. – ISBN 978-3-662-53248-5

[18] Estrada, Nicolas ; Astudillo, Hernán: Comparing scalability of message queue system: ZeroMQ vs RabbitMQ. In: *2015 Latin American Computing Conference (CLEI)*, 2015, S. 1–6

[19] Ferber, Jacques ; Gutknecht, Olivier ; Michel, Fabien: From Agents to Organizations: An Organizational View of Multi-agent Systems. In: Giorgini, Paolo (Hrsg.) ; Müller, Jörg P. (Hrsg.) ; Odell, James (Hrsg.): *Agent-Oriented Software Engineering IV*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, S. 214–230. – ISBN 978-3-540-24620-6

[20] FreeOpcUa: *OPC UA library for python >= 3.7*. – URL https://github.com/FreeOpcUa/opcua-asyncio. – Zugriffsdatum: 2024-04-08

[21] FreeOpcUa: *opcua-asyncio, minimal server implementation*. 2023. – URL https://github.com/FreeOpcUa/opcua-asyncio/blob/master/examples/server-minimal.py. – Zugriffsdatum: 2023-12-26

[22] Google Inc.: *Build a Thread network with nRF52840 boards and OpenThread*. – URL https://openthread.io/codelabs/openthread-hardware?hl=de#0. – Zugriffsdatum: 2024-04-08

[23] Google Inc.: *OpenThread released by Google is an open-source implementation of Thread®*. https://openthread.io

[24] Grinberg, Miguel: *Video Streaming with Flask*. – URL https://blog.miguelgrinberg.com/post/video-streaming-with-flask. – Zugriffsdatum: 2024-03-12

[25] Grunwald, Gerhard ; Plank, Georg ; Reintsema, Detlef ; Zimmermann, Uwe ; Bischoff, Rainer: COMMUNICATION, CONFIGURATION, APPLICATION: The three layer concept for Plug-and-Produce, 05 2008. – ISBN 978-989-8111-35-7

[26] Hintjens, Pieter: *ZeroMQ: messaging for many applications*. Ö'Reilly Media, Inc.", 2013

[27] Hohpe, Gregor ; Woolf, Bobby: Enterprise integration patterns. In: *9th conference on pattern language of programs* Citeseer (Veranst.), 2002, S. 1–9

[28] Häussge, Gina: *Octoprint REST API*. 2024. – URL https://docs.octoprint.org/en/master/api/index.html. – Zugriffsdatum: 2024-03-12

[29] Häussge, Gina: *OctoPrint The snappy web interface for your 3D printer.* 2024. – URL https://octoprint.org/. – Zugriffsdatum: 2024-03-12

[30] JetBrains s.r.o.: *PyCharm The Python IDE for data science and web development.* – URL https://www.jetbrains.com/pycharm/. – Zugriffsdatum: 2024-04-08

[31] Jong, Irmen de: *Pyro - Python Remote Objects.* – URL https://pyro4.readthedocs.io/en/stable/. – Zugriffsdatum: 2024-04-08

[32] Junginger, Markus O.: *A high-performance messaging system for peer-to-peer networks.* University of Missouri-Kansas City, 2003

[33] Karnouskos, Stamatis ; Leitao, Paulo ; Ribeiro, Luis ; Colombo, Armando W.: Industrial Agents as a Key Enabler for Realizing Industrial Cyber-Physical Systems: Multiagent Systems Entering Industry 4.0. In: *IEEE Industrial Electronics Magazine* 14 (2020), Nr. 3, S. 18–32

[34] Kravari, Kalliopi ; Bassiliades, Nick: A survey of agent platforms. In: *Journal of Artificial Societies and Social Simulation* 18 (2015), Nr. 1, S. 11

[35] Madiwalar, Basavaraj ; Schneider, Ben ; Profanter, Stefan: Plug and Produce for Industry 4.0 using Software-defined Networking and OPC UA. In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2019, S. 126–133

[36] Nordic Semiconductor: *Developing with nRF52 Series.* – URL https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/device_guides/working_with_nrf/nrf52/developing.html#ug-nrf52. – Zugriffsdatum: 2024-04-08

[37] Nordic Semiconductor: *nRF52840.* – URL https://www.nordicsemi.com/Products/nRF52840. – Zugriffsdatum: 2024-04-08

[38] Nova Fitness Co., Ltd.: *SDS011 Sensor Datenblatt.* 2015. – URL https://cdn-reichelt.de/documents/datenblatt/X200/SDS011-DATASHEET.pdf. – Zugriffsdatum: 2024-01-27

[39] OPC Foundation: *Discovery and Global Services.* – URL https://reference.opcfoundation.org/GDS/v105/docs/. – Zugriffsdatum: 2024-04-13

[40] OPC Foundation: *Information Modelling in OPC UA*. – URL https://reference.opcfoundation.org/DI/v102/docs/4.2.1. – Zugriffsdatum: 2024-03-12

[41] OPC Foundation: *OPC 11030: UA Modelling Best Practices*. – URL https://opcfoundation.org/resources/whitepapers/

[42] OPC Foundation.: *OPC Foundation Homepage*. – URL https://opcfoundation.org/. – Zugriffsdatum: 2024-04-08

[43] OPC Foundation: *OPC Classic Specification*. 2012. – URL https://opcfoundation.org/developer-tools/specifications-classic. – Zugriffsdatum: 2024-02-18

[44] OPC Foundation: *OPC 40100-1: Machine Vision - Control, Configuration management, recipe management, result management*. 2019. – URL https://reference.opcfoundation.org/MachineVision/v100/docs/. – Zugriffsdatum: 2024-03-12

[45] OPC Foundation: *OPC UA Specification*. 2023. – URL https://reference.opcfoundation.org/. – Zugriffsdatum: 2024-02-18

[46] OPC Foundation: *OPC UA FileDirectoryType*. 2024. – URL https://reference.opcfoundation.org/Core/Part5/v104/docs/C.3.1. – Zugriffsdatum: 2024-03-12

[47] OPC Foundation: *OPC UA FileType*. 2024. – URL https://reference.opcfoundation.org/Core/Part5/v104/docs/C.2. – Zugriffsdatum: 2024-03-12

[48] OPC Foundation: *OPC UA Secruity*. 2024. – URL https://reference.opcfoundation.org/Core/Part2/v105/docs/. – Zugriffsdatum: 2024-03-12

[49] OpenCV: *Automated CI toolchain to produce precompiled opencv-python, opencv-python-headless, opencv-contrib-python and opencv-contrib-python-headless packages.*. – URL https://github.com/opencv/opencv-python. – Zugriffsdatum: 2024-03-12

[50] OpenSistemas: *a general-purpose multi-agent system module written in Python*. https://osbrain.readthedocs.io/en/stable/#

[51] Pagot, Michele: *Nova SDS011 dust sensor python package and command line.* – URL https://github.com/michelepagot/pysds011. – Zugriffsdatum: 2024-03-12

[52] Pallets: *Flask: The Python micro framework for building web applications.*. – URL https://github.com/pallets/flask/. – Zugriffsdatum: 2024-03-12

[53] Parasumanna Gokulan, Balaji ; Srinivasan, D.: *An Introduction to Multi-Agent Systems*. Bd. 310. S. 1–27, 07 2010. – ISBN 978-3-642-14434-9

[54] Patro, Suman ; Potey, Manish ; Golhani, Amit: Comparative study of middleware solutions for control and monitoring systems. In: *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, 2017, S. 1–10

[55] Profanter, Stefan ; Perzylo, Alexander ; Rickert, Markus ; Knoll, Alois: A Generic Plug and Produce System Composed of Semantic OPC UA Skills. In: *IEEE Open Journal of the Industrial Electronics Society* 2 (2021), S. 128–141. – ISSN 2644-1284

[56] Python Software Foundation: *Python is a programming language that lets you work quickly and integrate systems more effectively.*. – URL https://www.python.org/. – Zugriffsdatum: 2024-04-08

[57] Python Software Foundation: *Python interface to Tcl/Tk*. 2024. – URL https://docs.python.org/3.11/library/tkinter.html. – Zugriffsdatum: 2024-03-12

[58] Rascom Computerdistribution Ges.m.b.H.: *Noctua NF-A12x25 5V PWM Datenblatt*. 2018. – URL https://noctua.at/pub/media/blfa_files/infosheet/noctua_nf_a12x25_5v_pwm_datasheet_de.pdf. – Zugriffsdatum: 2024-01-27

[59] Reagan, Rob ; Reagan, Rob: Message Queues. In: *Web Applications on Azure: Developing for Global Scale* (2018), S. 343–380

[60] Ren, Wei ; Cao, Yongcan: *Distributed coordination of multi-agent networks: emergent problems, models, and issues*. Springer Science & Business Media, 2010

[61] Rocha, Andre ; Di Orio, Giovanni ; Barata, José ; Antzoulatos, Nikolas ; Castro, Elkin ; Scrimieri, Daniele ; Ratchev, Svetan ; Ribeiro, Luis: An agent based framework to support plug and produce. In: *2014 12th IEEE International*

Conference on Industrial Informatics (INDIN), July 2014, S. 504–510. – ISSN 2378-363X

[62] Roose, Marco: *This library allows you to communicate with the HX711 load cell amplifier with a Raspberry Pi.* – URL https://github.com/mpibpc-mroose/hx711/. – Zugriffsdatum: 2024-03-12

[63] Sean Luke, George Mason U.: *Multiagent Simulation And the MASON Library.* 2022. – URL https://cs.gmu.edu/~eclab/projects/mason/manual.21.pdf. – Zugriffsdatum: 2023-12-26

[64] Sensirion: *SGP30 Sensor Datenblatt.* 2020. – URL https://sensirion.com/media/documents/984E0DD5/61644B8B/Sensirion_Gas_Sensors_Datasheet_SGP30.pdf. – Zugriffsdatum: 2024-01-27

[65] Sensirion: *SGP40 Sensor Datenblatt.* 2022. – URL https://sensirion.com/media/documents/296373BB/6203C5DF/Sensirion_Gas_Sensors_Datasheet_SGP40.pdf. – Zugriffsdatum: 2024-01-27

[66] Sensirion: *What is Sensirion's VOC Index?* 2022. – URL https://sensirion.com/media/documents/02232963/6294E043/Info_Note_VOC_Index.pdf. – Zugriffsdatum: 2024-01-27

[67] Sodian, Loigen ; Wen, Jaden P. ; Davidson, Leonard ; Loskot, Pavel: Concurrency and Parallelism in Speeding Up I/O and CPU-Bound Tasks in Python 3.10. In: *2022 2nd International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI)*, Sep. 2022, S. 560–564

[68] Sudeikat, Jan ; Köhler-Bussmeier, Michael: Towards Integrating Multi-Agent Organizations in OPC UA for Developing Adaptive Cyber-Physical Systems. In: Demmler, Daniel (Hrsg.) ; Krupka, Daniel (Hrsg.) ; Federrath, Hannes (Hrsg.): *INFORMATIK 2022*, Gesellschaft für Informatik, Bonn, 2022, S. 1565–1570

[69] Sustrik, Martin: *Broker vs. Brokerless.* – URL http://wiki.zeromq.org/whitepapers:brokerless. – Zugriffsdatum: 2024-03-12

[70] Tarkoma, Sasu: *Publish/subscribe systems: design and principles.* John Wiley & Sons, 2012

[71] The pip developers: *The Python package installer.* – URL https://pip.pypa.io/en/stable/. – Zugriffsdatum: 2024-04-13

[72] The VOLTTRON Community: *VOLTTRON™ an open source, scalable, and distributed platform.* 2022. – URL https://volttron.readthedocs.io/en/main/. – Zugriffsdatum: 2023-12-26

[73] The ZeroMQ project: *ZeroMQ: An open-source universal messaging library.* – URL https://zeromq.org/. – Zugriffsdatum: 2024-04-08

[74] Thread Group: *Thread Network Fundamentals.* https://www.threadgroup.org/Portals/0/documents/support/Thread%20Network%20Fundamentals_v3.pdf

[75] Thread Group: *What is Thread.* https://www.threadgroup.org

[76] Thread Group: *Thread 1.3.0 Features White Paper.* July 2022. – URL https://www.threadgroup.org/Portals/0/documents/support/Thread1.3.0WhitePaper_07192022_3990_1.pdf

[77] Unified Automation GmbH: *UaExpert—A Full-Featured OPC UA Client.* 2024. – URL https://www.unified-automation.com/products/development-tools/uaexpert.html. – Zugriffsdatum: 2024-03-12

[78] Vlassis, Nikos: *A Concise introduction to multiagent systems and distributed artifical intelligence.* In: *Synthesis Lectures On Artificial Intelligence And Machine Learning*

[79] Xie, Jing ; Liu, Chen-Ching: Multi-agent systems and their applications. In: *Journal of International Council on Electrical Engineering* 7 (2017), 01, S. 188–197

[80] Zhang, Qian ; Pardo, Michal ; Rudich, Yinon ; Kaplan-Ashiri, Ifat ; Wong, Jenny P. S. ; Davis, Aika Y. ; Black, Marilyn S. ; Weber, Rodney J.: Chemical Composition and Toxicity of Particles Emitted from a Consumer-Level 3D Printer Using Various Materials. In: *Environmental Science & Technology* 53 (2019), Nr. 20, S. 12054–12061. – URL https://doi.org/10.1021/acs.est.9b04168. – PMID: 31513393

A Anhang

A.1 Agentenkonfigurationen

In diesem Abschnitt wird auf die umgesetzten Agentenkonfigurationen eingegangen. Es werden die genutzten Python-Packages benannt und auf Besonderheiten in den Konfigurationen hingewiesen. Nach der Erklärung der Konfiguration ist diese im YAML-Format aufgeführt.

A.1.1 SGP30 Gassensor

Für die Kommunikation mit dem SGP30-Sensors [64] werden die Python-Packages adafruit-circuitpython-sgp30 [5] und adafruit-blinka [5] von Adafruit eingesetzt. Mithilfe dieser werden die eCO2 und TVOC Werte über I^2C ausgelesen. Da die Werte des SGP30 nach der Inbetriebnahme ungenau sind, wurde im Vorfeld ein Baseline-Wert ermittelt und in der Konfiguration als statische Baseline verwendet. Für die Umgebungstemperatur und Luftfeuchtigkeit sind Durchschnittswerte der Testumgebung verwendet worden. Der eC02-Wert wird zwar ermittelt und im OPC UA Server bereitgestellt, aber nicht weiter für die Bestimmung der Luftqualität im System verwendet, da dieser eine auf dem TVOC basierende Schätzung ist und somit eine zu hohe Ungenauigkeit aufweist.

```
config_name: gas_sensor_sgp30
agentType: Sensor
port: sda/scl
hw_address_identifier: '0x58'
bus_type: i2c
install_required_packages: |
  def install_package(self):
    import subprocess
    subprocess.check_call(['pip','install', 'adafruit-circuitpython-sgp30'],
    stdout=subprocess.DEVNULL)
```

```
10      subprocess.check_call(['pip','install', 'board'], stdout=subprocess.
        DEVNULL)
11      subprocess.check_call(['pip','install', 'adafruit-blinka'], stdout=
        subprocess.DEVNULL)
12      print("["+self.id+"]: Installed packages")
13  agent_method_code: |
14    def agent_method(self, hw_address='', opc_var_map=None):
15      import time
16      import board
17      import busio
18      import adafruit_sgp30
19      from adafruit_blinka.microcontroller.bcm283x import pin
20
21      try:
22        i2c = busio.I2C(pin.SCL, pin.SDA, frequency=100000)
23        sgp30 = adafruit_sgp30.Adafruit_SGP30(i2c)
24        sgp30.set_iaq_baseline(0x8973, 0x8AAE)
25        sgp30.set_iaq_relative_humidity(celsius=22.1, relative_humidity=44)
26        sensor_values = {
27          "eCO2": sgp30.eCO2,
28          "TVOC": sgp30.TVOC
29        }
30        opc_var_map['sensor_value']["sensor_value eCO2"].set_value(
        sensor_values["eCO2"])
31        opc_var_map['sensor_value']["sensor_value TVOC"].set_value(
        sensor_values["TVOC"])
32        pub_data_dict = {"agent_id": self.id, "timestamp": datetime.utcnow().
        isoformat(), "unit": "ppm/ppb", "precision": 15, "value_dict": True, "
        value": sensor_values}
33        self.publish_message(str(self.group_id)+":"+"airquality_data",
        pub_data_dict)
34      finally:
35        i2c.deinit()
36
37  properties:
38    name: eCO2/TVOC Sensor
39    manufacturer: Adafruit
40    unit: ppm/ppb
41    model_name: SGP30
42    precision: 15
43    sensor_value_type: dictionary
44    sensor_value: {eCO2 : 0, TVOC : 0}
45    hardware_group: airquality_sensor
```

```
46   subscriptions: ['gas_sensors', 'airquality_sensors', 'airquality_data']
```

Listing A.1: SGP30-Agentenkonfiguration

A.1.2 SGP40 Gassensor

In der Konfiguration des SGP40 [65] werden die Python-Packages von Adafruit (Circuitpython-SGP40 [6], Blinka [4]) zur Kommunikation mit dem Sensor eingesetzt. Des Weiteren wird das Python-Package sensirion-gas-index-algorithm von Sensirion [66] eingesetzt um aus den Rohwerten des Sensors den VOC-Index im Bereich von 1-500 zu ermitteln.

```
1  config_name: gas_sensor_sgp40
2  agentType: Sensor
3  port: sda/scl
4  hw_address_identifier: '0x59'
5  bus_type: i2c
6  install_required_packages: |
7    def install_package(self):
8      import subprocess
9      subprocess.check_call(['pip','install', 'adafruit-circuitpython-sgp40'],
       stdout=subprocess.DEVNULL)
10     subprocess.check_call(['pip','install', 'board'], stdout=subprocess.
       DEVNULL)
11     subprocess.check_call(['pip','install', 'adafruit-blinka'], stdout=
       subprocess.DEVNULL)
12     subprocess.check_call(['pip','install', 'sensirion-gas-index-algorithm'],
        stdout=subprocess.DEVNULL)
13     print("["+self.id+"]: Installed packages")
14 agent_method_code: |
15   def agent_method(self, hw_address='', opc_var_map=None):
16     import time
17     import board
18     import busio
19     import adafruit_sgp40
20     from adafruit_blinka.microcontroller.bcm283x import pin
21     from sensirion_gas_index_algorithm.voc_algorithm import VocAlgorithm
22
23     try:
24       voc_algorithm = VocAlgorithm()
25       i2c = busio.I2C(pin.SCL, pin.SDA)t
26       sgp40 = adafruit_sgp40.SGP40(i2c)
27       for _ in range(100):
```

```
28          voc_index = voc_algorithm.process(sgp40.raw)
29        opc_var_map['sensor_value']["sensor_value VOC-Index"].set_value(
    voc_index)
30        opc_var_map['sensor_value']["sensor_value Raw-Value"].set_value(sgp40.
    raw)
31        pub_data_dict = {"agent_id": self.id, "timestamp": datetime.utcnow().
    isoformat(), "unit": "voc_index", "precision": 0, "value_dict": False, "
    value": voc_index}
32        self.publish_message(str(self.group_id)+":"+"voc_index", pub_data_dict)
33      finally:
34        i2c.deinit()
35
36  properties:
37    name: VOC-Index Sensor
38    manufacturer: Adafruit
39    unit: VOC Index
40    model_name: SGP40
41    precision: 0
42    sensor_value_type: dictionary
43    sensor_value: {Raw-Value: 0, VOC-Index: 0}
44    hardware_group: airquality_sensor
45    subscriptions: ['gas_sensors', 'airquality_sensors', 'voc_index']
```

Listing A.2: SGP40-Agentenkonfiguration

A.1.3 DS18B20 OneWire Temperatursensor

Zur Abfrage der Temperaturwerte des DS18B20 [9] wird kein weiteres Python-Package genutzt. Der Wert wird direkt mithilfe der onewire-Adresse aus dem adressierten Sensor unter /sys/bus/w1/devices/[sensorAdresse]/w1_slave ausgelesen und verarbeitet. Danach wird dieser als float-Wert für die weitere Verwendung bereitgestellt. Die Konfiguration wurde basierend auf der Entwicklung von Scott Campbell [14] entwickelt.

```
1  config_name: one_wire_temp_sensor_ds18b20
2  agentType: Sensor
3  port: /sys/bus/w1/devices/
4  hw_address_identifier: '28-'
5  bus_type: onewire
6  install_required_packages: |
7    def install_package(self):
8      import subprocess
9      print("["+self.id+"]: Installed packages")
```

```
agent_method_code: |
  def agent_method(self, hw_address='', opc_var_map=None):
    import os
    import glob
    import time
    os.system('modprobe w1-gpio')
    os.system('modprobe w1-therm')

    try:
      base_dir = '/sys/bus/w1/devices/'
      device_folder = glob.glob(base_dir + self.hw_address)[0]
      device_file = device_folder + '/w1_slave'

      def read_temp_raw():
        with open(device_file, 'r') as f:
        lines = f.readlines()
        return lines

      lines = read_temp_raw()

      while lines[0].strip()[-3:] != 'YES':
        time.sleep(0.2)
        lines = read_temp_raw()

      equals_pos = lines[1].find('t=')
      if equals_pos != -1:
        temp_string = lines[1][equals_pos + 2:]
        temp_c = float(temp_string) / 1000.0
        self.opc_var_map['sensor_value'].set_value(temp_c)
        pub_data_dict = {"agent_id": self.id, "timestamp": datetime.utcnow().
    isoformat(), "unit": "celsius", "precision": 1, "value_dict": False, "
    value": temp_c}
        self.publish_message(str(self.group_id)+":"+"temperature_data",
    pub_data_dict)
    except IndexError as e:
      self.logger.error("[TemperatureSensor] An error occurred in: "+ str(
    self.hw_address)+" ", exc_info=1)
      return

properties:
  name: Onewire-Temperature-Sensor
  manufacturer: AZDelivery
  unit: Celsius
  model_name: DS18B20
```

```
  precision: 1
  sensor_value_type: float
  sensor_value: 0.0
  hardware_group: temperature_sensor
  subscriptions: ['temperature_sensors', 'temperature_data']
```

Listing A.3: DS18B20-Agentenkonfiguration

A.1.4 Wägezelle mit HX711 Amplifier

Für die Konfiguration des HX711 Amplifier [8] wird das Python-Package HX711 von Marco Roose [62] eingesetzt. In der Konfiguration wird die notwendige Tarierung der Wägezelle vorgenommen, um das Gewicht des eingesetzten Plexiglases und des Filamenthalters auf der Wägezelle zu berücksichtigen.

```
config_name: hx711_weight_measurement
agentType: Sensor
port: gpio
hw_address_identifier:
  - pin: 5
  state: LOW
  - pin: 6
  state: LOW
bus_type: gpio
install_required_packages: |
  def install_package(self):
    import subprocess
    subprocess.check_call(['pip', 'install', 'hx711'], stdout=subprocess.
    DEVNULL)
    print("["+self.id+"]: Installed packages")
agent_method_code: |
  from RPi import GPIO
  from hx711 import HX711
  is_setup_done = False

  def agent_method(self, hw_address='', opc_var_map=None):
    from hx711 import HX711
    from RPi import GPIO
    import statistics
    import time

    try:
```

```
27        hx711 = HX711(dout_pin=5, pd_sck_pin=6, channel='A', gain=64)
28        hx711.reset()
29        tare_value = 130835
30        raw_reading_1kg = 182000  # 182k is roughly for this sensor 1kg, with
      the plexiglas build on it
31        reference_weight_1kg = 1000
32        conversion_factor = reference_weight_1kg / (raw_reading_1kg -
      tare_value)
33        raw_values = hx711.get_raw_data()
34        weights_grams = [(raw_value - tare_value) * conversion_factor for
      raw_value in raw_values]
35        avg_weight = round(statistics.mean(weights_grams),2)
36        self.opc_var_map['sensor_value'].set_value(avg_weight)
37        pub_data_dict = {"agent_id": self.id, "timestamp": datetime.utcnow().
      isoformat(), "unit": "grams", "precision": 0, "value_dict": False, "value
      ": avg_weight}
38        self.publish_message(str(self.group_id)+":"+"weight_data",
      pub_data_dict)
39      except Exception as e:
40        self.logger.error("[hw711_weight_mesasurement]: Error in handling
      sensor: ", exc_info=1)
41
42  properties:
43    name: Weight Measurement Sensor
44    manufacturer: DIY Mall Inc.
45    unit: grams
46    model_name: HX711 Weight Sensor
47    precision: None
48    sensor_value_type: float
49    sensor_value: 0.00
50    hardware_group: weight_sensors
51    subscriptions: ['weight_data']
```

Listing A.4: HX711-Agentenkonfiguration

A.1.5 SDS011 Laser PM2.5/PM10 Sensor

Die Konfiguration des SDS011 [38] enthält nur eine Abfrage der seriellen Schnittstelle, welche über USB angeschlossen wird. Über diese werden alle 30 Sekunden die PM Werte ausgelesen. Die initiale Konfiguration des SDS011 wurde mit dem Python-Package pysds011 von Michele Pagot [51] durchgeführt, damit ein Duty-Cycle von 30 Sekunden eingehalten wird um den Partikelsensor zu schonen.

```
config_name: airquality_sensor_SDS011
agentType: Sensor
port: /dev/ttyUSB0
hw_address_identifier: USB_Serial
bus_type: USB
install_required_packages: |
  def install_package(self):
    import subprocess
    subprocess.check_call(['pip','install', 'pysds011'], stdout=subprocess.DEVNULL)
    print("["+self.id+"]: Installed packages")

agent_method_code: |
  def agent_method(self, hw_address='', opc_var_map=None):
    import serial
    serial_connection = serial.Serial("/dev/ttyUSB0")
    if not serial_connection.isOpen():
      print("Serial port not open so open it...")
      serial_connection.open()
    sensor_data = []
    for index in range(0,10):
      sensor_datum = serial_connection.read()
      sensor_data.append(sensor_datum)
    pmtwofive = int.from_bytes(b''.join(sensor_data[2:4]), byteorder='little') / 10
    pmten = int.from_bytes(b''.join(sensor_data[4:6]), byteorder='little') / 10
    opc_var_map["sensor_value"]["sensor_value pm25"].set_value(pmtwofive)
    opc_var_map["sensor_value"]["sensor_value pm10"].set_value(pmten)
    pub_data_dict = {"agent_id": self.id, "timestamp": datetime.utcnow().isoformat(), "unit": "micrograms/m3", "precision": 1, "value_dict": True, "value": {"pm2.5": pmtwofive, "pm10": pmten}}
    self.publish_message(str(self.group_id)+":"+"pm_data", pub_data_dict)

properties:
  name: Laser PM2.5 / PM10 Sensor
  manufacturer: Nova Fitness Co., Ltd.
  unit: micrograms/m3
  model_name: SDS011
  precision: 1
  sensor_value_type: dictionary
  sensor_value: { pm25: 0.00, pm10: 0.00 }
  hardware_group: airquality_sensor
```

```
39   subscriptions: ['gas_sensors', 'airquality_sensors', 'pm_data']
```

Listing A.5: SDS011-Agentenkonfiguration

A.1.6 Noctua NF-A12x25 5V PWM Lüfter

Zum Betrieb des 5V PWM-Lüfters [58] wurde eine Konfiguration entwickelt, um die PWM-DutyCycles anzupassen und die RPM des Lüfters zu errechnen. Es wird das Python-Package RPi.GPIO genutzt um direkt auf dem gewählten Pin die Konfiguration für PWM ausführen zu können.

```
1  config_name: fan_control_gpio
2  agentType: Actor
3  port: gpio
4  hw_address_identifier:
5    - pin: 13
6    state: HIGH
7    - pin: 19
8    state: HIGH
9  bus_type: gpio
10 install_required_packages: |
11   def install_package(self):
12     import subprocess
13     subprocess.check_call(['pip','install', 'RPi.GPIO'], stdout=subprocess.
       DEVNULL)
14     print("["+self.id+"]: Installed packages")
15
16 agent_method_code: |
17   import time
18   from RPi import GPIO
19
20   is_setup_done = False
21   pwm_control = None
22
23   def change_duty(pwm_duty):
24     from RPi import GPIO
25     if pwm_duty > 100:
26       pwm_duty = 100
27     if 100 > pwm_duty > 50:
28       pwm_duty = 50
29     if 50 > pwm_duty > 8:
30       pwm_duty = 15
```

```
31      if pwm_duty <= 8:
32        pwm_duty = 0
33      pwm_control.ChangeDutyCycle(pwm_duty)
34
35    def get_rpm(tacho_pin):
36      from RPi import GPIO
37      start_time = time.time()
38      pinread = None
39      for _ in range(100):
40        pinread = GPIO.wait_for_edge(tacho_pin, GPIO.FALLING, timeout=1000)
41        if pinread is None:
42          break
43        if pinread is None:
44          return 0
45      due_time = time.time() - start_time
46      frequency = 100 / due_time
47      rpm = int(frequency / 2 * 60)
48      return rpm
49
50    def agent_method(self, hw_address='', opc_var_map=None):
51      from RPi import GPIO
52      tacho_pin = 13
53      pwm_pin = 19
54      global is_setup_done
55      global pwm_control
56      global change_duty
57      global get_rpm
58      if not is_setup_done:
59        GPIO.setwarnings(False)
60        GPIO.setmode(GPIO.BCM)
61        GPIO.setup(pwm_pin, GPIO.OUT)
62        pwm_control = GPIO.PWM(pwm_pin, 100)
63        pwm_control.start(100)
64        GPIO.setup(tacho_pin, GPIO.IN)
65        self.subscribe(str(self.group_id)+":"+"fan_control")
66        pub_time = time.time()
67        is_setup_done = True
68
69      try:
70        self.opc_var_map["sensor_value"].set_value(get_rpm(tacho_pin))
71        pub_data_dict = {"agent_id": self.id, "timestamp": datetime.utcnow().
      isoformat(), "unit": "rpm", "precision": 0, "value_dict": False, "value":
       get_rpm(tacho_pin)}
72        self.publish_message(str(self.group_id)+":"+"fan_speed", pub_data_dict)
```

```
      topic, message = self.receive_message()
      if topic == str(self.group_id)+":"+"fan_control":
        change_duty(int(message))
        new_rpm = get_rpm(tacho_pin)
        self.opc_var_map["sensor_value"].set_value(new_rpm)
        self.publish_message(str(self.group_id)+":"+"fan_change",str(self.id)
    + ": changed fanspeed to " + str(new_rpm))
    except zmq.Again:
      pass

properties:
  name: Analog fan control
  manufacturer: Noctua
  unit: rpm
  model_name: NF-A12x25 5V
  precision: None
  sensor_value_type: int
  sensor_value: 0
  hardware_group: air_control_actors
  subscriptions: ['gas_sensors', 'airquality_sensors', 'fan_control']
```

Listing A.6: PWM-Lüfter-Agentenkonfiguration

A.1.7 Webcam

Für den Betrieb der Webcam wird in der Agentenkonfiguration auf das erste verfügbare Videogerät zugegriffen (video0) und ein Videostream mittels des Python-Packages OpenCV [49] erzeugt. Dieser Videostream wird dann in einer Flask-Anwendung [52] genutzt, welche einen Debug-Webserver bereitstellt und dort den Videostream anbietet. Des Weiteren wird unter /snapshot vom Webserver ein Snapshot des Videostreams angeboten, der von OpenCV bei Aufruf erzeugt wird. Die Entwicklung der Konfiguration basiert auf einem Artikel von Miguel Grinberg [24].

```
config_name: webcam_stream_agent
agentType: Camera
port: /dev/video0
hw_address_identifier: Webcam_C170
bus_type: USB
install_required_packages: |
  def install_package(self):
    import subprocess
```

```
9     subprocess.check_call(['pip', 'install', 'opencv-python', 'flask'],
      stdout=subprocess.DEVNULL)
10    print("["+self.id+"]: Installed packages")
11
12 agent_method_code: |
13   import cv2
14   from flask import Flask, render_template, Response, send_file,
       make_response
15   import io
16
17   app = Flask(__name__)
18   cap = cv2.VideoCapture(0)  # Use the webcam at index 0
19   snapshot_frame = None
20
21   def generate_frames():
22     from flask import Flask, render_template, Response, send_file,
       make_response
23     global snapshot_frame, cap, cv2, io
24     while True:
25       if snapshot_frame is not None:
26         ret, jpeg = cv2.imencode('.jpg', snapshot_frame)
27         if ret:
28           yield (b'--frame\r\n'
29                  b'Content-Type: image/jpeg\r\n\r\n' + jpeg.tobytes() + b'\r\n
      ')
30           snapshot_frame = None
31       else:
32         success, frame = cap.read()
33         if not success:
34           break
35         else:
36           ret, jpeg = cv2.imencode('.jpg', frame)
37           if not ret:
38             continue
39             yield (b'--frame\r\n'
40                    b'Content-Type: image/jpeg\r\n\r\n' + jpeg.tobytes() + b'\
      r\n')
41
42   @app.route('/')
43   def video_feed():
44     from flask import Flask, render_template, Response, send_file,
       make_response
45     global generate_frames
```

```
    return Response(generate_frames(), mimetype='multipart/x-mixed-replace; boundary=frame')

@app.route('/snapshot')
def snapshot():
    from flask import Flask, render_template, Response, send_file, make_response
    global snapshot_frame, cap, cv2, io
    success, frame = cap.read()
    if success:
        snapshot_frame = frame
        ret, jpeg = cv2.imencode('.jpg', frame)
        if ret:
            snapshot_frame = None  # Clear the snapshot frame after using
            response = make_response(send_file(io.BytesIO(jpeg.tobytes()), mimetype='image/jpeg'))
            response.headers['Content-Disposition'] = 'attachment; filename=snapshot.jpg'
            return response
        return "Failed to encode snapshot.", 500
    return "Failed to capture snapshot.", 500

    app.run(host='0.0.0.0', port=7777)  # Start the Flask server

properties:
  name: Webcam Stream Agent
  manufacturer: Logitech
  unit: None
  model_name: Generic Webcam
  precision: None
  sensor_value_type: None
  sensor_value: None
  url: "http:localhost:7777/"
  hardware_group: webcam
  snapshot_url: "http:localhost:7777/snapshot"
```

Listing A.7: Webcam-Agentenkonfiguration

A.2 Grafiken

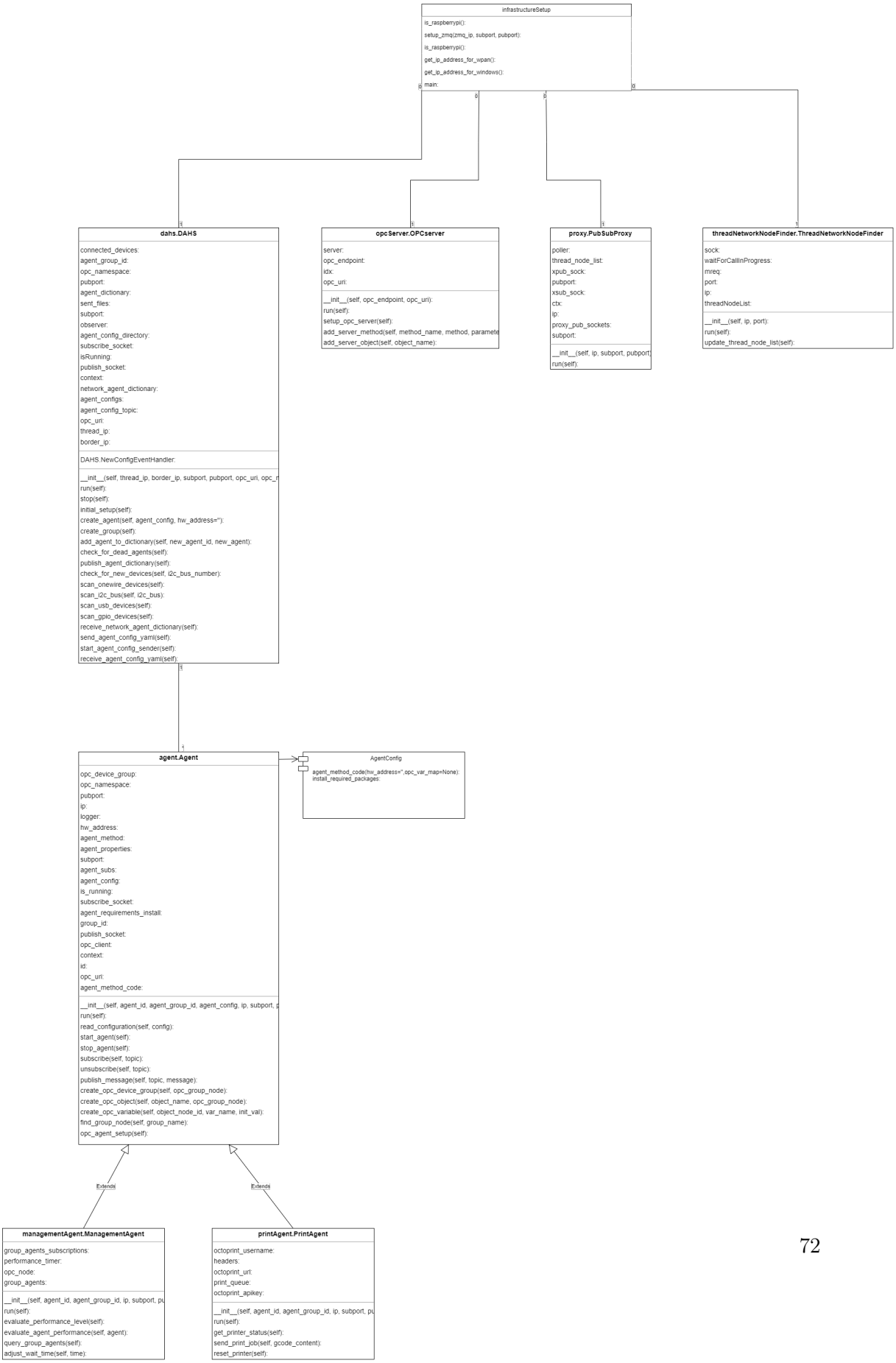

Abbildung A.1: UML-Diagramm mit allen Komponenten

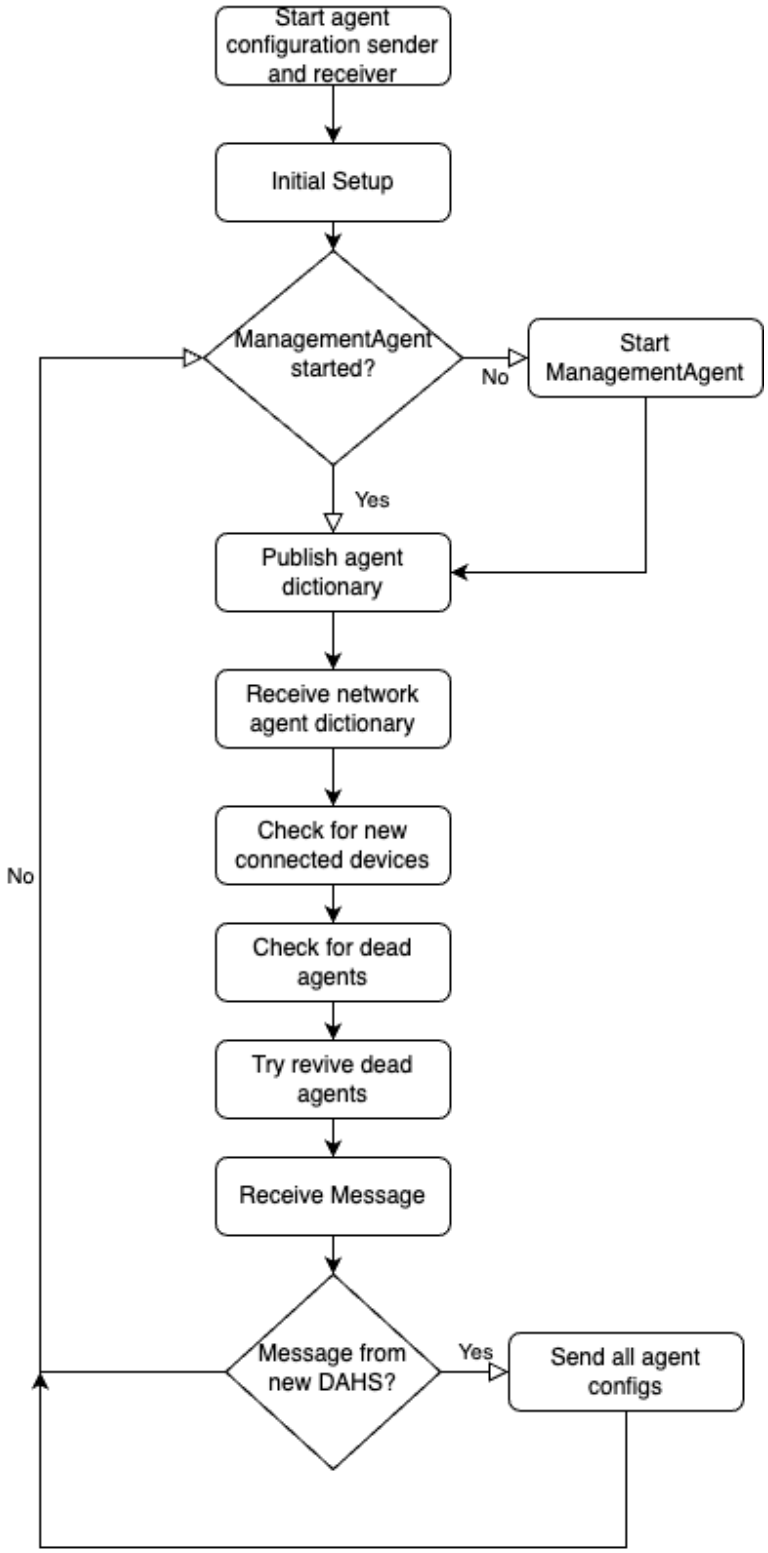

Abbildung A.2: Ablauf des DAHS, Processingloop des DAHS beginnend nach dem initialen Setup

Glossar

3D-Druck Ein Fertigungsprozess, bei dem Objekte durch schichtweises Hinzufügen von Material aufgebaut werden. Dies erfolgt auf Grundlage digitaler 3D-Modelle und findet Anwendung in verschiedenen Bereichen wie Prototypenbau, Produktentwicklung und Fertigung..

I²C Das I²C-Protokoll, auch als Inter-Integrated Circuit bekannt, ist ein synchrones seriell betriebenes Protokoll, das die Kommunikation zwischen Mikrocontrollern und anderen integrierten Schaltungen erleichtert. Es ermöglicht den Anschluss mehrerer Geräte an denselben Bus und unterstützt die bidirektionale Datenübertragung. I²C wird häufig in der Elektronik für die Kommunikation zwischen verschiedenen Komponenten verwendet..

MQTT MQTT (Message Queuing Telemetry Transport) ist ein leichtgewichtiges, offenes Netzwerkprotokoll für die Übertragung von Nachrichten zwischen Geräten. Es wurde für den Einsatz in verteilten Umgebungen mit begrenzten Ressourcen entwickelt, insbesondere im Internet of Things (IoT). MQTT unterstützt das Publish/Subscribe-Muster und ermöglicht die effiziente Übertragung von Nachrichten zwischen verschiedenen Komponenten..

OneWire-Protokoll Das OneWire-Protokoll ist ein seriell betriebenes Protokoll, das die Kommunikation mit Geräten über einen einzigen Datenleitung ermöglicht. Es wird häufig in Anwendungen zur Identifikation von digitalen Bauteilen verwendet, da diese eine eindeutige OneWire-Adresse beim Anschluss erhalten. Das Protokoll erlaubt die Kommunikation mit mehreren Geräten, die an einer einzigen Datenleitung angeschlossen sind..

PM10 Feinstaubpartikel mit einem Durchmesser von 10 Mikrometern oder kleiner. PM10-Partikel umfassen sowohl PM2.5 als auch größere Partikel. Sie können Atemwegsprobleme verursachen und haben Auswirkungen auf die Luftqualität..

PM2.5 Feinstaubpartikel mit einem Durchmesser von 2,5 Mikrometern oder kleiner. PM2.5-Partikel können aufgrund ihrer geringen Größe tief in die Atemwege eindringen und stehen im Zusammenhang mit verschiedenen gesundheitlichen Auswirkungen, einschließlich Atemwegs- und Herz-Kreislauf-Erkrankungen..

Publish-subscribe pattern Ein Entwurfsmuster für asynchrone Kommunikation, bei dem Komponenten Ereignisse (Events) oder Themen (Topics) abonnieren (subscribe) und Komponenten Nachrichten über Events oder an diese Topics veröffentlichen (publish). Eine Vermittlungskomponente (Broker) leitet die Nachrichten der Publisher an alle Subscriber weiter..

Python Eine weit verbreitete, interpretierte Programmiersprache, bekannt für klare Syntax und Vielseitigkeit. Entwickelt von Guido van Rossum, ist Python in Bereichen wie Webentwicklung, Datenanalyse und künstliche Intelligenz beliebt..

RabbitMQ RabbitMQ ist eine Open-Source-Message-Broker-Software, die das AMQP-Protokoll (Advanced Message Queuing Protocol) implementiert. Als Message Broker fungiert RabbitMQ als Vermittler zwischen verschiedenen Anwendungen, die Nachrichten austauschen. Es ermöglicht eine zuverlässige, skalierbare und flexible Nachrichtenübertragung zwischen verschiedenen Systemkomponenten..

YAML Eine benutzerfreundliche Daten-Serialisierungssprache für alle Programmiersprachen. YAML steht für "YAML Ain't Markup Language". Sie verwendet Einrückungen und Klartext, um Daten strukturiert darzustellen..

ZMQ ZeroMQ (ZMQ, ØMQ, 0MQ) ist eine leichte, flexible und skalierbare Messaging-Bibliothek, die verschiedene Kommunikationsmuster unterstützt. Es bietet eine einfache API für den Austausch von Nachrichten zwischen Anwendungen über verschiedene Transportprotokolle. ZeroMQ eignet sich für verteilte Systeme und ermöglicht eine effiziente, lose gekoppelte Kommunikation zwischen Komponenten..

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original