Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Nico André Töpfer

Entwicklung eines aggregierenden OPC UA Servers zur vertikalen Integration mehrerer Feldgeräte am Beispiel eines mobilen Bohrrobotersystems

Fakultät Technik und Informatik
Department Maschinenbau und Produktion

Faculty of Engineering and Computer Science
Department of Mechanical Engineering and Production Management

Nico André Töpfer

Entwicklung eines aggregierenden OPC UA Servers zur vertikalen Integration mehrerer Feldgeräte am Beispiel eines mobilen Bohrrobotersystems

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Maschinenbau - Entwicklung und Konstruktion (B.Sc.)*
am Department Maschinenbau und Produktion
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

in Zusammenarbeit mit:
Fraunhofer IFAM
Ottenbecker Damm 12
21684 Stade

Betreuender Professor: Prof. Dr. Dietmar Pähler
Zweitprüfer: Philip Koch, M.Sc.

Eingereicht am: 18.03.2022

HAW HAMBURG

Department Maschinenbau und Produktion

Aufgabenstellung

für die Bachelorthesis

von Herrn **Nico Andre Töpfer**

Matrikel-Nummer: [redacted]

Thema: **Entwicklung eines aggregierenden OPC UA Servers zur vertikalen Integration mehrerer Feldgeräte am Beispiel eines mobilen Bohrrobotersystems**

Stichworte:
- **I4.0**
- **Vertikale Integration**
- **OPC UA**
- **Aggregationsserver**

Hintergrund der Bachelorarbeit

Die Bachelorarbeit wird durchgeführt im Rahmen des Projektes „MFLEX2025" des Fraunhofer Institut für Fertigungstechnik und Angewandte Materialforschung, am Standort Stade.

Ziel- und Aufgabenstellung

Ziel der Forschungsarbeit ist eine Untersuchung inwiefern die automatische Aggregation mehrerer Feldgeräte, mit generischen OPC UA Servern, zu einem prozessspezifischen Bohrroboter zu realisieren ist. Hierbei soll eine effiziente, vom Anwendungsszenario unabhängige, Aggregation konzipiert und umgesetzt werden, welche die aktuellen Spezifikationen der OPC Foundation berücksichtigt.

Die Bachelorarbeit umfasst dabei folgende Aufgabenpakete:

- Projektplanung (Zeitplanung, Ablauf, Umfang)
- Konzeption (Recherche, Funktionen, Aufbau)
- Implementierung (Programmierung, zyklische Tests)
- Evaluation (Funktionstest, Auswertung)
- Dokumentation (Schriftliche Ausarbeitung)

Die im Rahmen der Arbeit generierten Ergebnisse sind umfassend und stringent zu dokumentieren. Die Ausführungen schließen mit einer ausführlichen Diskussion samt Ausblick auf mögliche weitere Umsetzungsmaßnahmen im Unternehmen.

20.12.2021
Datum

[redacted]

Nico Töpfer

Thema der Arbeit

Entwicklung eines aggregierenden OPC UA Servers zur vertikalen Integration mehrerer Feldgeräte am Beispiel eines Bohrrobotersystems

Stichworte

Industrie 4.0, Vertikale Integration, OPC UA, Aggregationsserver

Kurzzusammenfassung

Ziel dieser Ausarbeitung ist die Entwicklung eines Aggregierenden OPC UA Servers, welcher den Datenaustausch mit einer Vielzahl an Feldgeräten mit generischen OPC UA Servern ermöglicht. Hierzu wird zunächst das Kommunikationsprotokoll OPC UA erläutert, sowie die Rahmenbedingungen für eine Aggregation innerhalb des Projektes MFlex2025 definiert. Im Anschluss werden die genutzten Technologien zur Implementierung vorgestellt und ein Konzept zur systematischen Aggregation erarbeitet. Nachfolgend wird die Implementierung beschrieben, welche eine Erläuterung zur Klassensystematik und dem darauf aufbauenden Prozessablauf beinhaltet. Abschließend wird die Softwareimplementierung getestet und evaluiert.

Topic of Thesis

Development of an aggregating OPC UA Server for vertical integration of multiple field devices in case of a mobile drilling robot system

Keywords

Industry 4.0, Vertical Integration, OPC UA, Server Aggregation

Abstract

The goal of this thesis is the development of an aggregating OPC UA server, which allows data exchange with multiple field devices with generic OPC UA servers via one server interface. For this purpose, the communication protocol OPC UA is explained first, and the framework conditions for an aggregation within the MFlex2025 project are defined. The (software) technologies used for the implementation are then presented and a concept for systematic aggregation is developed. The implementation is described afterwards, which includes an explanation of the class system and the process flow based on it. Finally, the software implementation is tested and evaluated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abkürzung	Erläuterung
AGV	Automated Guided Vehicle
AMQP	Advanced Message Queuing Protocol
BMBF	Bundesministerium für Bildung und Forschung
BMWK	Bundesministerium für Wirtschaft und Klimaschutz
BSI	Bundesamt für Sicherheit in der Informationstechnik
BVE	Bohrvorschubeinheit
CPS	Cyber-Physical-Systems
CPU	Central processing unit
DLR	Deutsches Zentrum für Luft- und Raumfahrt
Fraunhofer IFAM	Fraunhofer-Institut für Fertigungstechnik und Angewandte Materialforschung
HTS	Hightech-Strategie des Bundesministerium
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
IoT	Internet der Dinge (engl. *Internet of Things*)
IP	Internet Protokoll
IR	Industrieroboter
IT	Informationstechnik
I4.0	Industrie 4.0
LAN	Local Area Network
LH	Left Hand
MES	Manufacturing Execution System
MQTT	Message Queuing Telemetry Transport
MRO	Maintenance, Repair and Operations
OPC UA	Open Plattform Communication Unified Architecture
RAMI4.0	Referenzarchitekturmodell Industrie 4.0
RH	Right Hand
SCADA	Supervisory Control and Data Acquisition
SiOME	Siemens OPC UA Modeling Editor
SPS	Speicherprogrammierbare Steuerung
TCP	Transmission Control Protocol
ToF	Time-of-flight

TSN	Time-Sensitive Networking
UDP	User Datagram Protocol
UML	Unified Modeling Language
URL	Uniform Resource Locator
ZVEI	Zentralverband Elektrotechnik- und Elektroindurstrie

1 Einleitung

Als Resultat eines global verteilten Käufermarktes stehen zurzeit viele Unternehmen unterschiedlichster Branchen weltweit vor der Herausforderung, dass Optimierungspotenzial bestehender Arbeits - und Produktionsabläufe mithilfe digitaler Neuerungen voll auszuschöpfen. Dies ist notwendig um sich auch künftig global gegen Konkurrenten behaupten zu können. Durch ein steigend volatiles Kaufverhalten an vielen Märkten und dem zunehmenden Anspruch an flexiblen und innovativen Produkten steigt der Grad der Industrialisierung einzelner Branchen beinahe exponentiell. Im Rahmen dieser Herausforderung gilt insbesondere die Globalisierung als maßgeblicher Beschleunigungsfaktor der industriellen Digitalisierung. Nach Petersen entsteht durch die ökonomische Globalisierung ein unabwendbar ansteigender Wettbewerbsdruck, der in einem Zwang nach Kostensenkung und Produktionsoptimierung resultiert. Die Optimierung kann anschließend durch den verstärkten Einsatz von zukunftssicheren Technologien planungssicher realisiert werden. [Pet15]

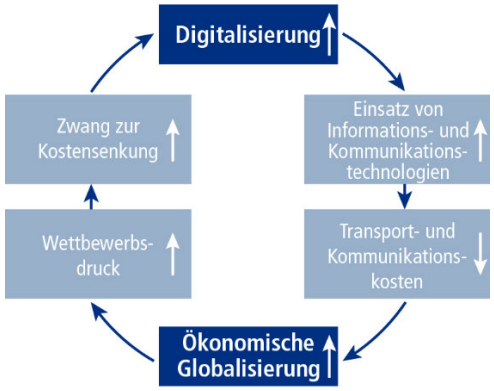


Abbildung 1.1: Wechselwirkungen zwischen Globalisierung und Digitalisierung [Pet15]

Auf die Mechanisierung der Produktion und Logistik folgt dahingehend zuerst die Automatisierung einzelner Prozessschritte und dann die Digitalisierung ganzer Produktionszweige. Was umgangssprachlich auch als vierte Industrielle Revolution bezeichnet wird, ist seit 2011 unter der Projektbezeichnung “Industrie 4.0“ (I4.0) im Rahmen des Zukunftsprojektes der Arbeitsgruppe “Plattform 4.0“ des Bundesministerium für Bildung und Forschung (BMBF) zusammengefasst. Ziel des Projektes im Rahmen der sogenannten Hightech-Strategie (HTS 2025) der Bundesregierung ist, die Förderung und aktive Unterstützung von Unternehmen und Forschungseinrichtungen zur Migration und planungssicheren Integration moderner Technologien in (bestehende) Industrieanlagen [BMB21]. Um die Produktivität dieser Anlagen nachhaltig steigern zu können, ist es notwendig den digitalen Wandel innerhalb der Unternehmen auf

Arbeits- und Prozessstrukturen auszuweiten. Dies führte im Jahr 2016 zur Auflage der Förderprogrammlinie “Zukunft der Arbeit“ des BMBF. Ziel dieser Programme ist die Förderung von Forschungsprojekten unterschiedlicher Institute, um neue Methodiken und Technologien zu entwickeln, welche Arbeitsprozesse sowohl im Sinne der Arbeitnehmer, als auch der Unternehmen möglichst flexibel und kongruent gestalten lassen. [BMB16]
Neben Arbeits- und Prozessabläufen liegt der Fokus von I4.0 zudem verstärkt auf der Vernetzung von Produktions- und Logistiksystemen. So sollen physische Objekte, wie bspw. Maschinen, Steuerungen oder Sensorik, mit virtuellen Systemen in der Cloud verbunden werden. Diese Verknüpfungen führen zu sogenannten Cyber-Physical-Systems (CPS)[Poe18]. Sie zeichnen sich dadurch aus, dass sie eine vertikale herstellerübergreifende Verbindung schaffen, die es physischen Komponenten erlaubt über eine Dateninfrastruktur (bspw. das Internet) zu kommunizieren. Die Vielzahl dieser Anwendungen und Systeme wird unter dem Begriff des Internet der Dinge (engl. *Internet of Things* (IoT) oder auch *Industrial Internet of Things* (IIoT)) zusammengefasst. Die hierdurch entstehenden Produktionssysteme sind über die Bereichsgrenzen hinaus vernetzt und können so zentral gesteuert und überwacht werden. Dies schafft eine neue Möglichkeit mittels intelligenter Softwarelösungen eine neue Stufe der Autonomie und Selbstüberwachung von Produktionsanalgen zu erreichen. Die über die CPS aufgenommenen Daten können so in der Cloud zentral und von überall zugänglich zusammengefasst, verarbeitet und weitergeleitet werden. Wie in Abbildung 1.2 dargestellt, lässt sich der Wandel von der Industrie 3.0 hin zur Industrie 4.0 nicht mehr durch ein klassisch, hierarchisches System verbildlichen. Vielmehr entsteht ein Netzwerk an Produkten der I4.0, in welchem jede Komponente in der Lage ist mit jeder anderen Komponente zu kommunizieren.

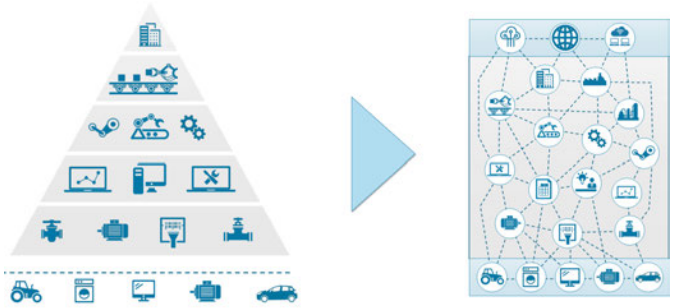
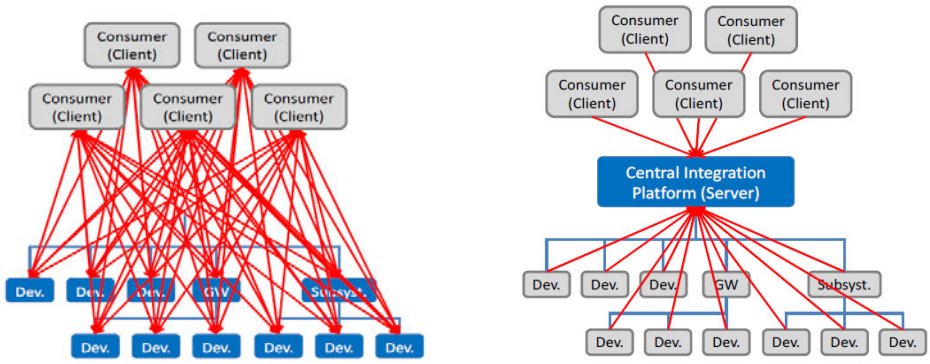


Abbildung 1.2: Vergleich Industrie 3.0 und 4.0 [Pla18b][Pla18c]

Um solch eine vernetzte Kommunikationsstruktur zu realisieren, muss zunächst sicher gestellt werden, dass jede Komponente die nötige Kommunikationsfähigkeit bereitstellen kann und die Kommunikation unabhängig von System und Hersteller einheitlich gestaltet wird. Dies ist zudem unabdingbar, um die Austauschbarkeit von Subsystemen in Produktionsanlagen nahtlos und flexibel gestalten zu können. Nur so kann der gewünschte Grad an Modularität und Wandlungsfähigkeit solcher Anlagen erreicht werden. OPC UA stellt hierbei den zentralen Standard für einen plattformunabhängigen Datenaustausch dar und wird bereits seit 2010 für die Vernetzung von Produktionssystemen aktiv eingesetzt. Das serviceorientierte Datenprotokoll wurde im Jahr 2008 als Nachfolger des OPC-Protokolls von der OPC Foundation vorgestellt und gilt bereits seit 2018 als festes Kriterium des Zentralverbands Elektrotechnik- und

Elektronikindustrie (ZVEI) für Produkte der Industrie 4.0 [Bil18]. Zudem empfahl die Arbeitsgruppe der Plattform 4.0 OPC UA als beste Möglichkeit zur Implementierung des Kommunikations-Layer im Sinne des RAMI 4.0-Referenzmodells [Hop17]. Ziel des Protokolls ist somit der generische und vor allem plattformunabhängige Datenaustausch zwischen Subsystemen eines Netzwerkes.

Mit jeder vernetzen Komponente innerhalb einer Anlage entsteht ein weiterer Endpunkt, der Daten und Services zur Überwachung und Steuerung der Komponente via OPC UA bereitstellt. Neben einer geographischen Verteilung dieser Endpunkte, kann es so auch zu einer hierarchischen Einordnung innerhalb des Netzwerkes kommen. So können bspw. Maschinen einer Station innerhalb einer Produktionslinie aus Sicht eines *Supervisory Control and Data Acquisition System* (SCADA) gebündelt ausgewertet werden, da sie möglicherweise zur Zeit am selben Fertigungsprozess beteiligt sind. Dies bedarf jedoch aktuell einer separaten Client-Server-Verbindung zu jeder beteiligten Komponente. Durch die stetige Ergänzung neuer Systeme einer solchen Anlage, wird schnell eine große Anzahl an sicheren Verbindungen benötigt, die den Datenverkehr innerhalb des Netzwerkes enorm erhöhen. Um die Anzahl der Verbindung und somit die Last des Netzwerkes zu reduzieren, werden zentrale Verarbeitungsstellen benötigt, die die Daten aus den unterschiedlichen Quellen automatisiert und gebündelt (aggregiert) für die Client-Zugriffe bereitstellen können (siehe Abb.1.3).



(a) Klassisches Client-Server-Netzwerk

(b) Client-Server-Netzwerk mit zentraler Verarbeitungskomponente

Abbildung 1.3: Wirkungsweise einer zentralen Aggregation von Datenströmen [GBB+14]

Unter Berücksichtigung der Spezifikationen der OPC Foundation zur Implementierung und Verwendung von OPC UA und den Kriterien für Industrie-4.0-Produkte des ZVEI, müssen zentrale Integrationspunkte softwaretechnisch umgesetzt werden, um den Datenaustausch einer Vielzahl an OPC UA Servern ressourcenschonend zu realisieren. Hierbei können diese Softwarelayer bspw. bei der Aggregation von Feldgeräten einer Maschine bis hin zur Kommunikation eines *Manufacturing Execution System* (MES) mit einer gesamten Anlage eingesetzt werden.

1.1 Problemstellung und Motivation

Das Fraunhofer-Institut für Fertigungstechnik und angewandte Materialforschung (IFAM) in Stade arbeitet an modularen Fertigungssystemen, die eine möglichst flexible und wandlungsfähige Produktion von Flugzeugkomponenten im Bereich der zivilen Luftfahrt ermöglichen sollen. Der Fokus der Abteilung "Automatisierung und Produktionstechnik" liegt hierbei insbesondere auf der Kommunikation mehrerer Teilsysteme zur zentralen Steuerung und Überwachung des gesamten Fertigungssystemes. Hierbei sollen alle eingebunden Subsysteme über eine Kommunikationsschnittstelle nach dem OPC UA Standard verfügen, sodass ein flexibel und modular aufgebautes Netzwerk bereitgestellt werden kann. Ein direkter Vorteil der Modularität ist die Austauschbarkeit einzelner Teilsysteme durch die plattformunabhängige Kommunikationsarchitektur. Bereits bei kleineren Versuchsaufbauten kann es so, durch eine Vielzahl an angebundenen Feldgeräten, wie bspw. Sensoren, Steuerungen und Messtechnik zu einer Anhäufung an Client-Server-Verbindungen innerhalb eines lokalen Netzwerkes kommen. Dies resultiert zumeist in einer manuellen- und somit zeit- und kostenintensiven Konfiguration einzelner Schnittstellen an einzelnen Serverendpunkten. Durch die Implementierung einer automatisierten und zentralisierten Aggregation kann das Fraunhofer IFAM zeigen, dass die Anzahl der Client-Server-Verbindungen in solch einem Netzwerk möglichst niedrig gehalten und die Anzahl der Endpunkte zur Bereitstellung von Daten reduziert werden kann. Dies ist insbesondere dann sinnvoll, wenn im späteren Einsatz des Gesamtsystems ein übergeordnetes Produktionsleitsystem an die Teilsysteme angebunden werden soll und durch die Aggregation der Teilsysteme nur noch eine Verbindung notwendig ist (Siehe Abb. 1.3).

1.2 Zielsetzung und struktureller Aufbau der Arbeit

Zur Implementierung einer Softwarelösung für die automatische Aggregation einer Vielzahl von generischen OPC UA Servern, wird vorab in Kapitel 2 dieser Ausarbeitung, ein detaillierter Blick auf den aktuellen Stand der Technik geworfen. Hierzu werden zunächst grundlegende Systematiken und Funktionen von OPC UA anhand der Spezifikationen erläutert, um im späteren Verlauf des Kapitels bereits existierende Softwarelösungen für die Aggregation zu betrachten. Im dritten Kapitel wird ein Anwendungsszenario anhand des Projektes MFlex2025 des Fraunhofer IFAM analysiert, in welchem die Kommunikation mehrerer Feldgeräte über OPC UA realisiert werden soll. Das vierte Kapitel befasst sich, aufbauend auf Kapitel 2, mit der softwaretechnischen Implementierung eines Aggreagtionsservers in Python. Hierbei werden zunächst Anforderungen an die Software gestellt, worauf aufbauend ein Konzept zur Aggregation relevanter Daten und Bereitstellung von Services des OPC UA-Protokolls entwickelt wird. Anschließend werden die verwendeten Technologien zu Implementierung vorgestellt, zu denen unter anderem eine Software-Bibliothek zählt, die grundlegende Funktionen zur Client-Server-Kommunikation enthält. Abschließend wird in Kapitel 4 detailliert auf die Umsetzung der Aggregation in Python eingegangen. Im letzten Kapitel wird die Softwareimplementierung hinsichtlich des Testszenarios evaluiert und Verbesserungspotenzial in Form eines Ausblicks aufgezeigt.

2 Stand der Technik

In diesem Kapitel wird der aktuelle Stand der Technik betrachtet. Hierzu wird zunächst die Digitalisierung in der Produktion allgemein thematisiert, bevor anschließend einige wichtige Grundlagen des Kommunikationsprotokolls OPC UA näher erläutert werden. Darauf aufbauend werden abschließend drei theoretische Konzepte zur systematischen Aggregation einer beliebigen Anzahl generischer OPC UA Server vorgestellt.

2.1 Automatisierung und Digitalisierung in der Produktion

Wie bereits in der Einleitung angesprochen bildet die angestrebte Vernetzung industrieller Komponenten der Vision Industrie 4.0 keine einfache hierarchische Struktur mehr ab. Durch die Kommunikation zwischen Komponenten unterschiedlicher Hierarchieebenen, muss das Gesamtsystem daher vielmehr als Netz unterschiedlicher Teilsysteme betrachtet werden. Hierdurch reicht das Modell der klassischen Automatisierungspyramide, wie sie bspw. nach Siepmann [RS16] basierend auf der DIN EN 62264 aufgeführt wurde, fortan nicht mehr aus, um die Komplexität und Einordnung von I4.0-Komponenten abzubilden. Siepmann arbeitet hier mit einer in 5 Hierarchieebenen aufgeteilten Darstellung. Wie in Abbildung 2.1 aufgezeigt, ordnet er die Automatisierungspyramide nach der hierarchischen Relevanz einzelner Systeme innerhalb der automatisierten Fertigung. So kommunizieren die Teilsysteme lediglich mit Komponenten der direkt unter- oder übergeordneten Ebenen. Das Modell der Automatisierungspyramide diente hierbei unter anderem als Leitfaden zur Implementierung und Verknüpfung von Leitsystemen der Produktion und Systemen der Unternehmensführung, basierend auf Überlegungen aus den achtziger Jahren. Die Pyramide wurde seitdem mehrfach überarbeitet und so existieren, neben dem Model nach Siepmann, eine Vielzahl unterschiedlicher Interpretationen.

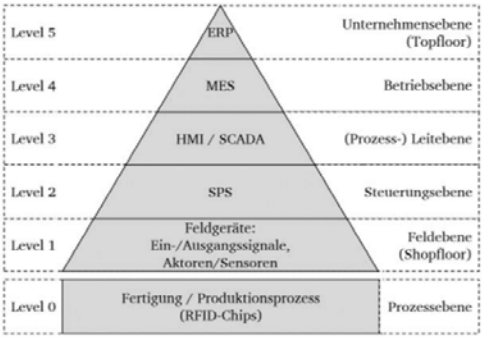


Abbildung 2.1: Automatisierungspyramide nach Siepmann 2016 [RS16]

Um eine strukturell geordnete und vor allem einheitliche Herangehensweise an die Implementierung der Industrie 4.0 zu kommunizieren, hat die Arbeitsgruppe der Plattform I4.0 ein dreidimensionales Referenzarchitekturmodell mit der Bezeichnung *RAMI 4.0* erstellt und in der IEC PAS 63088 spezifiziert. Dieses Modell stellt mittels Schichtsystem (engl. *Layer*) einzelne Kernbereiche der Industrie 4.0 separiert dar, um so komplexe Abläufe übersichtlich aufgeteilt in einzelne Pakete über den Produktlebenszyklus zu verdeutlichen. RAMI 4.0 kann somit als Rahmen grundlegender Mindestanforderungen an Systeme und Komponenten der I4.0 interpretiert werden und soll dafür sorgen, dass künftig alle Teilnehmer der Industrie 4.0 untereinander möglichst einheitlich kommunizieren können. [Pla18a]

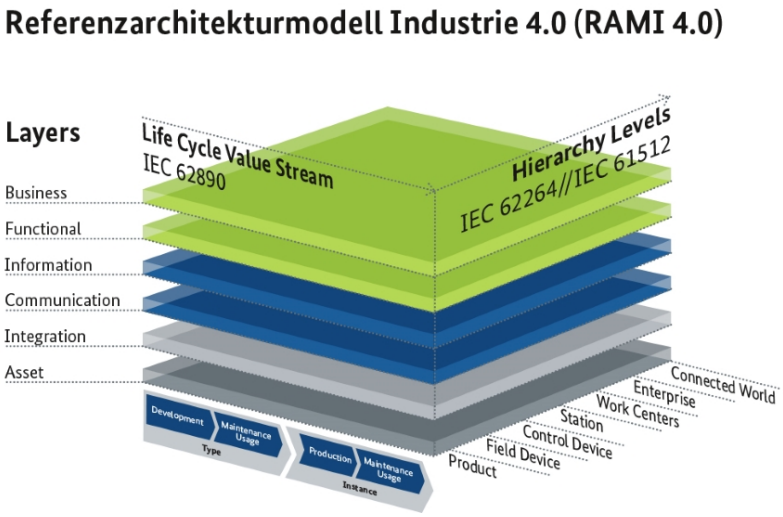


Abbildung 2.2: Referenzarchitekturmodell RAMI 4.0 [Kos15]

Wie in Abbildung 2.2 zu sehen, wird die klassisch hierarchische Aufteilung der Automatisierungspyramide (engl. *Hierarchy Levels*) im RAMI-Modell durch den Wertschöpfungslebenszyklus (engl. *Life Cycle Value Stream*) ergänzt. Er charakterisiert die Komponente, in diesem Zusammenhang auch als Asset bezeichnet, über den gesamten Produktlebenszyklus hinweg mit bestimmten Zuständen, an bestimmten Orten und zu bestimmter Zeit. Somit soll jedes Asset während der gesamten Lebensdauer stetig überwacht und die eben aufgeführten Kenngrößen dokumentiert werden. Der größte Unterschied zur klassischen Automatisierungspyramide besteht allerdings darin, dass das RAMI 4.0-Referenzmodell die digitalen und physischen Eigenschaften bzw. Funktionalitäten einer Industrie 4.0-Komponente durch das Schichtsystem (vertikale Achse) voneinander getrennt aufzeigt. Hier wird das digitale Abbild einer I4.0-Komponente schichtweise erläutert. Die in diesen Schichten enthaltenen Eigenschaften und Funktionalitäten sind von essentieller Bedeutung für den Integrationsprozess einer I4.0-Komponente, da ohne die Berücksichtigung dieser, keine nahtlose Integration in ein I4.0-System garantiert ist. [Pla18a]

So befindet sich das Asset als physisches Objekt auf dem untersten Layer. Der darüber liegende Integration-Layer dient als Übergang der physikalischen Welt in die digitale Welt der Informationen. Bei einem Industrieroboter würde diese Funktion bspw. der Controller übernehmen. Bei Implementierung dieses Layer wird also grundlegend festgelegt, welche Informationen und Funktionen später digital verfügbar sein können. Aufbauend darauf legt der Communication-Layer fest, wie der Zugriff auf diese Daten

von außerhalb des Systems gestaltet sein soll. Hier sollte OPC UA als grundlegende Kommunikationsarchitektur implementiert werden. Der darüber liegende Information-Layer beschreibt schließlich die (modifizierten) Daten die vom Asset bereitgestellt werden. Im Functions-Layer können basierend auf einem Teil der bereitgestellten Daten im Anschluss, die physischen Funktionalitäten des Assets digital abgebildet werden. Abschließend repräsentiert der Business-Layer den ökonomischen Zusammenhang der I4.0-Komponente im Gesamtsystem und kann so dafür genutzt werden, um bspw. Risiken zu überwachen oder Stückkosten zu senken. [Pla18a]

Die Layer im RAMI 4.0-Referenzmodell oberhalb des Asset-Layer sind nach DIN SPEC 91345 auch als "virtuelle digitale und aktive Repräsentanz einer I4.0-Komponente im I4.0-System" definiert ([Pla20] S.8). Diese Repräsentanz wird auch als Verwaltungsschale bezeichnet. Demzufolge besteht eine I4.0-Komponente jeweils aus einem physischen Asset, welches durch die Verwaltungsschale in das I4.0-System integriert wird.

2.2 Grundlagen OPC UA

Die Informationsmodellierung mit OPC UA gilt insbesondere für Einsteiger oft als komplexe und undurchsichtige Herausforderung. So kann die Vielzahl an Begriffen und Definitionen den Umgang mit OPC UA anfangs erschweren, was oft zum Vergleich von OPC UA mit vermeintlich besser aufgebauten Netzwerkprotokollen, wie bspw. *Message Queuing Telemetry Transport* (MQTT) oder *Advanced Message Queuing Protocol* (AMQP) führt. Der Fragestellung nach OPC UA oder MQTT als besseres Kommunikationsprotokoll, wird später im Kapitel detailliert nachgegangen. Die konforme Umsetzung eines individuellen Anwendungsszenarios mit OPC UA kann für Einsteiger daher meist nicht direkt ersichtlich sein. OPC UA bietet durch die eindeutige Definition und Regelung mittels Spezifikationen einen klaren und einheitlichen Weg, mit äußerst vielen Möglichkeiten zur Implementierung eines interoperablen Framework. Zudem lobte das Bundesamt für Sicherheit in der Informationstechnik (BSI), OPC UA für seine hohen Sicherheitsstandards ([SMA+18] S.15).

Wie bereits in Kapitel 1 erwähnt, gilt OPC UA nach dem ZVEI als festes Kriterium für Komponenten der Industrie 4.0. Dies veranlasst aktuell viele Hersteller weltweit dazu, sich mit der Thematik der Informationsmodellierung näher auseinanderzusetzen, um ihre Produkte ebenfalls für I4.0 zu rüsten [Bil18]. So definieren Anwender die Funktionalitäten und Parameter ihres Systems mit OPC UA und können die IT-Sicherheit mit individuellen Zugriffsrechten versehen, um unerwünschten Datenaustausch zu vermeiden. Die Implementierung von OPC UA wird hierbei in aktuell 24 Spezifikationen, welche von der OPC Foundation herausgegeben und stetig aktualisiert werden, definiert. Sie spezifizieren die grundlegenden Regelungen zur skalierbaren und sicheren Informationsverarbeitung, sowie dem Datenaustausch via OPC UA. Des weiteren gibt die OPC Foundation Herstellern und Verbänden die Möglichkeit, zur Definition domänenspezifischer Spezifikationen, genannt *Companion Specifications*. Diese Companion Specifications ermöglichen die Beschreibung von Regelungen, Methoden und Ereignissen, welche spezifisch für bestimmte Branchen oder Maschinenarten übergreifend gelten sollen. Dies kann bspw. Zulieferern

zur Orientierung bei der Implementieren von OPC UA in ihren eigenen Produkten dienen ([SMA+18] S.27).

Um Daten auszutauschen bietet OPC UA zwei Mechanismen, welche unabhängig voneinander verwendet werden können. Der Server stellt hierbei jeweils die Daten oder Dienste bereit, die der Client abzufragen versucht. Dies kann ein sog. bestätigter oder unbestätigter Informationsaustausch sein:

Client-Server-Modell

Die meisten Anwendungen von OPC UA basieren auf dem Client-Server-Modell, welches mittels *Request-Response*-Systematik den Datenaustausch realisiert. Dies bedeutet im Detail, dass jeder Informationsaustausch bidirektional bestätigt wird. So wissen beide Kommunikationspartner, Client und Server, ob die Datenpakete beim jeweils Anderen angekommen sind. Stellt bspw. der Client eine Anfrage oder erhält Informationen, bestätigt er dies dem Server automatisch. So kann sichergestellt werden, dass keine Pakete verloren gehen und das der Server als übergeordnete Komponente und letzte Instanz im Kommunikationsprozess, die Datenhoheit behält. Dieses Modell wird daher oft mithilfe einer synchronen Kommunikationsweise genutzt (siehe Abbildung 2.3), weshalb es eher nicht für den Echtzeit-Datenaustausch, wie bspw. beim Time-Sensitive Networking (TSN), geeignet ist. Zu beachten ist hier zudem der Unterschied zum Peer-To-Peer Modell. Bei diesem können sowohl der Client, als auch der Server auf ein und dem selben Gerät ausgeführt werden ([SMA+18] S.20/64).

Publisher-Subscriber-Modell

Im Gegensatz zum Client-Server-Modell basiert das Publisher-Subscriber-Modell auf einer asynchronen Informationsverteilung (siehe Abbildung 2.3). Im Detail bedeutet dies, dass der Publisher (Server) die ausgewählte Daten an eine Multicast-Gruppe versendet. Die Subscriber (Clients) empfangen die Daten anschließend, indem sie die Multicas-Gruppe abonnieren. Der Auslöser für den Server die Informationen zu versenden ist hierbei meist ein Event, wie bspw. eine Datenänderung. Ein Vorteil dieser Kommunikationsweise liegt hierbei insbesondere darin, dass der Absender die Empfänger nicht kennen muss. Dies entkoppelt die beiden Kommunikationspartner voneinander, was zudem dazu führt, dass der Server nicht auf eine Rückmeldung der Clients warten muss. Aufgrund dessen wird das Publisher-Subscriber-Modell häufig bei Echtzeitanwendungen genutzt, wie bspw. bei der Übertragung über UDP/IP. AMQP und MQTT nutzen ebenfalls die Publisher-Subscriber-Kommunikation, zählen jedoch nicht zu den echtzeitfähigen Protokollen. Zudem ist der Einsatz des Publisher-Subscriber-Modells insbesondere dann sinnvoll, wenn ein konstanter Datenaustausch zur Blockierung bestimmter Dienste des Servers führen kann. In diesem Fall wird beim Publisher-Subscriber-Modell eine Art Warteschlange eingesetzt, die dafür genutzt werden kann, um Informationen zu übertragen ohne durch die Verarbeitungsgeschwindigkeit des abonnierenden Systems gebremst zu werden. Ein entscheidender Nachteil hierbei ist jedoch, dass der Herausgeber der Daten nicht überprüfen kann, ob die Informationen beim Abonnenten ankommen. Zudem kann er im Fall von gefilterten Informationen nicht zwischen den Abonnenten einer Gruppe unterscheiden, wodurch je nach Beschränkung der jeweiligen Konsumenten, separate Ausgabekanäle eröffnet werden müssen. Dies reduziert zumeist die Gesamtanzahl der Abonnenten enorm. ([SMA+18] S.20/66 ff.)

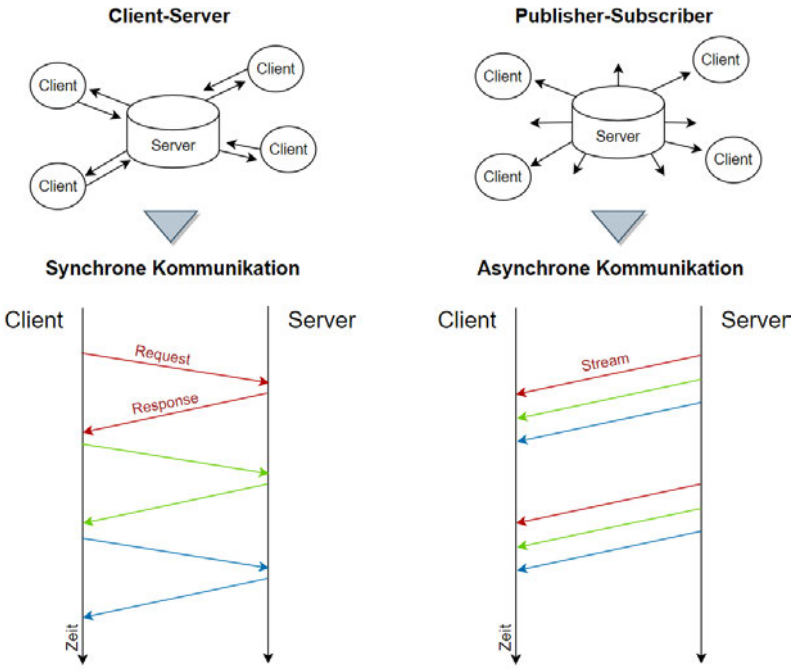


Abbildung 2.3: Synchrone und Asynchrone Kommunikation

Zu beachten ist, dass die von OPC UA bereitgestellten Mechanismen vollkommen gelöst von den eigentlichen Kommunikationsprotokollen zu betrachten sind. So werden für das Client-Server-Modell die Kommunikationsprotokolle TCP/IP bzw. HTTPS, für eine verschlüsselte Datenübertragung, genutzt. Für einen Informationsaustausch nach dem Publisher-Subscriber-Modell werden hingegen doe bereits erwähnten Protokolle UDP, AMQP und MQTT verwendet. Daraus resultiert, dass sich die oben genannte Fragestellung nach OPC UA oder MQTT, aus Sicht der OPC Foundation gar nicht erst stellt.
Bei prozessspezifischer Implementierung von OPC UA kann so, basierend auf den oben vorgestellten Mechanismen und Spezifikationen, ein digitales Abbild des phyischen Systems in der Verwaltungsschale erstellt werden. Unter Verwendung von Companion Specifications sind Hersteller zudem in der Lage, bereits vorhandene Systeme vereinheitlicht mit einer OPC UA-Schnittstelle auszustatten.

2.3 OPC UA Adressraum

Wie bereits im vorigen Kapitel angeschnitten, wird bei der Implementierung von OPC UA, zwischen dem Transportmechanismus und der Datenmodellierung unterschieden. So beschreiben sowohl die Spezifikationen der OPC Foundation, als auch die Companion Specifications lediglich die Syntax und Semantik, wie die Daten modelliert werden, jedoch nicht die Methodik wie sie übertragen oder softwaretechnisch implementiert werden.
Ein OPC UA Server besteht aus einem Adressraum, in welchem einzelne Daten in Form von Knoten (engl. *Nodes*) dargestellt sind. Die Strukturierung dieser Knoten, deren Eigenschaften (engl. *Attributes*) und Verbindungen (Referenzen, engl. *References*) werden vom Informationsmodell des Adressraumes vorgegeben. Der Adressraum eines OPC UA Server ist hierbei als vernetzter Graph zu interpretieren, da

bei OPC UA jeder Knoten mit jedem anderen Knoten durch eine Referenz verbunden werden kann (siehe Abb.2.4). Bei den Referenzen wird zwischen hierarchischen- und nicht-hierarchischen-Referenzen unterschieden. Zur vereinfachten Darstellung werden häufig die hierarchisch vernetzten Knoten in Form von Baumstrukturen dargestellt. Das Informationsmodell des Servers setzt sich aus Knotenmengen, sogenannten *NodeSets* zusammen, welche aufbauend aufeinander definiert sein können. Jeder Knoten eines NodeSet ist hierbei einem bestimmten Namensraum (engl. *Namespace*) zugeordnet. Ein NodeSet kann aus einem oder mehreren Namensräumen bestehen, welche sich durch ihre *Namespace-URL* definieren. Alle Namensräume eines Adressraumes werden auf dem Server im *NamespaceArray* gespeichert. Um einen Knoten später einem Namensraum eindeutig zuordnen zu können, wird dem Knoten der Index des Namensraumes im NamespaceArray als Information angehängt. In Kombination mit einer Kennung, dem *Identifier*, entsteht so für jeden Knoten eine, im Adressraum des Servers einzigartige *NodeID* zur eindeutigen Identifizierung. Der Identifier kann hierbei bspw. durch einen ganzzahligen Wert oder einen String (Zeichenfolge) repräsentiert werden.

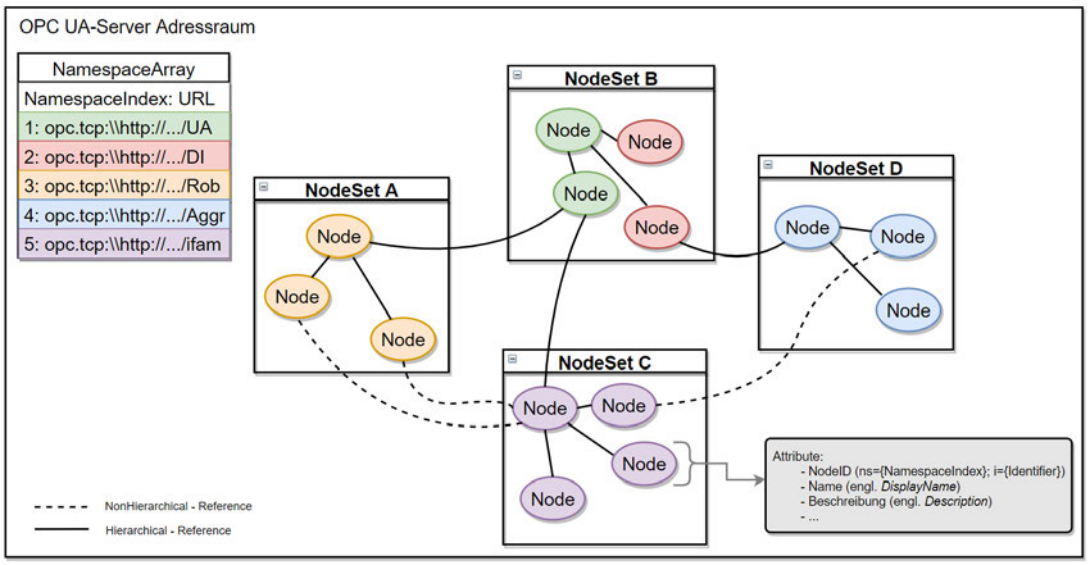


Abbildung 2.4: Aufbau des Adressraumes eines OPC UA Server

Als Grundlage für alle künftigen NodeSets hat die OPC Foundation zudem ein Basis-NodeSet veröffentlicht, welches im Standard IEC 62 541 festgehalten ist ([SMA+18], S.47 ff.). Dieses NodeSet, bestehend aus einem Namensraum mit der URL: “http://opcfoundation.org/UA/“, definiert hierbei Knoten und Referenzen, welche als Grundlage für jeden OPC UA Server dienen sollen. Dem Namensraum des Basis-NodeSets wird zudem im NamespaceArray des Servers stets der Index 0 reserviert. Um die in den NodeSets definierten Informationen speichern und austauschen zu können, beschreibt die OPC Foundation im Part 6 ihrer Normreihe zu OPC UA, ein Schema zur Überführung der Informationsmodelle in das XML-Format. Auf Basis dieses Schemas können NodeSets mithilfe sogenannter *OPC UA Modeling Editoren* manuell konstruiert und anschließend als XML-Datei exportiert werden. Im Anschluss kann das NodeSet dann vom Server eingelesen und dem Adressraum hinzugefügt werden.

Da OPC UA unter Verwendung von objektorientierten Ansätzen entwickelt wurde, hat die OPC Foundation in Spezifikation Part 3, Knoten anhand der nachfolgenden Knotentypen (engl. *NodeClasses*) klassifiziert:

- Objekte (engl. *Objects*)
- Variablen (engl. *Variables*)
- Methoden(engl. *Methods*)
- Datentypen (engl. *DataTypes*)
- Referenztypen (engl. *ReferenceTypes*)

Jeder Knoten enthält hierbei, neben der Typenbezeichnung, weitere Attribute, welche es dem Informationsmodell erlauben den Knoten zu identifizieren, beschreiben und einzuordnen. So enthält bspw. jedes Objekt im Adressraum Informationen über seinen Namen, eine Beschreibung, die jeweilige Zugriffsberechtigung und seine NodeID. ([OPC16], [SMA+18] S.49 f.)

Objekte, Objekttypen und Variablen

Mithilfe von Objekt-Knoten lassen sich physische Objekte im Adressraum eines OPC UA Servers repräsentieren. Hierbei kann es sich sowohl um einzelne Komponenten, als auch um ganze Subsysteme einer Anlage handeln. So kann bspw. ein klassischer Sechsachsroboter mit Achsen, Motoren, Getrieben und Steuerung, aus einzelnen Objekten im Adressraum zusammengesetzt werden. Die Subkomponenten des Roboters könnten so via OPC UA z.B. separat überwacht werden.
Um ein neues Objekt dem Adressraum hinzuzufügen, bedarf es vorerst einer Definition des Objekttypen im Objekttypen-Verzeichnis des Servers. In Form dieses Objekttypen wird dort eine Art Bauplan des späteren Objektes definiert. Dies geschieht meist durch Spezifizierung eines bereits definierten Typen mittels *HasSubType*-Referenz. Hierbei werden die Attribute bzw. Komponenten des bereits definierten Typen, ähnlich wie beim Prinzip der Klassenvererbung im Bereich der objektorientierten Softwareentwicklung, für die neue Typendefinition übernommen. Ist der Objekttyp definiert, kann anschließend das Objekt im Objekte-Verzeichnis des Servers erstellt werden (siehe Abb. 2.5). Zur späteren Identifizierung werden die Objekten bei Instanziierung durch eine *HasTypeDefinition*-Referenz mit ihrem jeweiligen Objekttypen verknüpft. Auch Variablen-Knoten folgen diesem Schema, wodurch bspw. mehrdimensional aufgebaute Variablen vorerst im Variablentypen-Verzeichnis zu definieren sind. Zudem müssen Variablen stets eine Referenz zum entsprechenden Datentypen aufweisen, um festzulegen welche Art von Daten sie speichern können.
Die nachfolgende Abbildung 2.5 zeigt beispielhaft die vereinfachte Definition und Instanziierung eines Industrieroboters der SCARA-Bauweise mit vier Achsen. Die Baumstruktur auf der linken Seite der Abbildung zeigt hierbei die Typdefinitionen, während die rechte Seite die Objektinstanzen darstellt. ([SMA+18] S.50 f.)

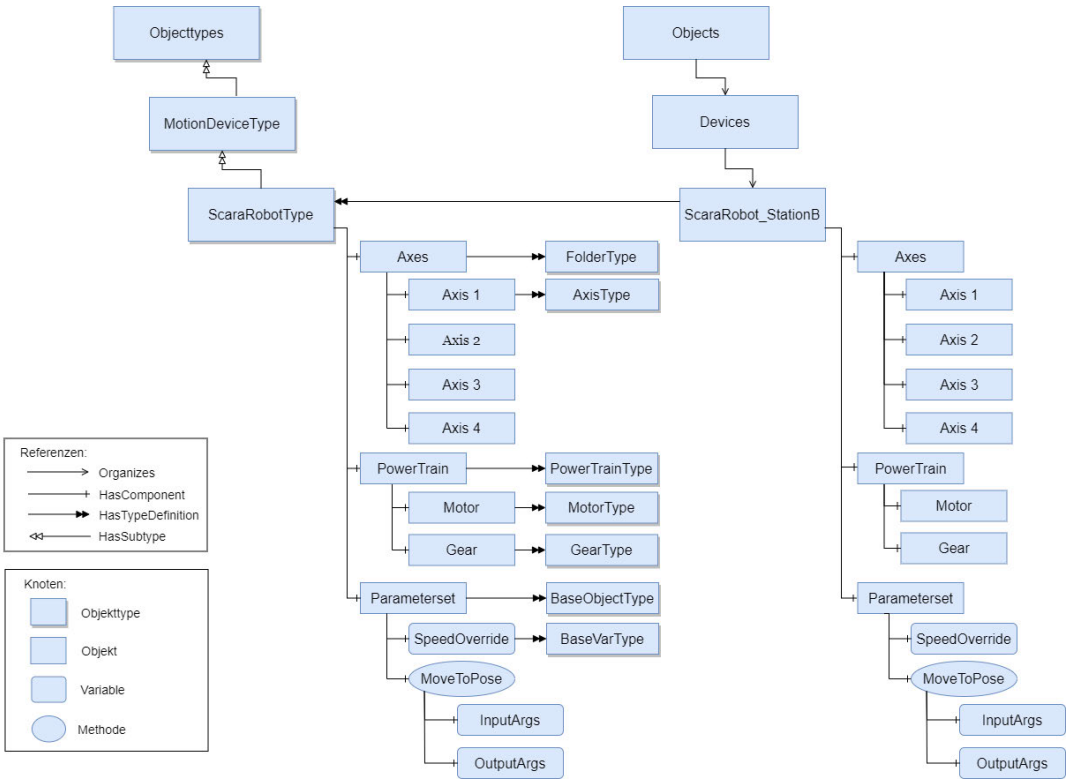


Abbildung 2.5: Objekt und Objekttypen

Referenzen

Referenzen sorgen für Verbindungen zwischen unterschiedlichen Knoten innerhalb des Adressraumes und ihre Bedeutung wird in Referenztypen im entsprechenden Verzeichnis des Servers definiert. Sie werden, wie bereits erwähnt, unterteilt in hierarchische- und nicht-hierarchische Referenzen. Da Referenzen stets gerichtet Verbindungen sind, ist neben den zu referenzierenden Knoten auch die Orientierung der Verbindung (Forward oder Invers) festzulegen. Zumeist wird dies durch die Definition eines Start- bzw-Endknoten festgelegt. Eine bidirektionale Referenzierung zwischen zwei Knoten ist ebenfalls möglich. Neben den in Abbildung 2.5 dargestellten Referenzen gibt es eine Vielzahl weiterer Referenztypen, zu denen auch Definitionen sehr spezifischer Referenzen gehören. Meist sind diese in den Companion Specifications zu finden. Bei der Darstellung von Referenzen in Diagrammen, wie in Abbildung 2.5, ist zu beachten, dass nicht für jeden Referenztyp eine standardisierte Notation zur Darstellung existiert. ([SMA+18] , S.51 f.)

Methoden

Methoden-Knoten haben im Adressraum eines OPC UA Servers eine sehr besondere Bedeutung, da sie eine Funktionalität des physischen Objektes abbilden können. Sie werden daher auch im Adressraum

stets einem übergeordneten Objekt mit entsprechender Referenz zugeordnet. Zur Abbildung einer Funktionalität bieten Methoden-Knoten dem Client die Möglichkeit mithilfe des Call-Services ausgeführt zu werden (Method-Service-Set, [OPC16]). Basierend darauf muss jede Methode mit einer Funktion softwareseitig verknüpft werden, welche aufgerufen wird sobald die Methode ausgeführt wird. Methoden können Ein- und Ausgabeparameter besitzen, deren Eigenschaften (Datentyp, Dimension etc.) zuvor im Informationsmodell zu definieren sind. Mithilfe dieser Parameter können so Argumente an die Funktion übergeben oder Rückgabewerte im Adressraum bereitgestellt werden.

Services

Wie bereits in der Einleitung erwähnt gilt OPC UA als service-orientierte Architekur, weshalb die OPC Foundation in der Spezifikation Part 4 den OPC UA Adressraum um sogenannte Services erweitert. Bei Services handelt es sich um Funktionen/Dienstleistungen des Servers, um mit dem Client interagieren zu können. Jeder Service wird hierbei einem bestimmten Service-Set zugeordnet, welches zuvor in den Spezifikationen definiert wurde. So besitzen OPC UA Server neben den Services zum hinzufügen, entfernen oder modellieren von Knoten und Referenzen (*NodeManagement*-Service-Set), unter anderem auch Funktionalitäten zum lesen und schreiben von Variablen, Objekten und deren Attributen (*Attribute*-Service-Set). Um im Adressraum von Knoten zu Knoten entlang der Referenzierungen navigieren zu können, bietet das *View*-Service-Set den *Browse*-Service. Dieser erlaubt zudem auch eine gefilterte Navigation, wodurch dem Client ermöglicht wird sich im Adressraum bspw. nur entlang bestimmter Referenzen zu bewegen. [OPC21]

Zur Überwachung einzelner Knoten bietet das *MonitoredItem*-Service-Set in Kombination mit dem *Subscription*-Service-Set dem Client die Möglichkeit, sogenannte *MonitoredItems* zu definieren und anschließend eine Subscription auf diese zu erstellen. MonitoredItems spezifizieren hierbei bestimmte Knoten (Objekte, Variablen) innerhalb des Adressraumes. Mit einer Subscription wird anschließend die Überwachung gestartet. Der Client kann nun entweder intervallbasiert mit einer bestimmten Abtastrate die Daten der Knoten auslesen, oder mittels eventbasierter Überwachung nur eine Übermittlung der Daten bei Änderung erfolgen lassen. Die Kommunikation folgt hierbei dem *Request-Response-Paradigma*, wie bereits in 2.2 beschrieben. [OPC21]

2.4 Konzepte zur Aggregation generischer OPC UA Server

Nachfolgend werden zwei wissenschaftliche Veröffentlichungen vorgestellt, welche sich schwerpunktmäßig mit der Aggregation generischer OPC UA Server befasst haben. Neben der theoretischen Vorgehensweise, werden hierbei auch Konzepte zum Aufbau einer realen Implementierung erläutert und teilweise umgesetzt, sowie getestet.

Wie bereits in der Einleitung beschrieben neigen künftige Anlagennetze zu einer stetig steigenden Anzahl an Client-Server-Verbindungen, da es im Zuge der Industrie 4.0 gängiger wird, einzelne Maschinen

und Komponenten mittels cloud-basierter Produktionssysteme zu überwachen, zu konfigurieren oder zu steuern. Um die Anzahl der Verbindungen möglichst gering zu halten und somit das Gesamtnetz vor Überlastung zu schützen, werden Integrationsplattformen auf der Informationsebene implementiert. Diese sind in der Lage Informationen einzelner Datenquellen zu bündeln und über einen einzelnen Endpunkt für Client-Zugriffe bereitzustellen (siehe Abb. 1.3). In der Veröffentlichung "OPC UA Server Aggregation – The Foundation for an Internet of Portals" [GBB+14] aus dem Jahr 2014, beschreiben Forscher der TU Ingolstadt in Kooperation mit dem ABB Corporate Research Center, ihre wissenschaftliche Untersuchung zur Implementierung einer solchen zentralen Integrationsplattform auf Basis von OPC UA. Um die Anforderungen an die Implementierung fundiert definieren zu können, hat die Arbeitsgruppe die nachfolgenden grundlegenden Bestandteile eines Informationsmodells definiert, welche bei einer Aggregation zu berücksichtigen sind:

- Typen-Informationen (Typdefinitionen)
- Struktur-Informationen (Objekte und Referenzen)
- Instanz-Informationen (Daten von Variablen und Objekten)
- Verhalten von Objekten

Diese grundlegenden Informationen müssen Integrationsplattformen den Autoren zufolge, bei einer Aggregation berücksichtigen. Ob es sich hierbei um Teilstrukturen oder ganze Namensräume handelt, spezifizieren die Forscher aus Ingolstadt nicht. Zudem stellte die Arbeitsgruppe fest, dass eine zentrale Aggregation neben der Zusammenführung von Informationen auch eine erhebliche Relevanz im Bezug auf Datensicherheit besitzt. Die Sicherheitsoptionen (Verschlüsslung und Übertragung der Daten) der zu aggregierenden Server dürfen daher nicht durch eine zentrale Aggregation außer Kraft gesetzt werden, woraus folgt, dass die Integrationsplattform für eine strikte Einhaltung der Regeln zur Benutzerverwaltung und Datensicherheit sorgen muss. ([GBB+14], S.1 f.)

Auch Forscher der Aalto Universität aus Finnland haben sich mit dem Konzept einer strukturellen Aggregation von Informationsstrukturen des OPC UA Adressraumes befasst [STE+16]. In ihrer Veröffentlichung aus dem Jahr 2016 differenzieren die Autoren hierbei insbesondere zwischen zwei spezifischen Anwendungsszenarien einer Integrationsplattform. Dem ersten Szenario zufolge könnte mittels Aggregation die Transformation des gesamten Adressraumes eines Quellservers in den eines aggregierenden Servers definiert werden, wohingegen das zweite Szenario lediglich die Zusammenführung einzelner Daten (-strukturen) betrachtet. Beide Varianten sind, laut den Forschern aus Aalto über eine regelbasierte Aggregation umsetzbar, ähnlich dem Schema der XML-Transformation. Wie bereits in 2.3 angesprochen, wird bei der XML-Transformation ein vordefiniertes Schema eingesetzt, welches eine Reihe von Transformationsregeln bereitstellt, um die Knotenstrukturen des Adressraumes eines OPC UA Servers in das XML-Format zu übersetzen und als NodeSet zu speichern. Auch bei der Aggregation von (Teil-) Strukturen kann dieses Prinzip genutzt werden. Durch die eindeutige Definition von Regeln, inwiefern bestimmte Knoten aggregiert werden sollen, ist der Anwender im Vorfeld in der Lage klare Transformationen festzulegen. ([STE+16], S.2)

Neben der Definiton von Regeln zur spezifischen Aggregation steht zudem die Kommunikation zu Quellservern als essentielle Funktion der Integrationsplattform im Vordergrund. Beide der oben beschriebenen Szenarien zur teilweisen bzw. ganzheitlichen Aggregation bedingen hierbei, nach der Arbeitsgruppe aus Finnland, Client-Verbindungen zu den Quellservern. Diese werden im Zuge des Aggregationsprozesses genutzt, um mithilfe der OPC UA Services benötigte Informationen abzurufen. Das Verhältnis von Quellservern und OPC UA Clients des aggregierenden Servers beträgt hierbei stets 1:1. Basierend auf diesen Betrachtungen stellen die Wissenschaftler aus Aalto ein konzeptionelles Design, bestehend aus drei Grundkomponenten für einen aggregierenden Server vor (siehe Abb. 2.6). Sie unterteilen den Server hierbei in den Adressraum (OPC UA Server), eine aggregierende Komponente und eine Vielzahl von OPC UA Clients. ([STE+16], S.2)

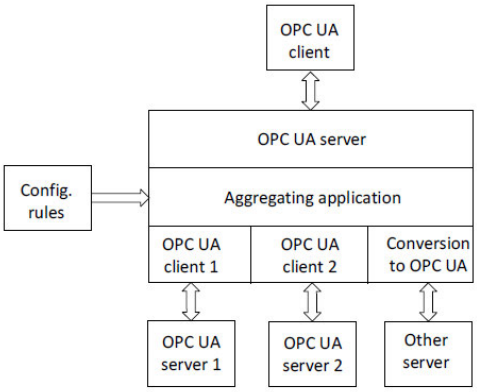


Abbildung 2.6: Konzept eines aggregierenden OPC UA Servers ([STE+16], S.2)

Während die Clients der Kommunikation mit unterliegenden Servern dienen, modelliert die aggregierende Komponente (Aggregation application, siehe Abb. 2.6) im Zuge des Aggregationsprozesses den Adressraum des OPC UA Servers. Laut der Autoren, sorgt sie hierbei für die notwendigen, regelbasierten Transformationen der Knotenstrukturen. Außerdem kann sie, je nach Design, auch für die Bereitstellung der OPC UA Services genutzt werden. Zur Definition der Transformationsregeln stellt die Gruppe der Aalto Universität zudem eine spezifische Notation vor, welche eine eindeutige Zuordnung von Knoten ermöglichen soll. Im Zuge derer werden Knoten grundlegend in rechte (engl. *Right Hand* (RH)) und linke (engl. *Left Hand* (LH)) Knoten unterteilt. Knoten, die sich im Adressraum der zu aggregierenden Quellserver befinden werden hierbei als linke Knoten bezeichnet. Als rechte Knoten hingegen, werden die Knoten gekennzeichnet, welche dem Adressraum des aggregierenden Server im Lauf der Aggregation hinzugefügt werden sollen. Auf diese Weise kann die aggregierende Komponente später die relevanten Knoten auf den Quellservern identifizieren und dem aggregierenden Server entsprechend hinzufügen. Durch eine manuelle Zuordnung von Knoten beider Seiten, bspw. in Form einer Liste, wird so ein sog. *Mapping* (dt. Kartierung) erzeugt. Dieses Mapping definiert die Regeln der Aggregation (Transformationsregeln). Laut den Autoren aus Finnland, soll das Mapping hierbei direkt in einem Text-Editor realisiert werden. Die Knoten sollen hierbei mithilfe des Pfades von der sogenannten *Root-Node* (hierarchisch oberster Knoten im Adressraum) bis zum jeweiligen Knoten definiert werden. Hierdurch wird das

Informationsmodell des aggregierenden Server Schritt-für-Schritt definiert. Anschließend entwickelten die Forscher einen Algorithmus zur Transformation, welcher das vordefinierte Mapping als Set von Regeln einließt und interpretiert. Auf Basis dessen vollzieht der Algorithmus die Aggregation der Knoten, unterteilt in zwei Abschnitte. Nach Verbindungsaufbau mit dem Quellserver durchsucht die Software im ersten Schritt vorerst alle Knoten des Quellserver und versucht jene zu identifizieren, welche für die Aggregation relevant sein könnten. Im zweiten Schritt überprüft der Algorithmus jeden gefundenen Knoten auf Übereinstimmung mit einer der vordefinierten Regeln aus dem Mapping. Sollte eine Übereinstimmung gefunden werden, so kann der Knoten entsprechend der Regel aggregiert werden. Dieses Vorgehen wird zyklisch wiederholt, bis alle Knoten des Quellservers untersucht worden. Zwischen unterschiedlichen Knotentypen unterscheiden die Forscher aus Finnland hierbei nicht, wodurch die Fragestellung entstehen könnte, inwiefern Objekt- oder Datentypen ebenfalls im Vorfeld manuell zugeordnet werden müssen. [STE+16]

Weiterführend stellt die Arbeitsgruppe der Aalto Universität zudem noch die Handhabung von OPC UA Service-Anfragen nach der Aggregation konzeptionell vor. Im Zuge des Aggregationsprozesses soll der Algorithmus demnach Informationen zu den unterliegenden Servern sammeln, wodurch im Anschluss eine weitere Software-Komponente in der Lage sein soll, Anfragen an die Quellserver weiter zu leiten. Dies geschieht hierbei unter anderem durch erneute Nutzung des Mappings. Durch die genaue Zuordnung im Mapping kann der Aggregationsserver die jeweiligen Knoten auf den Quellservern eindeutig identifizieren und Service-Anfragen entsprechend gestalten. [STE+16]

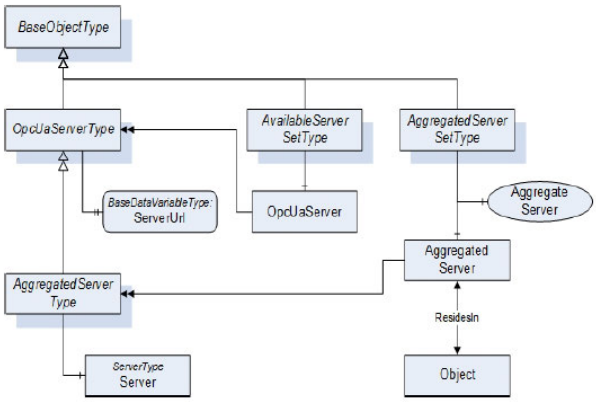


Abbildung 2.7: Steuerung der Aggregation im Adressraum [GBB+14]

Auch die Arbeitsgruppe der TU Ingolstadt stellt in ihrem Paper einen regelbasierten Ansatz zur strukturellen Aggregation generischer OPC UA Server vor. Die Wissenschaftler aus Ingolstadt haben hierbei zur Konfiguration der zu aggregierenden Server (Name, URLs etc.) ein eigenes Konfigurationstool (Aggregation Configurator) zur erleichterten Bedienung entwickelt. Das Konfigurationstool kann hierbei entweder vor Start des aggregierenden Server oder während der Laufzeit genutzt werden. Vor Server-Start werden die so erstellten Daten mithilfe einer separaten Datei oder einer Datenbank zwischengespeichert, bevor sie später von der Software eingelesen werden. Wird das Tool genutzt während der Server läuft, so kann die Konfiguration direkt über den Adressraum des aggregierenden Servers vorgenommen werden. Hierfür haben die Forscher der TU Ingolstadt ein spezielles Informationsmodell entwickelt, welches durch

die Software automatisch auf dem Server zur Verfügung gestellt wird. Es enthält neben der Konfiguration der Quellserver auch Typdefinitionen um die Aggregation bestimmter Knoten zu spezifizieren. Wie in Abbildung 2.7 dargestellt, werden hierbei im Adressraum des aggregierenden Server unter *AvailableServer*, *OpcUaServerType*-Instanzen referenziert, welche für eine Aggregation zur Verfügung stehen. Diese Instanzen enthalten die URL des Quellserver als Attribut zum späteren Verbindungsaufbau und könnten bspw. durch eine Autodetect-Funktion des aggregierenden Servers zuvor gefunden werden. Mit der Methode *Aggregate Server* kann der Client im Anschluss die Aggregation einzelner Server starten, wobei diese daraufhin dem Verzeichnis der aggregierten Server hinzugefügt und aus dem Verzeichnis der verfügbaren Server gestrichen werden. Die *OpcUaServerType*-Instanzen erlauben dem Client zudem während der gesamten Laufzeit den aktuellen Status der aggregierten Servers konstant zu überwachen. Mithilfe der *ResidesIn*-Referenz kann nach dem Aggregationsprozess jedes neu hinzugefügte Objekt auf dem aggregierenden Server mit seinem jeweiligen Herkunftsserver verknüpft werden, sodass eine Rückverfolgung erleichtert wird. [GBB+14]

Aufbauend darauf erweiterte die Forschungsgruppe aus Ingolstadt das Informationsmodell zusätzlich um Typdefinitionen zur Konfiguration der Aggregation. So kann der Client mithilfe einiger *RuleSetType*-Definitionen die Aggregation von Typen und Instanzen spezifizieren. Wie in Abbildung 2.8 dargestellt wird dem Nutzer weiterführend durch den neu definierten Service *TypeManagement* die Möglichkeit geboten, spezifische Typdefinitionen in den Adressraum des aggregierenden Server zu importieren oder vom Server zu exportieren. So können dem Adressraum des aggregierenden Server auch während der Laufzeit neue Typdefinitionen manuell hinzugefügt werden. Für diese Prozesse werden die spezifischen Methoden *ExportType* und *ImportType* vom aggregierenden Server zur Verfügung gestellt.

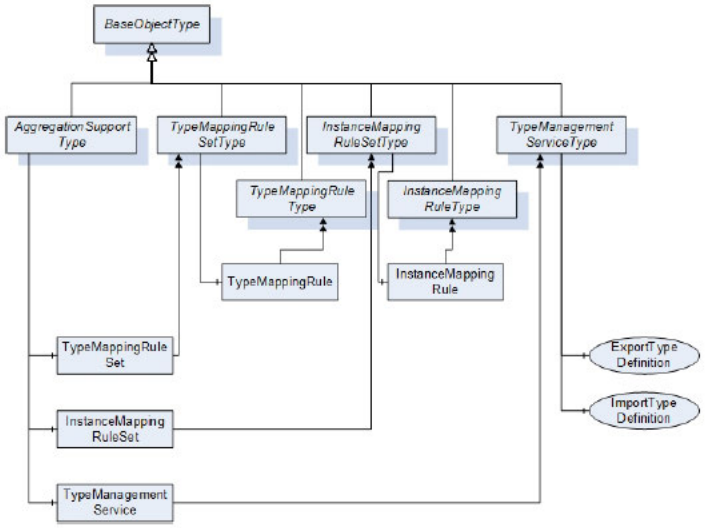


Abbildung 2.8: Konfiguration der Aggregation von Typdefinitionen und Instanzen ([GBB+14])

Im Vergleich der beiden Konzepte aus Finnland und Deutschland fällt auf, dass bei beiden Varianten die Basis einer regelbasierten Aggregation gewählt wurde. Beide Konzepte definieren hierbei grundlegende Regeln zur Identifikation der Quellserver und der zu aggregierenden Knotenstrukturen. Während

die Arbeitsgruppe der Aalto Universität jedoch auf ein durchgehend manuelles Mapping jedes einzelnen Knoten vor dem Aggregationsprozess setzt, entwickelten die Forscher der TU Ingolstandt in Zusammenarbeit mit dem ABB Research Center ein Informationsmodell zur zeiteffizienten Definition der notwendigen Verbindungsparameter und Aggregationsregeln. Das Informationsmodell wird hierbei standardmäßig im Adressraum des aggregierenden Servers zur Verfügung gestellt und ermöglicht neben der Konfiguration von Regeln zur Aggregation von Instanzen und Typen auch den separaten Start des Aggregationsprozesses mithilfe eines Methodenaufrufs, während der Server-Laufzeit. Dies stellt im Vergleich zum Konzept der Aalto Universität einen erheblichen Vorteil dar, da der Server hierbei auch nach der Aggregation durch den Nutzer weiter angepasst und überwacht werden kann. Zusätzlich können die einzelnen Aggregationen separat gestartet und überwacht werden, wohingegen die Lösung aus Finnland einen automatischen Start beim initialisieren der Software auslöst.

Im Gegensatz zur TU Ingolstadt behandelt das Konzept der Aalto Universität jedoch auch die Handhabung von OPC UA Service Anfragen. Durch das präzise festgelegte Mapping (Linke und Rechte Knoten) ist ein zusätzlicher Baustein der Software in der Lage, eine Rückverfolgung jedes Knoten im Adressraum des aggregierenden Server zum ursprünglichen Knoten auf dem Quellserver durchführen, wodurch Service-Anfragen wie bspw. Methodenaufrufe oder Subscription-Anfragen entsprechend umgesetzt werden können. Bei Gegenüberstellung der Systemarchitekturen beider Konzepte fällt auf, dass die Arbeitsgruppe der TU Ingolstadt hier ein deutlich ausgereifteres System vorstellt. Neben separaten Komponenten zur Handhabung von Typdefinitionen enthält die Software, im Vergleich zur Implementierung der Aalto Universität, auch Module zur Umsetzung von Sicherheitsanforderungen und kann den OPC UA Discovery Service zum automatischen identifizieren von OPC UA Servern im Netzwerk, einbinden.

3 Forschungsprojekt MFlex2025

Im folgenden Kapitel soll das bereits in Kapitel 1.2 erwähnten Verbundprojektes MFlex2025 detailliert erläutert werden. Hierzu wird zunächst das Gesamtziel im Rahmen des nationalen zivilen Luftfahrtforschungsprogramms VI thematisiert, worauf aufbauend die Betrachtung einzelner technischer Arbeitsziele innerhalb des Projektes folgt. Des weiteren werden die in diesem Projekt involvierten Module und Feldgeräte vorgestellt, um im Anschluss die modulare Schnittstellenkommunikation der Analage in den Kontext dieser Thesis einordnen zu können.

3.1 Projektvorhaben

Um die Produktion von modernen, zivilen Luftfahrzeugen stetig zu optimieren unterstützt die Bundesregierung bereits seit 1995 im Rahmen des Luftfahrtforschungsprogrammes LuFo, Vorhaben zur Entwicklung nachhaltiger, effizienter und leistungsfähiger Technologien in der Luftfahrt. Ziel der Programmlinie des Bundesministerium für Wirtschaft und Klimaschutz (BMWK) ist u.a. die langjährige und somit planbare Förderung von einzelnen, industriegeführten Verbundprojekten zur Effizienzsteigerung der Produktion von Flugzeugen und einer umweltfreundlicheren Luftfahrt. Die seitdem entstandene Forschungsinfrastruktur, bestehend aus Hochschulen und Forschungseinrichtungen, sowie Großversuchsanlagen und Flugversuchsträgern, startete 2019 im Zuge des industriellen Wandels der I4.0 in die sechste Auflage des LuFo (LuFo VI). Die im Rahmen dieser Auflage geförderten Forschungs- und Technologievorhaben werden hierbei in Programmlinien und Fachbereiche eingeordnet. Des weiteren wird die sechste Auflage unterteilt in Förderungsperioden, die sogenannten Programmaufrufe, welche aktuelle Herausforderungen, wie bspw. das Pariser Klimaabkommen, als Themenschwerpunkte zur Bearbeitung innerhalb einzelner Programmlinien herausstellen. [H. 16] [BMW20]

Die Programmlinie D unter der Bezeichnung *Industrie 4.0 / MRO* des dritten Aufrufs im Rahmen von LuFo VI widmet sich hierbei insbesondere der Weiterentwicklung traditioneller Produktionssysteme. Durch eine vertikale Integration der Teilsysteme über die gesamte Wertschöpfungskette hinweg, in Kombination mit horizontaler Integration über die Wertschöpfungspartner, sollen so intelligente und adaptive Fertigungsnetzwerke entstehen ([H. 16]). Neben der Produktion sollen diese modularen Systeme auch Anwendung in den Bereichen Betrieb, Wartung, Reparatur und Überholung (MRO) finden, womit schließlich der gesamte Produktlebenszyklus abgedeckt werden kann. Zur Programmlinie D zugehörig ist u.a. der Fachbereich 4 “Innovative Strukturen für Luftfahrzeuge“, unter welchem auch das Verbundprojekt MFlex2025 gefördert wird. Mit dem renommierten Flugzeughersteller Airbus als Verbundführer und fünf weiteren Verbundpartnern zu denen auch das Fraunhofer IFAM zählt, soll das Projekt MFlex2025

eine flexible und ressourceneffiziente Flugzeugproduktion mithilfe mobiler Robotereinheiten realisieren. Um solche mobilen Robotersysteme wirtschaftlich effizient in die variantenreiche Produktion von Luftfahrzeugen integrieren zu können, zielt MFlex2025 insbesondere auf die Entwicklung möglichst wandlungsfähiger und flexibler, automatisierter Systeme ab. Durch den Fokus des Projektes auf Interoperabilität und Flexibilität, soll später im realen Einsatz sichergestellt werden, dass das System zügig an neue Aufgabenstellungen und Rahmenbedingungen adaptierbar ist. Während Wartezeiten oder Produktionsunterbrechungen bestimmter Stationen einer Produktionslinie, könnte ein mobiles Fertigungssystem so bspw. seinen Standort ändern und an einer anderen Station mit einer anderen Tätigkeit weiterhin effizient genutzt werden. [H. 16] [BMW20]

Einen Kernfaktor zur Entwicklung solch flexibler Systeme stellt die Austauschbarkeit von Komponenten oder Subsystemen dar. Bezogen auf das vorherige Beispiel könnte dies bedeuten, dass das mobile Fertigungssystem beim Stationswechsel z.B. auch einen Werkzeugwechsel vornimmt. Im Fall von Störungen des Systems könnten dahingehend auch einzelne Komponenten problemlos zeit- und kosteneffizient getauscht werden, ohne sie erneut aufwendig in das Gesamtsystem integrieren zu müssen. Diese Austauschbarkeit ist jedoch direkt geknüpft an ein hohes Maß an Modularität innerhalb des Systems. Das Vorhaben im Rahmen von MFlex2025 richtet sich daher insbesondere an eine durchgängig modulare Implementierung von hard- und softwarebasierten Systemen, welche gemeinsam im Verbund entwickelt und verbessert werden. Der Verbundführer Airbus definiert hierbei neben den verwendeten Modulen, auch die zu implementierenden Standards und Schnittstellen, wodurch eine nahtlose Integration der Ergebnisse in die zukünftige, digitale Fertigung sichergestellt werden soll.

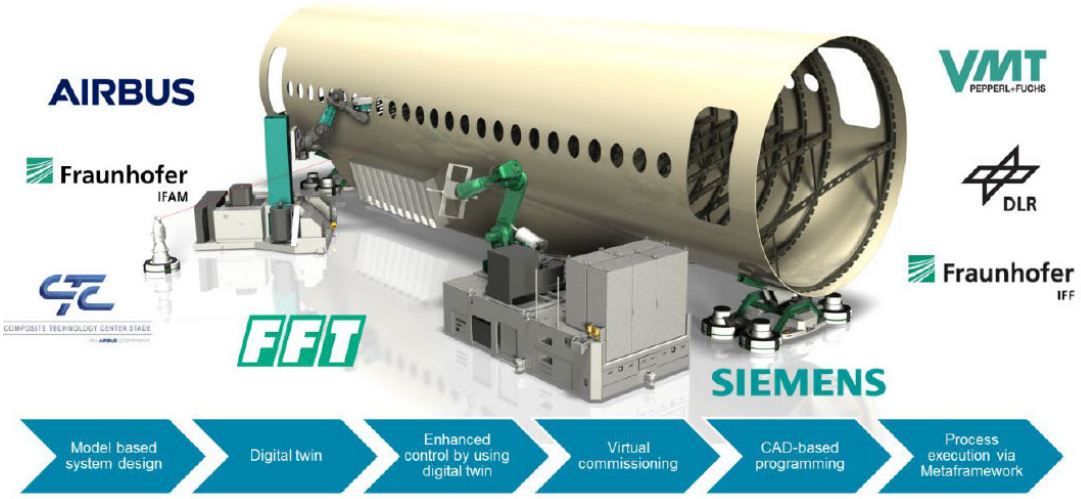


Abbildung 3.1: Konzeptzeichnung des Verbundvorhabens MFlex2025 (Quelle: IFAM intern)

Die Arbeitspakete im Rahmen von MFlex2025 umfassen somit neben der Weiterentwicklung modularer, mobiler und roboterbasierter Produktionssysteme, auch entsprechende IT-Lösungen zur flexiblen Einbindung der einzelnen Module in das Anlagensystem. Zur Begrenzung des Aufwandes innerhalb des Projektes werden zwei mobile Robotersysteme aus vorangegangenen Förderprojekten als Basis genutzt. Wie in Abbildung 3.1 skizziert, handelt es sich hierbei um mobile Systeme, welche jeweils aus den Mo-

dulen Bewegungsplattform, Roboter, Endeffektor und Messtechnik bestehen. Der Anwendungsbereich der beiden Systeme beläuft sich hierbei auf das Bohren einer Längsnaht, sowie das Anbringen von Dekoration. Letzteres System wird im Rahmen von MFlex2025 allerdings nicht in Form eines physischen Demonstrators in Betrieb genommen.

3.2 Komponenten und Modulares Leitsystem

Zur Umsetzung einer möglichst flexiblen Anlagenarchitektur werden die einzelnen Komponenten der mobilen Robotersysteme, welche im Rahmen von MFlex2025 zum Einsatz kommen, zunächst auf Basis ihrer Anwendung in Modulen kategorisiert (siehe Abb. 3.2). Durch Kombination von Modulen entstehen anschließend variable Analgensysteme, welche je nach Bedarf eingesetzt werden können. Dies erlaubt den Projektpartnern im späteren Kontext zu zeigen, wie die Wandlungsfähigkeit solcher Gesamtsysteme durch Modularisierung gesteigert werden kann. Ziel des Projektes ist demnach u.a. die einzelnen Komponenten als Standardmodule für unterschiedlichste Einsatzzwecke nutzbar zu machen.

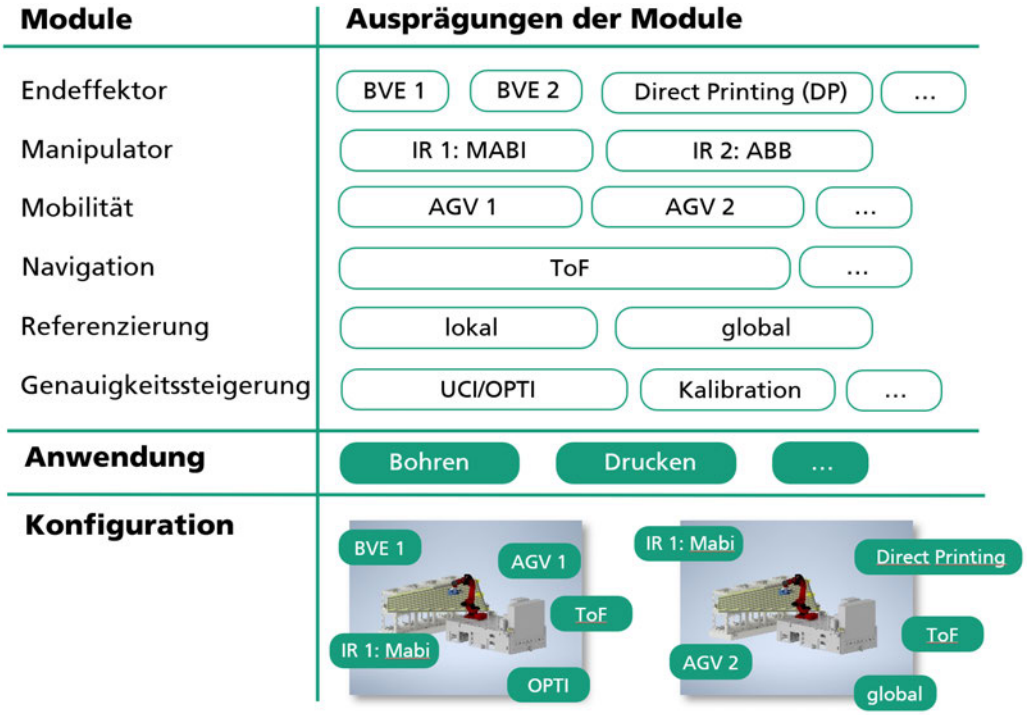


Abbildung 3.2: Modulares Konzept-MFlex2025 (Quelle: IFAM intern)

Wie in Abbildung 3.2 dargestellt und im vorherigen Kapitel bereits kurz beschrieben, setzt sich ein mobiles Robotersystem im Rahmen von MFlex2025 grundlegend aus den Modulen Endeffektor, Manipulator, Mobilität, Navigation, Referenzierung und Genauigkeitssteigerung zusammen. Jedes Modul stellt hierbei im Rahmen der Anwendungsdomäne der Anlage wichtige autonome Fertigkeiten über die Softwareschnittstelle zur Verfügung. Je nach Tätigkeit, werden anschließend die benötigten Module ausgewählt und die Ausprägungen einzelner Module in Form spezifischer Komponenten bestimmt. So wird

bspw. für einen autonomen Bohrprozess das Modul Manipulator mit einem Industrieroboter (IR) und das Modul Endeffektor mit einer Bohrvorschubeinheit (BVE) belegt. Sollte der Arbeitsraum des IR für die gewünschte Tätigkeit bspw. nicht ausreichen oder eine Bewegung zwischen Arbeitsplätzen nötig sein, kann die Anlage durch das Modul Mobilität mithilfe einer mobilen Plattform als Modulausprägung (engl. *Automated Guided Vehivle* (AGV)) erweitert werden. Das AGV kann anschließend bspw. mithilfe von Lidar- oder Time-of-flight-Systemen (ToF) im Modul Navigation sicher verfahren. Zur Umsetzung präziser Bohrungen am Bauteil kann zusätzlich das Modul Genuigkeitssteigerung mit einem Sensorsystem zur optischen Einmessung als Ausprägung genutzt werden. Zur beliebigen Integration einzelner Komponenten in das Anlagensystem, werden zunächst vereinheitlichte Schnittstellen zur Kommunikation definiert. Diese Schnittstellendefinition geschieht hierbei im Communications-Layer des jeweiligen Assets (siehe RAMI-Modell 2.1) und wird im Rahmen von MFlex2025 ebenfalls mittels OPC UA als führendem Kommunikationsstandart der Industrie 4.0 umgesetzt werden. Auf Basis dessen besitzt jede Modulausprägung einen OPC UA Server, welcher wichtige Funktionalitäten und Informationen zur Überwachung und Koordinierung des Feldgerätes bereitstellt. Dieser Server kann hierbei direkt auf dem Feldgerät bzw. dessen Steuerungskomponente oder in Form einer vorgeschalteten Softwarelösung (Edge-Device) ausgeführt werden. Auf die Lösung mittels Edge-Device wird hierbei zumeist nur dann zurück gegriffen, wenn sich gewünschte Funktionalitäten nicht direkt auf dem Feldgerät realisieren lassen.

Durch OPC UA können anschließend die generischen Strukturen von überliegenden System aufgegriffen und mittels Client-Server-Verbindung genutzt werden. Ein solches, übergeordnetes System zur Koordination und Überwachung der Feldgeräte kommt auch innerhalb von MFlex2025 zum Einsatz. Um die einzelnen Module im späteren Kontext einer variablen Anlagenarchitektur zentral integrieren zu können, wird daher im Zuge des Projektes ein anwender- und bedienerfreundliches modulares Leitsystem in Form eines übergeordneten Softwareframework entwickelt. Das Leitsystem fungiert hierbei jedoch vorrangig als Metaframework der Anlage, da es bereits bestehende oder anderweitig entwickelte Teilsysteme verbindet und koordiniert, aber nicht in ihrer Funktion ersetzt. Das Metaframework greift die generischen Schnittstellen der einzelnen Komponenten auf und ermöglicht so bspw. die Integration von Autonomiefunktionalitäten in Form von IT-Diensten. Im Detail bedeutet dies, dass die Entwicklung des autonomen Verhaltens nicht Anlagen- oder System-spezifisch programmiert wird, sondern übergeordnet bereitgestellt werden kann. Diese Vereinheitlichung ermöglicht zudem, der Anlage mit wesentlich geringerem Aufwand neue Aufgaben zuordnen zu können.

Um demonstrativ zeigen zu können, inwiefern die modulare Bauweise auch real zum Einsatz kommen kann, wird im Rahmen des Projektes ein physischer Demonstrator in der Forschungshalle des Fraunhofer IFAM in Stade aufgestellt. Bei diesem Demonstrator handelt es sich um das, in Kapitel 3.1 bereits angesprochene, mobile Anlagensystem zum autonomen Bohren an Flugzeugkomponenten. Die Anlage setzt sich hierbei, wie im Modularen Konzept in Darstellung 3.2 zu sehen, aus den Modulen Endeffektor, Manipulator, Mobilität, Navigation, Referenzierung und Genauigkeitssteigerung zusammen und soll zur Validierung der Projektziele auch innerhalb koordinierter Versuche an echten Faserverbundbauteilen Bohrungen vollziehen. Um passgenaue Bohrungen auf der Bauteiloberfläche durchführen zu können, wird das Modul Endeffektor hierzu mit einer Bohrvorschubeinheit (BVE 1) als Modulausprägung besetzt. Die BVE, welche in Abbildung 3.3 in Form eines Rendering dargestellt ist, wurde bereits vor

Projektbeginn im Rahmen einer Masterarbeit am Fraunhofer IFAM entwickelt und beinhaltet eine Linearachse, auf welcher eine Hochfrequenzspindel des Typen SLQ100 zum Bohren montiert ist. Beim Bohrprozess übernimmt die BVE somit den gesamten Fertigungsaufwand und der Manipulator dient lediglich zur Positionierung der Einheit, wodurch eine größere Genauigkeit der Bohrungen möglich wird. Weiterführend ist die BVE mit einem Pneumatiksystem versehen, welches dem Manipulator erlaubt, sich beim Bohrprozess gegen das Bauteil zu verspannen, wodurch das Gesamtsystem für diesen Zeitabschnitt enorm an Steifigkeit gewinnt und störenden Schwingungen den Bohrvorgang geringfügiger beeinflussen.

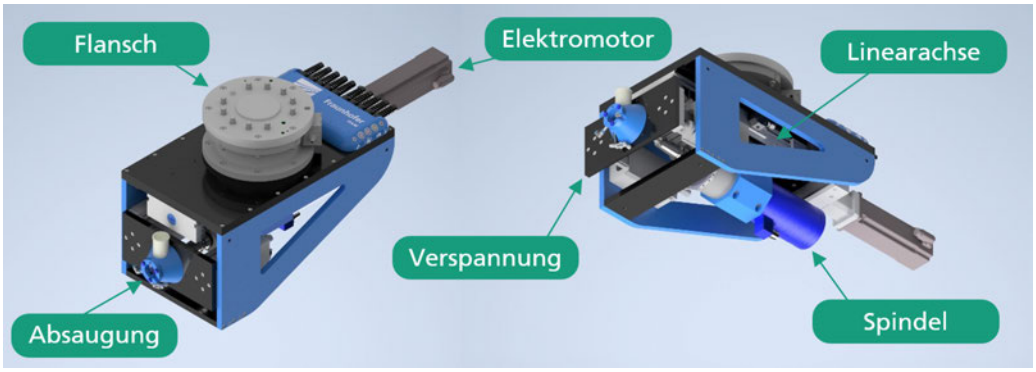


Abbildung 3.3: Rendering der Bohrvorschubeinheit (BVE)

Wie bereits beschrieben wird das Modul Mobilität im Rahmen von MFlex2025 mit einer mobilen Plattform besetzt. Diese verschafft der Anlage den Vorteil bei der Bearbeitung größerer Bauteile, nicht auf den Arbeitsraum des Roboters begrenzt zu werden und gegebenenfalls den Bearbeitungsstandort innerhalb der Produktion wechseln zu können. Das AGV wurde hierbei im Rahmen eines früheren Projektes (Projektname: "MBFAST18") in Zusammenarbeit mit der Firma FFT entwickelt und ist in der Lage präzise Verfahrbewegungen innerhalb eines definierten Bereiches durchzuführen. Wie in Darstellung 3.4 zu sehen wird der Industrieroboter inkl. Steuerung zur Nutzung von MBFAST18 direkt auf der Plattform montiert. Zur Referenzierung nutzt die Anlage hierbei ein Navigationsmodul der Firma ESPACE2001. Dieses orientiert sich mithilfe von Lidar-Technik an festen Reflektoren in der Halle und Referenziert sich anhand eines festen Koordinatensystems.

Zur Genauigkeitssteigerung des Demonstrators wurde die am Endeffektor montierte BVE mit einem Präzisionsmodul der Firma 3D.aero besetzt. Das Präzisionsmodul beinhaltet hierbei ein Stereokamerasystem zur Aufnahme von Bilddaten und kann anhand von spezifischen Merkmalen am Werkstück eine Referenzierung des Manipulators bzw. der BVE zum Bauteil vornehmen. Hierzu fährt das AGV zunächst auf eine definierte Position vor dem Bauteil, sodass der Manipulator eine Reihe von Messpositionen sicher anfahren kann. An diesen Messpositionen kann das Präzisionsmodul anschließend Messungen in Form von Aufnahmen vornehmen. Abschließend wird anhand dieser Messungen und mithilfe eines Algorithmus zur Objekterkennung eine Transformation abgeleitet, durch welche das Leitsystem im späteren Verlauf korrigierte und somit besonders präzise Bohrungen am Bauteil koordinieren kann.

Abbildung 3.4: Mobiler Demonstrator für den Bohrprozess (Quelle: IFAM intern)

Abbildung 3.4 zeigt an dieser Stelle noch einmal den physischen Demonstrator, so wie er im Rahmen von MFlex2025 zur Durchführung von Versuchen und Tests einzelner Komponenten genutzt wurde. Des weiteren ist in Anhang A.1 ein Rendering der Gesamtanlage inkl. Bauteil dargestellt. Wie dort zu sehen imitiert das verwendete Bauteil das Seitenleitwerk eines Flugzeuges und ist montiert auf einem festen Spannfeld. Das Bauteil besteht hierbei aus einem Kohlefaserverstärktem Kunststoff (CFK). Der in Abbildung 3.4 dargestellte Industrieroboter dient im Rahmen von MFlex2025 als Ausprägung des Modules Manipulator und wird in der realen Anlage mit einem MabiMax100 der schweizer Firma MABI besetzt. Dieser Sechsachsroboter weißt eine Traglast von 100kg am Rand des Arbeitsraumes, wodurch die Steuerung (Siemens Sinumerik840dsl) die BVE später genau positionieren kann. Außerdem kommt ein Lasertracker der Marke Leica zur Validierung der Versuche zum Einsatz, welcher jedoch nicht in Abbildung 3.4 zu sehen ist.

3.3 Use Case OPC UA Aggregation

Wie bereits angesprochen wird im Rahmen von MFlex2025 ein Leitsystem zur Koordination und Überwachung des Bohrvorgangs an definierten Stellen des Bauteils genutzt (siehe Abb. 3.2). Das Leitsystem muss hierfür einige essentielle Funktionalitäten aufweisen, zu denen unter anderem die Ablaufplanung, sowie die Berechnung der notwendigen Koordinaten zum Verfahren des AGV und des Roboters gehören. Die Ablaufplanung umschließt aufgrund dessen neben dem Bohrvorgang an sich auch die Referenzierung des AGV und des Roboters gegenüber dem Bauteil, wobei hier auch die Messungen des Präzisions-

modules berücksichtigt werden müssen. Zusammengefasst übernimmt das Leitsystem die nachfolgend aufgelisteten Aufgaben:

- Bohrpositionen aufnehmen
- AGV Zielposition bestimmen und Verfahrbewegung an Navigationsmodul kommunizieren
- Präzisionsmodul starten
- Messpositionen berechnen und Roboterposen kommunizieren
- Messungen starten
- Ergebnisse der Messungen abfragen und Bohrpositionen durch Lösung der Transformationen bestimmen
- Roboter-Posen bestimmen und kommunizieren
- BVE und Spindel koordinieren

Zur Ablaufplanung werden diese Prozesse im Anschluss strukturiert aneinandergereiht, sodass der gewünschte Bohrvorgang so präzise wie möglich umgesetzt werden kann. Wie bereits der Auflistung zu entnehmen ist, bedingt hierbei nahezu jeder Prozess einer stabilen Kommunikation der einzelnen Module mit dem Leitsystem. Da im Rahmen von MFlex2025 die Kommunikation ausschließlich über OPC UA realisiert werden soll, muss das Leitsystem demnach ebenfalls eine generische OPC UA Schnittstelle aufweisen. Diese soll so Client-Zugriffe auf die OPC UA-Schnittstellen der Feldgeräte ermöglichen, um Informationen auszutauschen oder Aktionen, unter Verwendung der Services des OPC UA-Protokolls, zu koordinieren.

Wie in der Einleitung bereits beschrieben und in Abbildung 1.3 dargestellt, sorgt auch beim echten Demonstrator die Kommunikation des Leitsystems mit jedem einzelnen Feldgerät für eine Vielzahl an Verbindungen innerhalb des Netzwerkes. Sollte die Anlage im Anschluss an das Projekt bspw. zusätzlich noch mit einem SCADA oder MES System verknüpft werden, kann es so schnell zu einem Vielfachen der aktuell benötigten Verbindungen und einem damit einhergehenden hohen Konfigurationsaufwand kommen, wodurch die Zuverlässigkeit der Anlage sinken und das Netzwerk stark ausgelastet werden kann. Um dieser Problematik vorbeugend entgegen zu wirken, kann hier die vollautomatische Aggregation der einzelnen OPC UA-Schnittstellen genutzt werden. Durch Zusammenführung relevanter Knotenstrukturen kann so die Anzahl der Client-Verbindungen auf ein Minimum reduziert werden, wodurch die Wahrscheinlichkeit des Ausfalls von Verbindungen sinkt. Zudem ergibt sich durch die Aggregation der Vorteil, einer zentral gesteuerten Sicherheit für Anwenderzugriffe. Dies bedeutet im Detail, dass Clients ausschließlich über den Aggregationsserver auf die Anlage zugreifen können, wodurch die Möglichkeit entsteht die Zugriffe durch zentral definierte Sicherheitsfunktionen auf dem Server zu koordinieren und auf mögliche Verstöße zu reagieren. Des weiteren bietet eine zentrale Integrationsplattform die Perspektive einer Erweiterung der Kommunikationsfähigkeit der Anlage, durch Bereitstellung von Schnittstellen zur Kommunikation mit anderen Softwarelösungen des IoT.

4 Softwareentwicklung

Dieses Kapitel befasst sich mit der Entwicklung des aggregierenden Servers. Hierzu werden zunächst die Bedürfnisse des Endnutzers identifiziert und Anforderungen an die Software definiert, aus denen später spezifische Funktionalitäten des Servers abgeleitet werden können. Des weiteren wird die Nutzung und Konfiguration des Softwarepaketes thematisiert, um im Anschluss ein Konzept zur Aggregation generischer OPC UA Server aufzustellen. Der schematische Aufbau der Softwareimplementierung kann abschließend mithilfe von Klassen und Methoden, abgeleitet aus der Anforderungsanalyse, definiert und die prozessstrukturelle Integration Schritt für Schritt erläutert werden.

4.1 Anforderungsanalyse

Zur erfolgreichen Entwicklung eines neuen Softwareproduktes ist es essenziell die Anforderungen und Bedürfnisse des Nutzers an die Implementierung zu kennen und zu analysieren. Neben den Randbedingungen und grundlegenden funktionalen Anforderungen, meist geprägt durch die Use-Cases, spielen so auch subjektive Faktoren wie bspw. die Benutzerfreundlichkeit oder das Design eine wichtige Rolle. Im Rahmen der Systementwicklung gilt daher die Anforderungsanalyse (engl. *Requirements Analysis*) als zentrales Tool zur Ermittlung von Forderungen, Bedürfnissen und Wünschen des Auftraggebers (engl. *Stakeholder*) an das zu entwickelnde System. Hierbei kommen in der Informatik eine Vielzahl an unterschiedlichen Verfahren zur Analyse der Anforderungen zum Tragen, wobei die meisten Methodiken mit dem Ziel eines Lastenheftes durchgeführt werden. Ein Lastenheft, oder im Rahmen der agilen Softwareentwicklung auch *Product-Backlog* genannt, dokumentiert die analysierten Anforderungen an das Produkt zur Orientierung im späteren Entwicklungsprozess. Das Lastenheft formuliert die Anforderungen hierbei so allgemein wie möglich und reduziert Einschränkungen auf das Nötigste, um beim Lösungsprozess nicht durch zu konkrete Anforderungen eingeschränkt zu werden.

Um Anforderungen analysieren zu können und anschließend ein Product-Backlog daraus abzuleiten, müssen diese zunächst aufgenommen und dokumentiert werden. Hierbei sollte möglichst von Beginn an darauf geachtet werden, die Anforderungen der Stakeholder in funktionale und nicht-funktionale Anforderungen zu unterteilen. Als Funktional werden Anforderungen hierbei nur dann bezeichnet, wenn deren Umsetzung direkt der Zweckbestimmung des Produktes dienen. Als Beispiel könnte hier im Fall der Aggregation generischer OPC UA Server die Bereitstellung einer Server-Instanz für den Client-Zugriff dienen. Denn ohne den Server an sich bereitzustellen, fehlt die grundlegende Plattform zur Aggregation von Knotenstrukturen und Clients wird der Zugriff auf die benötigten Daten verwehrt. In diesem Fall wird der Zweck des Produktes nicht erfüllt. Nicht-Funktionale Anforderungen hingegen gelten eher als

nebensächlich zur Zweckbestimmung und beschreiben in der Softwareentwicklung zumeist die Qualität in welcher das System Leistungen erbringen soll. Anforderungen an die Performance der Software zum Beispiel dienen nicht der Aggregation generischer Knotenstrukturen, sondern verbessern in erster Linie lediglich die Rechendauer der Operation und optimieren so den Prozess. Sie werden aufgrund dessen als nicht-funktionell eingestuft.

Neben der Aufteilung nach funktionalen und nicht-funktionalen Anforderungen sollte bei der Anforderungsanalyse stets systematisch vorgegangen werden. So bietet sich im Bereich der Softwareentwicklung bspw. das Volere-Schema (Volere-Template), entwickelt im Jahr 1995 von James und Suzanne Robertson, als geeignete Grundlage zur systematischen Definition und Analyse von Anforderungen an[Jam07]. Volere ist hierbei das Resultat aus langjähriger Praxiserfahrung der Autoren, insbesondere in den Bereichen Anforderungsmanagement (engl. *Requirement Enghineering*) und Geschäftsanalyse. Das Schema beinhaltet hierbei eine Ansammlung von Techniken und Vorlagen zur Definition, Klassifizierung und Analyse von Anforderungen aus dem Software-Engineering Bereich. Volere unterteilt die Anforderungen hierbei in 27 Anforderungstypen, die fünf Kategorien zugeordnet werden. Der in Abbildung 4.1 dargestellte Ausschnitt aus dem Inhaltsverzeichnis des Volere-Template zeigt beispielhaft zwei Kategorie und ihre Anforderungstypen. Wie abzulesen handelt es sich hier erneut um die Einordnung funktionaler und nicht-funktionaler Anforderungen, jedoch mit dem Unterschied, dass die Anforderungen im Anschluss erneut in 3-8 Anforderungstypen klar unterteilt werden. Dies hat den direkten Vorteil, dass Anforderungen genauer eingestuft, analysiert und bewertet werden können. [Jam07]

Abbildung 4.1: Funktionale und Nicht-Funktionale Anforderungen nach Volere ([Jam07], S. 2)

Um eine möglichst vereinheitlichte und dennoch detailreiche Definition von Anforderungen an das Produkt zu realisieren, beinhaltet Vorlere eine Vorlage, die sogenannte *Requirements-Shell*. Die Requirements-Shell soll hierbei als standardisierte Definitionsumgebung verstanden werden, welche der Entwickler nutzen kann, um Anforderungen zu konkretisieren, zu begründen und einzustufen. Zudem soll das ausgefüllte Dokument Informationen über Konflikte mit anderen Anforderungen beinhalten und ein Abnahmekriterium definieren, welches später im Kapitel vertieft thematisiert wird. Wie bereits angedeutet ist

das Dokument hierbei stets für jede Anforderung separat auszufüllen. Im Anschluss an die Requirements-Shell, können die aufgenommenen Daten in einer Liste zusammengefasst, bewertet und separat analysiert oder verglichen werden. Bezogen auf die Entwicklung eines OPC UA Aggregationsservers, im Rahmen der Bachelor-Thesis, zeigt Darstellung 4.2, beispielhaft die Definition eines zeitlichen Rahmens zur Ausarbeitung in Anlehnung an die Requirements-Shell des Volere-Template.

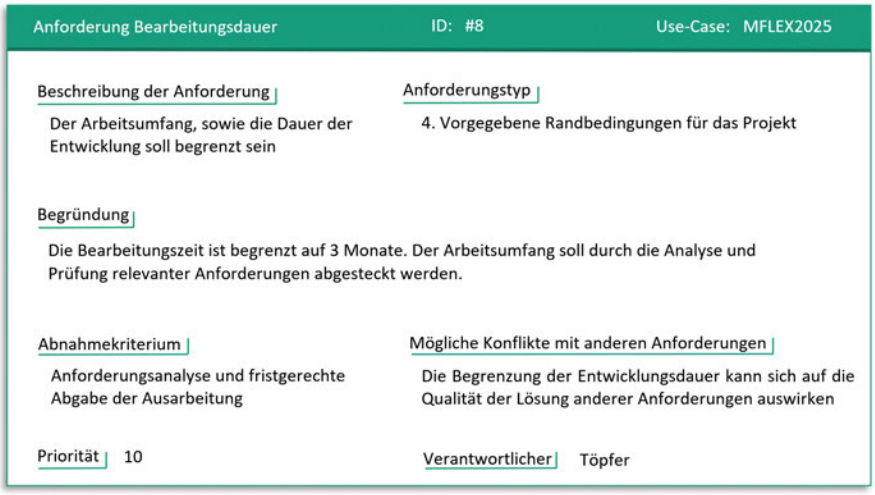


Abbildung 4.2: Definition von Anforderungen nach Robertson (In Anlehnung an [Jam07], S. 6)

Wie in Abbildung 4.2 zu sehen, fordert die Vorlage zunächst eine genaue Bezeichnung der Anforderung, sowie eine eindeutige ID und die Zuordnung zu einem Use Case, welches in diesem Fall das Projekt MFlex2025 ist. Weiterführend folgt eine Kurzbeschreibung und eine Begründung, warum genau die Anforderung im Anwendungsfall notwendig ist. Auf der rechten Seite befindet sich zusätzlich das Feld zur Definition des Anforderungstyps. In diesem Fall wird die Anforderung im Bezug auf die Bearbeitungsdauer dem Anforderungstypen “4.Vorgegebene Randbedingungen für das Projekt“ zugeordnet, da die Bearbeitungszeit durch die Vorgaben der Hochschule begrenzt ist. Des weiteren befinden sich im unteren Bereich der Vorlage Informationen zum Abnahmekriterium der Anforderung. Das Volere-Template folgt hierbei der Einstellung, das jede Anforderung bereits bei der Definition an ein Abnahmekriterium geknüpft sein muss, um sie messbar zu machen. Das Abnahme- bzw. Eignungskriteriums vereint hierbei gleich mehrere Vorteile. Einerseits können auf Basis dessen, Lösungsansätze direkt von Beginn an darauf untersucht werden, ob sie die Anforderung ausreichend erfüllen können und andererseits kann mithilfe des Abnahmekriteriums die Definition der Anforderung geprüft werden. Sollte für eine Anforderung hierbei kein Abnahmekriterium gefunden werden können, so ist die Anforderung nach Volere entweder mehrdeutig oder nicht detailliert genug formuliert und sollte korrigiert werden.

Auf Basis der Reuqirements-Shell können anschließend weitere Anforderungen an die Softwarelösung zur Aggregation generischer OPC UA Server mit Bezug auf den Use Case MFlex2025 definiert und in einer Tabelle aufgelistet werden. Als Beispiel für funktionale Anforderungen ist in Abbildung 4.1 ein Ausschnitt dieser, in Tabelle A.1 ganzheitlich aufgeführten, Auflistung aller Anforderungen an die Software dargestellt. Alle drei Anforderungen sind vom selben Anforderungstyp und fassen die Bedingungen

an die Funktionen und Daten der Software zusammen. Die erste Anforderung befasst sich hierbei mit dem OPC UA Server an sich, welcher durch die Implementierung bereitgestellt werden soll. Begründet wird die Anforderung hierbei dadurch, dass ohne Server keine Informationen für einen Client-Zugriff zur Verfügung gestellt werden können, wodurch die Zielfunktion der Aggregation generischer Knotenstrukturen nicht umsetzbar wird. Das Abnahmekriterium besteht in diesem Fall aus einem Test der Client-Server-Verbindung inkl. dem Austausch von Informationen. Sollte die Verbindung problemlos nachweisbar sein, gilt die Anforderung als erfüllt.

Tabelle 4.1: Definition von Funktionalen Anforderungen an die Software

Id	Anforderungstyp	Beschreibung	Begründung	Abnahmekriterium	Konflikte/Machbarkeit	Priorität (1-10)
1	9. Anforderungen an Funktionen und Daten des Produktes	Die Software soll einen OPC UA Server für Client-Zugriffe bereitstellen	Um aggregierte Informationen bereitstellen zu können, soll die Software einen OPC UA-Server beinhalten, welcher einen Client-Zugriff ermöglicht	Test der Erreichbarkeit innerhalb des Netzwerkes durch einen Client-Zugriff	--	10
2	9. Anforderungen an Funktionen und Daten des Produktes	Der Server soll Knotenstrukturen aggregierter OPC UA-Server abbilden	Zur Realisierung einer OPC UA Server Aggregation müss der Aggregationsserver Knotenstrukturen der Quellserver adaptieren und Abbilden können	Die Konfigurierten Knotenstrukturen sind auf dem Server durch einen Browse-Request verfügbar	--	9
3	9. Anforderungen an Funktionen und Daten des Produktes	Die Software soll die Abfrage von Daten aggregierter OPC UA-Server ermöglichen	Um aggregierte Knotenstrukturen funktional nutzbar zu machen, sollen Attribute und Daten aggregierter Knoten auf dem Aggregationsserver verfügbar gemacht werden	Abfrage der Daten mittels DataAccess-Request	--	8

Der Tabelle 4.1 ist außerdem zu entnehmen, dass die zweite Anforderung an das Produkt die Zielfunktion selbst definiert. Hierbei handelt es sich um die Aggregation generischer Knotenstrukturen unterschiedlicher Quellserver. Erweitert wird diese Funktion zudem durch Anforderung drei, welche den Zugriff auf Daten bzw. Attributen aggregierter Knoten definiert. Beide Anforderungen sind hierbei von essentieller Bedeutung, da sie zu den Hauptfunktionalitäten der Software zählen. Die Priorität der Bedingungen wird dementsprechend hoch angesetzt. Zur Validierung möglicher Lösungsansätze um die Bedingungen erfüllen zu können, sollen Tests in Form von client-seitigen Service-Anfragen (*Browse-Request*, *Subscription-Request*) an den aggregierenden Server durchgeführt werden. Wie bei der ersten Anforderung gilt auch hier, sollten die Anfragen der Clients bzw. die Antwort des Server problemlos nachweisbar und überprüfbar sein, ist die Anforderung als erfüllt zu betrachten.

Ergänzend zu den funktionalen Anforderungen, werden in Abbildung 4.2 drei nicht-funktionale Anforderungen vorgestellt. Der Definition nach, handelt es sich hierbei um Anforderungen, welche die Qualität der Anwendung bezogen auf den Use Case steigern sollen. Sie dienen dementsprechend nicht direkt der Zweckbestimmung des Produktes. SO definiert Anforderung 10 bspw. die Dokumentation der Anwendung, sowohl in schriftlicher Form, als auch in Form von Kommentaren direkt im Code. Begründet durch den Rahmen der Bachelor-Thesis ist eine schriftliche Ausarbeitung notwendig. Kommentare im Code können hierbei zudem vertiefend zum Verständnis der Vorgehensweise beitragen und helfen später die Software mit weiteren Funktionalitäten auszubauen oder Fehler schneller lokalisieren zu können. Weitere Qualitätsmerkmale des Produktes werden durch die Anforderungen 11 und 12 sichergestellt, welche sich auf die Performance und Sicherheit der Implementierung beziehen. So sollte bei der Aggregation generischer OPC UA Server darauf geachtet werden, dass die dort vorherrschenden Sicherheitskonzepte vom Aufbau einer verschlüsselten Verbindung bis hin zum Zugriff auf einzelne Knoten im Adressraum

übernommen werden, oder eine entsprechend höherwertige Sicherheitsoption auf der Integrationsplattform umgesetzt wird. Validiert wird dies, über entsprechend verschlüsselte Client-Zugriffe auf Knotenstrukturen des aggregierenden Server. Darüber hinaus wird in Anforderung 12 eine performante und vollautomatische Aggregation definiert. Im Abnahmekriterium wird hierbei zudem beschrieben, dass die automatische Aggregation durch ein Signal, etwa einem Befehl des Client, gestartet werden darf. Zur Beurteilung der Performance soll die Zeit gemessen werden, welche pro aggregiertem Knoten anfällt. Da ein Adressraum jedoch eine variable Anzahl von Knoten besitzen kann, wird hier auf das Use Case MFlex2025 verwiesen, wo mit einem überschlägigen Mittelwert von ca. 200-600 relevanten Knoten zur Aggregation gerechnet werden kann. Da die Aggregation lediglich einmal pro Knotenstruktur und Quellserver ausgeführt werden muss, wird für die Gesamtdauer ein akzeptabler Zeitrahmen von 2 min angenommen. Pro aggregierten Knoten folgt somit zur Erfüllung der Maximalanforderung von 600 relevanten Knoten eine durchschnittliche Aggregationsdauer von ungefähr 0,2 s.

Tabelle 4.2: Definition von Nicht-Funktionalen Anforderungen an die Software

Id	Anforderungstyp	Beschreibung	Begründung	Abnahmekriterium	Konflikte/Machbarkeit	Priorität (1-10)
10	11. Benutzbarkeitsanforderungen	Die Softwareimplementierung soll kommentiert und dokumentiert werden	Durch Kommentare und Beschreibungen im Code und der Ausarbeitung, können einzelne Prozesse und Strukturen künftig einfacher interpretiert und identifiziert werden	Beschreibung von Klassen und Methoden im Code, sowie die schriftliche Ausarbeitung	--	7
11	12. Sicherheitsanforderungen	Der Server soll die Sicherheitsstandarts aggregierter Server übernehmen	Zum sicheren Informationsaustausch soll der Server die von den aggregierten Servern verwendeten Verschlüsslungsmethoden und Nutzerbeschränkungen adaptieren	Client-Zugriff über entsprechend verschlüsselte Verbindung	--	2
12	12. Performance Anforderungen	Die Aggregation soll performant und automatisch durchgeführt werden	Um eine effiziente Verwendung der Software gewährleisten zu können, muss die Aggregation vollautomatisch durchgeführt werden und soll hierbei möglichst wenig Zeit in Anspruch nehmen	Automatische Aggregation der relevanten Informationen (Kann durch Auslöser angestoßen werden). Die Dauer des Aggregationsprozesses (ca. 600 Knoeten inkl. Attribute) pro Knoten soll hierbei 0,2 s nicht überschreiten.	Performance-Verbesserungen oft sehr zeitintensiv. Machbarkeit daher beschränkt durch Bearbeitungsdauer	3

Das Product-Backlog (Tabelle A.1) besitzt neben den Spalten für die Beschreibung, die Begründung und das Abnahmekriterium zwei weitere Spalten. Diese definieren mögliche Konflikte mit anderen Anforderungen und stufen die Anforderung in ihrer Wichtigkeit bezogen auf das Use-Case ein. Wie aus Tabelle A.1 des Anhangs abzulesen, sorgt nahezu keine der definierten Anforderungen an die Software-Implementierung für einen Konflikt mit anderen Anforderungen. Lediglich das Optimierungspotenzial im Hinblick auf die Prozessdauer könnte durch den zeitlichen Entwicklungsrahmen eingeschränkt werden. Da es im Verbundprojekt MFlex2025 jedoch eher um die Machbarkeit eines Lösungsansatzes und weniger um eine besonders performante Lösungen geht, wurde die Anforderung der Performance mit einer vergleichsweise niedrigeren Priorität eingestuft. Dies bestätigt sich zum Beispiel durch den Vergleich mit unveränderlichen Randbedingungen bzw. essentiellen Funktionen des Produktes, welche in Form von Anforderungen mit der Priorität 10 definiert worden sind. Hierzu zählen unter anderem die festgelegte Bearbeitungsdauer und die Aggregation generischer Knotenstrukturen selbst (Tabelle 4.2).

4.2 Verwendete Toolchain

Zur Umsetzung der Anforderungen des Product-Backlog bei der Implementierung der Softwarelösung wird zunächst eine sogenannte *Toolchain* zur Programmierung gewählt. Als Toolchain werden in der Softwareentwicklung alle verwendeten Software-Werkzeuge zur Erstellung der Software bezeichnet. Die Toolchain besteht in diesem Fall aus einer integrierten Entwicklungsumgebung, der verwendeten Programmiersprache und der einer Bibliothek zur Nutzung bereits vorhandener Routinen.

Eine besonders große Bedeutung kommt hierbei der Wahl der Programmiersprache zuteil. Sie wird zumeist durch den vorhandenen Anwendungsfall bzw. durch die Anforderungen an die Software bestimmt. Programmiersprachen werden hierbei grundlegend unterteilt in einfache und höhere Sprachen. Höhere Sprachen zeichnen sich insbesondere durch ein gehobeneres Maß an Komplexität aus und können so bspw. von Mikroprozessoren nicht unmittelbar verarbeitet werden. Befehlsfolgen, definiert in Hochsprachen, müssen daher zuvor mithilfe eines *Compiler* übersetzt werden, bevor sie von einfacheren System interpretiert werden können. Hochsprachen werden weiterführend unterteilt in strukturierte und objektorientierte Programmiersprachen. Strukturierte Programme folgen hierbei dem Schema der prozeduralen Programmierung, was bedeutet, dass das Programm in Teilprogramme zerlegt wird, welche wiederum einzelne Prozessabläufe mittels aufeinander folgender Befehle ausführen. Im Gegensatz hierzu richten sich objektorientierte Programmiersprachen an realen Dingen und Attributen, sowie deren Handhabung. Das reale Objekt wird hierbei mithilfe einer Klasse abgebildet und durch Attribute, sowie Methoden charakterisiert. Hierdurch kann die Komplexität der entstehenden Programme deutlich gesenkt und reale Prozesse detaillierter abgebildet werden. Zudem wird die Wiederverwendbarkeit einzelner Programmabschnitte bspw. durch Vererbung deutlich gesteigert. So dient eine Klasse in der objektorientierten Programmierung als ”Bauplan” für ein bestimmtes Objekt, aus welcher weitere Klassen mit ähnlichen Eigenschaften abgeleitet werden können. Dies wird allgemein als Vererbung bezeichnet und dient vorrangig dazu, bereits vorhandene Eigenschaften einer (Basis-)Klasse zu übernehmen. Im Rahmen der Vererbung gibt es jedoch auch die Möglichkeit die Eigenschaften der Basisklasse zu überschreiben, was als *Polymorphie* bezeichnet wird. Hierbei können bspw. Datentypen und Bezeichner neu definiert und Methoden umgestaltet werden. Weiterführend zeichnen sich einzelne Programmiersprachen insbesondere durch ihre Performance und effiziente Anwendbarkeit innerhalb spezifischer Szenarien aus. So wird die Programmiersprache Java bspw. aufgrund ihrer kompakten Laufzeitumgebung zumeist im Bereich von Anwendungen auf Webseiten angewandt, wohingegen die maschinennahe Hochsprache C++ durch ihre performante Interpretationsweise häufig Anwendung in der Programmierung von Spielen oder Desktopanwendungen findet. [PH09]

Mit Bezug auf den Kontext des hier behandelten Use Case der Aggregation generischer OPC UA Server im Rahmen eines Forschungsprojektes, fällt die Wahl der Programmiersprache auf Python. Im Gegensatz zu deutlich komplexer aufgebauten Hochsprachen, wie bspw. C++, lassen sich mithilfe von Python Automatismen und Abläufe mit einem deutlich niedrigeren Zeitaufwand zügig implementieren. Dies liegt grundlegende an der einfach zu verstehenden Syntax von Python, welche der englischen Sprache ähnelt. Des weiteren sorgen die inhärenten Einrückungen innerhalb der Syntax für eine übersichtliche Programmstruktur, welche selbst von Anfängern im Bereich der Programmierung zügig interpretiert und

angewandt werden kann. Im Vergleich zu C++ gilt Python jedoch als weniger performante Hochsprache, was insbesondere bei großen und komplexen Prozessen in langen Rechenzeiten resultieren kann. Dies liegt grundlegend an der Interpretationsweise von Python. Während Programme in C++ direkt in Maschinensprache übersetzt (kompiliert) werden können, wird im Fall von Python-Code eine zwischengeschaltete Übersetzung (Interpretation) benötigt, welche zu einem Zeitverlust führt. Im aktuellen Use Case innerhalb des Forschungsprojektes MFlex2025 steht jedoch die Machbarkeit einer Softwarelösung zur Aggregation von OPC UA Servern im Vordergrund, wodurch die Performance der Implementierung, wie bereits in Kapitel 4.1 erläutert, lediglich als nebensächlich zu bewerten ist (siehe 4.2, Performance Anforderungen). Um eine zeitnahe und vor allem leicht nachvollziehbare Lösung zur beschriebenen Problemstellung der Aggregation zu realisieren, stellt Python daher eine gute Wahl unter der Vielzahl an objektorientierten Programmiersprachen dar. Zudem gibt es bereits einige Bibliotheken, die den Umgang mit OPC UA in Python unterstützen und grundlegende Routinen zur Client-Server-Kommunikation bereitstellen. Auf diese wird später im Kapitel vertiefend eingegangen.

Wie bereits beschrieben müssen Programme, welche in der Programmiersprache Python verfasst wurden, mithilfe eines *Interpreter* analysiert und anschließend durch einen Compiler in Maschinensprache übersetzt werden. Um als Entwickler Code verfassen zu können bedarf es zusätzlich eines sogenannten Editors. Dieser ermöglicht hierbei lediglich das Verfassen von einfachem Code. Wird der Editor durch einen entsprechenden Interpreter, dem Compiler und weiteren Funktionen ergänzt, spricht man von einer Entwicklungsumgebung (engl. *Integrated Development Environment* (IDE)). Die IDE sorgt so für eine grafische Schnittstelle zwischen Mensch und Maschine und kann durch Erweiterungen, bspw. zur Unterstützung beim Verfassen von Code mit korrekter Syntax, beliebig angepasst werden. Weiterführend liefern die meisten IDE zusätzlich einen sogenannten Debugger. Dieses Tool erlaubt dem Entwickler beim Test der Software in laufende Prozesse einzugreifen und bspw. bestimmte Werte zu bestimmten Zeitzuständen zu lesen oder Fehler im Programm zu analysieren.

Um ein Programm ausführen zu können, wird der Prozessablauf (Interpretation, Kompilierung, Ausführung etc.) innerhalb eines sogenannten *(Kernel-)Threads* (Ausführungsstrang) gestartet. Ein Thread bildet hierbei genau einen Prozess zur Abarbeitung des im Programm definierten Ablaufs ab. Innerhalb dieses einen Threads können demnach Aktionen und Berechnungen der CPU nur sequenziell ausgeführt werden. Im Kontext des Programmes bedeutet dies, dass lediglich eine Zeile Code zur zeit in einem Thread ausgeführt werden kann. Da es besonders bei objektorientierten Implementierungen jedoch auch zu einer Vielzahl an von einander unabhängigen Prozessschritten kommen kann, wird die Ausführung des Programmes, unter Verwendung von nur einem Thread, unnötig verlängert. So können z.B. insbesondere im Bereich der Kommunikation zwischen Client und Server innerhalb eines Netzwerkes längere Wartezeiten zwischen Anfrage und Antwort entstehen, die den Thread blockieren können. In solch einem Fall kann der Thread innerhalb der Wartezeit keine weiteren Aktionen durchführen, selbst wenn sie vollkommen unabhängig ausführbar wären. Wie bereits in Abbildung 2.3 für den Bereich der Kommunikation beschrieben, kann auch hier das Prinzip eines asynchronen Ablaufs zur Lösung des Problems genutzt werden. Im Kontext der Kommunikation zwischen Server und Client wurde dies durch das Publisher-Subscriber-Modell gelöst. Bezogen auf die objektorientierte Programmierung mit Python gibt es hier zwei verschiedene Lösungsansätze, das sogenannte Multi-Threading und die asynchrone Programmie-

rung mithilfe der Bibliothek AsyncIO. Beide Ansätze lösen das zeitkritische Problem nachhaltig, beruhen hierbei jedoch auf grundlegend verschiedenen Methodiken.

Das sogenannte Multi-Threading basiert hierbei auf dem wohl naheliegendsten Ansatz, weitere Threads für die gleichzeitige Ausführung unterschiedlicher Abläufe zu starten. So ist der Main-Thread (Hauptstrang), in welchem das Programm ausgeführt wird, in der Lage andauernde Tätigkeiten durch das eröffnen neuer Threads "auszulagern" (siehe Abb. 4.3). Dies sorgt einerseits für einen deutlich schnelleren Programmablauf, jedoch ist die Zusammenführung der einzelnen Threads oft mit komplexen, zeitkritischen Prozessen verbunden. Zudem ist die Kommunikation der Threads untereinander oft nicht leicht realisierbar. So kann es für den Fall, dass ein Thread Informationen eines anderen benötigt, oft zu Problemen bei der Ausführung der Anwendung kommen. Dennoch gilt der Multi-Threading Ansatz als beliebtes Vorgehen bei Erstellung komplexer Programme, insbesondere unter Verwendung von rechenstarken Systemen mit mehreren CPU's.

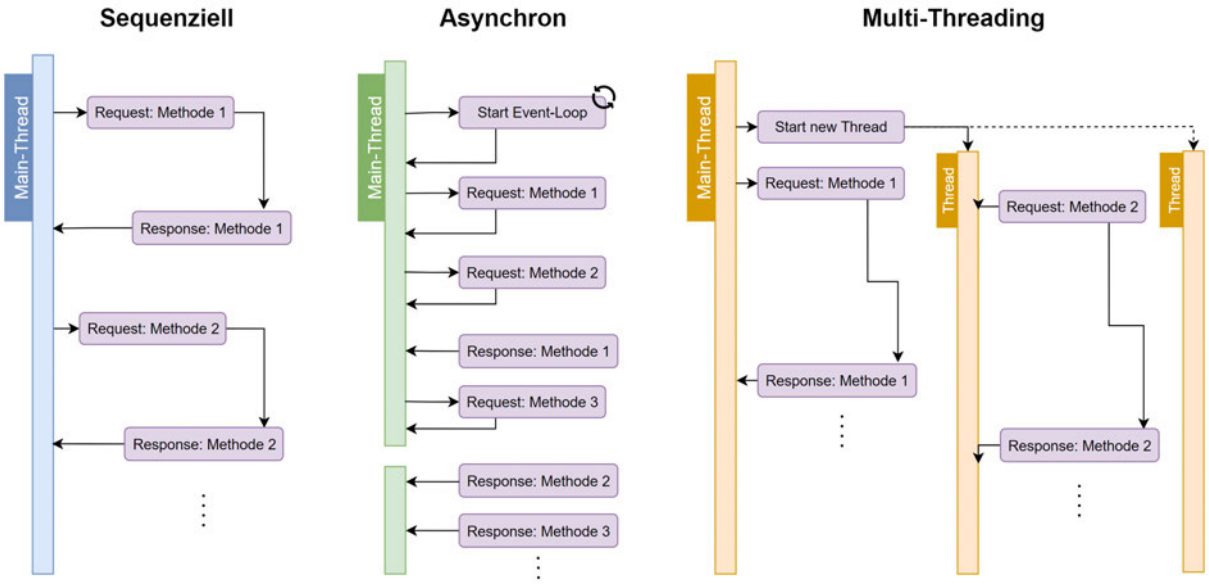


Abbildung 4.3: Vergleich Asynchrone Programmausführung und Multi-Threading

Eine grundlegend andere Lösung liefert die direkte Programmierung asynchroner Abläufe mithilfe der Bibliothek *AsyncIO*. AsyncIO, welche bereits seit Python 3.4 Teilmodul der Standardimplementierung ist, liefert Funktionalitäten zum Umgang mit asynchronen Prozessabläufen (engl. *Coroutines*) auf nur einem Thread. Im Gegensatz zum Multi-Threading werden somit keine weiteren Threads benötigt. Der Grundgedanke von AsyncIO ist hierbei die Definition kooperativer Funktionen, welche in ihrer Ausführung an bestimmten Stellen unterbrochen werden können (siehe Abb. 4.3). Um Wartezeiten im Programmablauf zu minimieren, wird dem ausführenden Thread so die Möglichkeit geboten, mit der Ausführung anderer, asynchroner Teilabläufe fortzufahren. Es entsteht der Anschein, dass unabhängige Teilabläufe gleichzeitig ausgeführt werden, wohingegen sie eigentlich vielmehr abwechselnd bearbeitet werden. Ebenso wie beim Multi-Threading ist das Resultat auch hier eine gesteigerte Zeiteffizienz.

Die Grundlage asynchroner Implementierungen mit AsyncIO stellt der *Event-Loop* dar, welcher bei Ausführung des Programmes gestartet wird. In Form einer andauernden Ereignisschleife werden in ihm alle asynchronen Prozessabläufe koordiniert ausgeführt. Er bildet daher den Hauptprozess im Thread und prüft kontinuierlich, wo asynchron ausführbare Aktionen beginnen, andauern oder beendet werden. Zur Definition dieser Abläufe stellt AsyncIO drei essentielle Funktionalitäten zur Verfügung, die an die Schlüsselwörter *async*, *await* und *task* gebunden sind. So werden asynchron ausführbare Programmsegmente (bspw. Methoden/Funktionen) durch das vorangestellte Schlüsselwort *async* gekennzeichnet. Der Begriff *async* deklariert hiermit direkt im Code die Abschnitte, die später als Coroutine asynchron ausgeführt werden dürfen. Innerhalb dieser Coroutines kommt nun das Schlüsselwort *await* zum Einsatz, welches die kooperativen Stellen der Routine kennzeichnet. An diesen Stellen kann später das Programm unterbrochen werden, um Ressourcen für andere Prozesse bereitzustellen und blockierende Tätigkeiten zu umgehen. Darauf aufbauend stellt AsyncIO zudem Funktionalitäten zum Erstellen und Verwalten sogenannter *Tasks* zur Verfügung. Der Grundgedanke hinter der Nutzung von Tasks ist es, asynchrone Teilprogramme zu starten, ohne auf ihre Ausführung warten zu müssen. Der Entwickler bekommt somit die Möglichkeit vollkommen unabhängige Teilprozesse ohne weiteren Einfluss auf dem Programmablauf nebenläufig auszuführen und zu überwachen. Die Ausführung geschieht hierbei so zeitnah wie möglich, jedoch ist es unbekannt und unerheblich, ob sie parallel oder in Etappen erfolgt. Wird eine Task anschließend durch das *await*-Schlüsselwort belegt, wird an diesem Punkt des Programmes auf die Ausführung der Coroutine innerhalb der Task gewartet.

Im Kontext der Implementierung einer OPC UA Server-Aggregation für das Use-Case von MFlex2025 stellen beide Lösungsmöglichkeiten, sowohl das Multi-Threading, als auch die asynchrone Implementierung mittels AsyncIO, eine zeit- und ressourceneffiziente Variante dar. Während Multi-Threading jedoch eher für rechenintensive und langwierige Implementierungen auf großen Systemen vorgesehen ist, bietet die asynchrone Variante den unmittelbaren Vorteil einer zügigen und übersichtlicheren Umsetzung. Auch wenn das Multi-Threading, bei entsprechend verfügbarer Rechenleistung, eindeutig die leistungsstärkere Variante darstellt, kann im Rahmen von MFlex2025 auf die hochkomplexe und zeitkritische Kommunikation zwischen einzelnen Threads und dem damit verbundenen, oft hohen Implementierungsaufwand verzichtet werden. Die asynchrone Implementierung bietet hingegen ebenfalls eine zeiteffiziente und ressourcensparende Variante und kann zudem durch die Bibliothek *opcua-asyncio* (*AsyncUA*, siehe [Hei22]) seitens der Kommunikation mithilfe von OPC UA erweitert werden. AsyncUA bietet Entwicklern so bereits server- und client-seitige Implementierungen asynchroner Programmstrukturen zur Kommunikation mit OPC UA und stellt Funktionalitäten wie bspw. OPC UA-Services zur Verfügung. Die Bibliothek erlaubt dem Entwickler somit nicht nur die Verwendung asynchroner Programmstrukturen zur zeiteffizienten Ausführung, sondern bietet zudem grundlegende Funktionalitäten zur prozessspezifischen Implementierung unter Verwendung von OPC UA und der Kommunikation zwischen Client und Server an.

4.3 Architektur und Design

Dieses Kapitel beschreibt die Lösungsfindung in Form eines Konzeptes zur Aggregation generischer Knotenstrukturen des OPC UA Adressraumes. Hierzu wird zunächst der grundlegende Ablauf der Aggregation, unter Berücksichtigung der Anforderungen an die Software, konzeptionell definiert. Basierend auf diesem Ablauf wird anschließend ein Klassenmodell entwickelt, welches die Aufgabenpakete des Ablaufs aufgreift und somit die einzelnen Programmbausteine definiert. Das Klassenmodell baut hierbei grundlegend auf den bereits existierenden und in Kapitel 2 behandelten, Konzepten auf.

Um ein Konzept zur Aggregation generischer Knotenstrukturen anderer OPC UA Server zu entwickeln, muss zunächst untersucht werden, inwiefern einzelne Knoten systematisch zu aggregieren wären und welche Softwarekomponenten hierfür grundlegend benötigt werden würden. Hierzu werden die in Kapitel 2.4 bereits vorgestellten Konzepte als Grundlage dafür genutzt, ein neues Konzept, passend für den Use Case MFlex2025, abzuleiten. Bevor der eigentliche Prozess der Aggregation jedoch näher analysiert werden kann, muss zunächst festgelegt werden, wie die Softwareimplementierung zu konfigurieren sein soll. Wie der Anforderung 6 des Product-Backlog zu entnehmen ist, soll hierbei auf eine möglichst unkomplizierte und zeitsparende Konfiguration geachtet werden. Auf Grundlage der beiden Konzepte aus Kapitel 2.4, ergeben sich hierzu zwei mögliche Umsetzungsvorschläge. Zum einen könnte die Konfiguration, wie von Forschern der Aalto Universität beschrieben, als manuelle Eingabe in eine Konfigurations-Datei durchgeführt werden. Hierfür entwickelte die Forschungsgruppe aus Finnland eine spezifische Notation, mithilfe derer die Konfiguration in Form eines Mappings von Knoten beider Server gespeichert und später von der Softwareimplementierung eingelesen werden kann. Die Forschungsgruppe der TU Ingolstadt hingegen, stellte ein Konzept zur Konfiguration der Aggregation direkt im Adressraum des Servers vor. Auf Basis eines, eigens für die Software entwickelten, Informationsmodelles kann der Nutzer so die Aggregation durch Client-Zugriffe auf Variablen und Methoden konfigurieren und separat starten. Dieses Konzept bietet hierbei, gegenüber dem Modell aus Finnland, den Vorteil, dass die Aggregation für den Nutzer jederzeit anpassbar bleibt, ohne einen Server-Neustart erzwingen zu müssen.

Auf Basis dessen wird auch im Zuge der Aggregation am Beispiel von MFlex2025 ein Informationsmodell zur Konfiguration genutzt, welches in Kapitel 4.4 detailliert erläutert wird. Erweiternd zum Vorgehen der TU Ingolstadt bietet dieses jedoch neben der Konfiguration im Adressraum nach Server-Start, zudem auch die Möglichkeit den Adressraum des aggregierenden Servers vor Server-Start vollkommen frei zu gestalten und aggregierte Strukturen an beliebigen Stellen im Informationsmodell einzubinden. Dies geschieht, indem im Informationsmodell Knoten spezifischen Objekttyps (*AggregationEndpointtype*, siehe 4.4)) definiert werden, sodass die Softwareimplementierung diese nach Import des NodeSets finden und auslesen kann. Diese Knoten werden im Folgenden als AggregationEndpoints bezeichnet, da sie die genaue Position markieren, an der die aggregierten Knotenstrukturen nach dem Aggregationsprozess anknüpfen werden (siehe Abb. 4.4). Neben der Position der AggregationEndpoints definiert das Informationsmodell an dieser Stelle zudem relevante Parameter, die die Verbindung zum Quellserver spezifizieren.

Wie anhand von Abbildung 2.4 in Kapitel 2.2 erläutert, besteht der Adressraum eines OPC UA Server aus Knotenstrukturen unterschiedlicher Knotentypen. Jeder Knoten besitzt hierbei eigene Attribute und

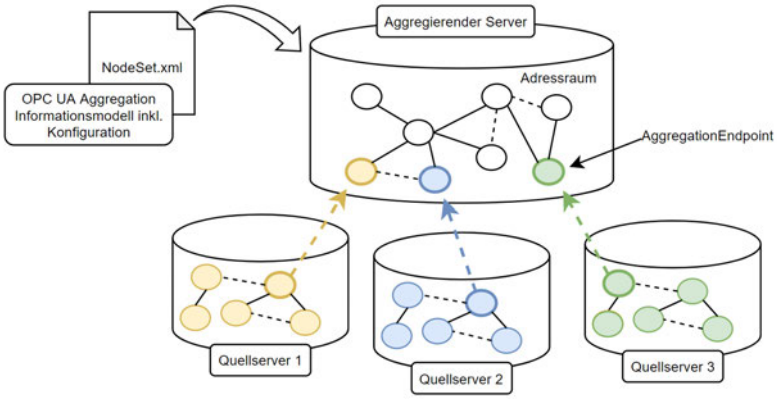


Abbildung 4.4: Konzept zur Konfiguration der Aggregation anhand eines Informationsmodelles

Referenzen, die den Knoten und seine Beziehungen zu anderen Knoten beschreiben und ihm spezifische Eigenschaften verleihen. Da es sich bei der Aggregation von Knoten grundlegend erst einmal um die Erstellung einer Kopie des Knotens auf dem aggregierenden Server handelt, müssen die notwendigen Informationen über den Knoten zunächst aufgenommen und gespeichert werden. Zum Austausch dieser Informationen muss jedoch zunächst eine Verbindung zum Quellserver hergestellt werden. Wie in Abbildung 2.6 aus dem Konzept der Aalto Universität abzuleiten, wird hierfür ein sogenannter OPC UA Client benötigt. Mithilfe der Konfiguration des AggregationEndpoints ist dieser Client anschließend dazu in der Lage eine Verbindung zum Quellserver herzustellen und OPC UA Service-Anfragen (bspw. Browse-Request) durchzuführen. Auf diese Weise können die benötigten Daten der relevanten Knoten zyklisch gesammelt und an den aggregierenden Server weitergeleitet werden. Basierend auf diesen Informationen kann der aggregierende Server nun ein genaues Abbild des Knotens in seinem Adressraum, an entsprechender Stelle, erstellen. Im Fall des ersten zu aggregierenden Knotens, wäre dies die Stelle des dazugehörigen AggregationEndpoint. Dieser Prozess wird hierbei im Zuge der Aggregation grundlegend für jeden Knoten separat durchgeführt.

Um die Prozesse der Konfiguration und Aggregation im Hinblick auf den Gesamtprozess besser einordnen zu können, wurde das in Abbildung 4.5 dargestellte, Vier-Phasen-Konzept entwickelt. Die vier Phasen bilden hierbei alle notwendigen Prozessschritte grob ab, die nach aktuellem Stand der Entwicklung benötigt werden, um den aggregierenden Server zu starten, die Aggregation durchzuführen und den Adressraum im Anschluss OPC UA konform nutzen zu können.

Die erste Phase dient hierbei der, bereits thematisierten, Konfiguration der Aggregation und des Servers. Wie zuvor erläutert wird hierzu vom Benutzer ein Informationsmodell, basierend auf dem Aggregation-NodeSet (siehe 4.4), definiert. Das Informationsmodell enthält hierbei dedizierte AggregationEndpoints, an denen die aggregierten Knotenstrukturen später anknüpfen können. Neben dem Informationsmodell müssen jedoch zuvor weitere Parameter zur Konfiguration des Servers selbst definiert werden. Hierzu zählen der Server-Name, die URL und der Endpoint unter welchem der aggregierende Server nach dem Start für Client-Zugriffe erreichbar sein soll. Da diese Informationen bereits vor Import des Informationsmodelles zur Verfügung stehen müssen, werden diese direkt im Code der Anwendung als Attribute des Servers definiert und später in Kapitel 4.5 erläutert.

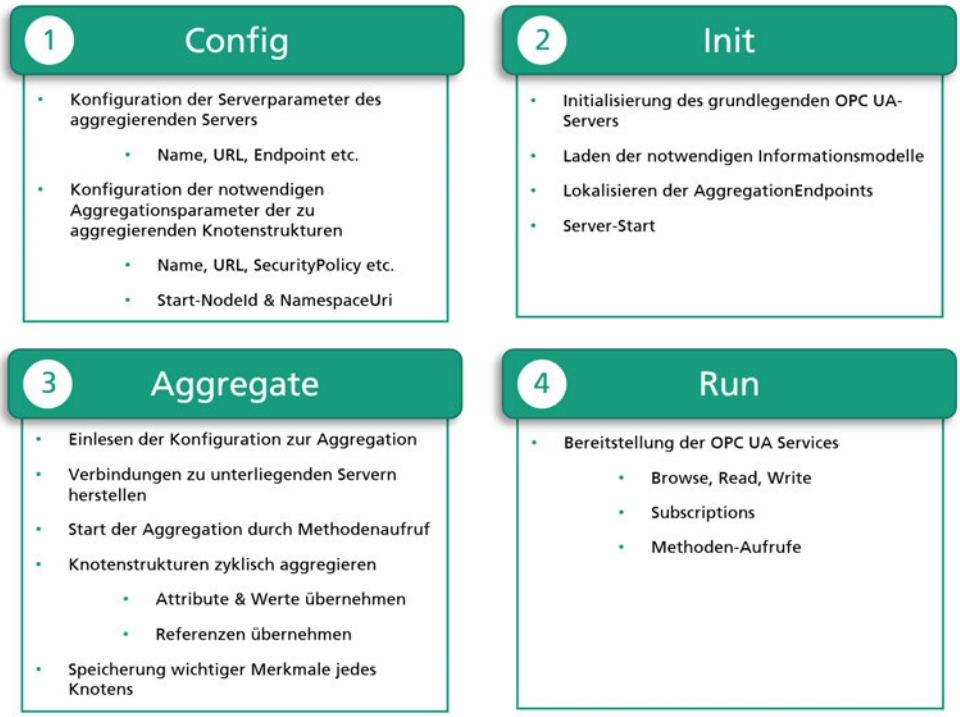


Abbildung 4.5: Phasenkonzept des Ablaufs der Aggregation

In Phase zwei des Phasenkonzeptes aus Abbildung 4.5 geht es weiterführend um die Initialisierung des aggregierenden OPC UA Server. Hierzu wird zunächst ein grundlegend leerer OPC UA Server erzeugt, welcher lediglich das Basis-NodeSet der OPC Foundation beinhaltet und im Anschluss anhand der zuvor im Code definierten Parameter konfiguriert werden kann. Darauf aufbauend werden die benötigten Informationsmodelle, zu denen auch das Informationsmodell zur Konfiguration der Aggregation zählt, in den Adressraum des aggregierenden Servers importiert. Ist dies reibungslos durchgeführt worden, kann die Softwareimplementierung den Adressraum des aggregierenden Server nach den AggregationEndpoints durchsuchen und die relevanten Parameter zur Identifizierung dieser Knoten speichern. Im Anschluss darauf folgt der eigentliche Start des Servers.

Ist der Server gestartet und unter der vordefinierten URL für Client-Zugriffe erreichbar, wird seitens der Software auf den Start der Aggregation gewartet (siehe Abb. 4.5, Phase drei). Wie im nachfolgenden Kapitel 4.4 detaillierter erläutert, wird die Aggregation hierbei durch einen Methodenaufruf des Client gestartet. Wie zuvor bereits erläutert benötigt die Aggregation generischer Knotenstrukturen eine sichere Verbindung zum Quellserver. Nach dem Start der Aggregation werden daher die Werte der relevanten Variablen des AggregationEdnpoint eingelesen und alle notwendigen Verbindungen durch die OPC UA Clients hergestellt. Im Anschluss daran, wird mit der zyklischen Aggregation der relevanten Knotenstrukturen begonnen. Wie bereits erwähnt wird die Aggregation hierbei für jeden Knoten separat durchgeführt, indem die Software notwendige Informationen (Attribute, Werte und Referenzen) über den jeweiligen Knoten sammelt und diesen im Anschluss im Adressraum des aggregierenden Server erstellt. Des weiteren werden die Informationen jedes Knoten zwischengespeichert, um Änderungen bestimmter Werte später abgleichen zu können. Darauf aufbauend kann somit später der Zusammenhang des

neu erstellten Knotens auf dem aggregierenden Server zum ursprünglichen Knoten auf dem Quellserver leichter nachvollzogen werden.

Nachdem die eigentliche Aggregation aller Knotenstrukturen abgeschlossen ist, geht die Implementierung in Phase vier “Run“ über. Diese Phase beschreibt hierbei die gesamte Laufzeit nach dem Start des OPC UA Servers, in welcher keine aggregierenden Prozesse stattfinden. Sollte indes eine weitere Aggregation zu einem spätere Zeitpunkt ausgelöst werden, springt die Implementierung erneut zu Phase drei zurück. Während sich der Server in Phase vier befindet, besteht die Hauptaufgabe der Implementierung in der Bereitstellung der OPC UA Services. So muss die Software im Fall einer Service-Anfrage durch einen Client, die notwendigen Prozesschritte auf dem Quellserver auslösen, Informationen weiterleiten und geänderte Werte im Adressraum des aggregierenden Server aktualisieren. Wird bspw. eine Subscription-Anfrage auf einen bestimmten Knoten gestellt (MonitoredItem, siehe 2.3), muss die Implementierung die Verbindung zum Quellserver herstellen und eine entsprechende Service-Anfrage platzieren. Sollte sich nun eine Änderung der Daten und Informationen des Knoten ergeben, muss die Software diese auf dem Quellserver aufnehmen und an den aggregierenden Server weiterleiten.

Nachdem nun der strukturelle Aufbau der Konfiguration und Nutzung der Softwareimplementierung im Phasenkonzept definiert worden sind, können die beschriebenen Prozesse entsprechenden Klassen zur objektorientierten Programmierung zugeordnet werden. Hierfür wurde das, in Darstellung 4.6 abgebildete, grundlegende Klassenkonzept erarbeitet.

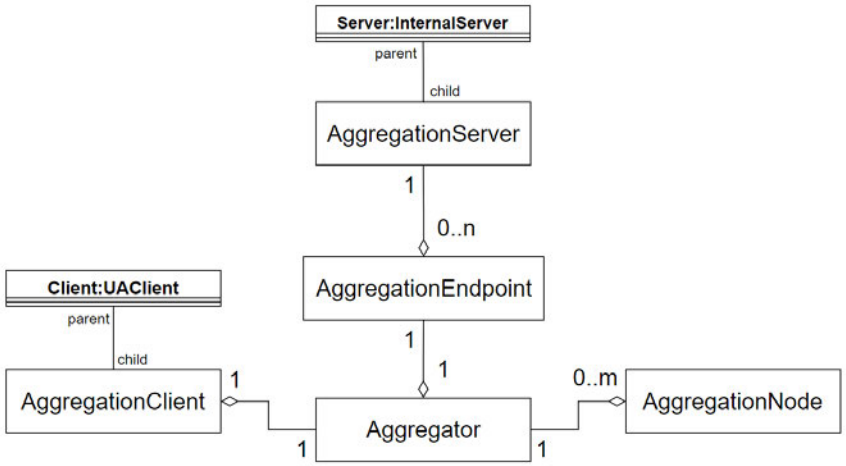


Abbildung 4.6: Grundlegendes Klassenkonzept der OPC UA Server Aggregation

Wie der Abbildung zu entnehmen, wird die Implementierung grundlegend aufgeteilt in fünf Klassen. Diese Klassen werden später zur Erstellung von Objekten genutzt, welche die programmierten Eigenschafte und Methoden der Klassen beinhalten (siehe Kapitel 4.2). Bei Erstellung der Abbildung wurde sich an den grundlegenden Regelungen eines Unified Modeling Language-Klassendiagramms (UML-Klassendiagramm) orientiert, wodurch die Pfeile zwischen den Klassen, die relative Anzahl an erstellten Objekten wiedergeben. Des weiteren werden die Eltern-Klassen, die durch die Vererbung in direkter

Abhängigkeit mit den abgeleiteten Klassen zur Aggregation stehen, durch Pfeile mit der Beschriftung “parent/child“ dargestellt.

Wie dahingehend der Abbildung zu entnehmen, bildet die Klasse *AggregationServer*, welche von der Klasse *Server* der Bibliothek AsyncIO abgeleitet wird, die zentrale Komponente der Softwareimplementierung. Sie ist die einzige Klasse, von der nur ein Objekt, der aggregierende OPC UA Server, erstellt wird. Hauptaufgabe dieser Klasse ist die Bereitstellung des OPC UA Server selbst, sowie der Import notwendiger Informationsmodelle in den OPC UA Adressraum. Des weiteren liest diese Klasse die Konfiguration zur Aggregation aus dem Adressraum ein und erstellt Instanzen der Klasse *AggregationEndpoint* und *Aggregator* (siehe 4.5, Phase zwei). Diese beiden Klassen werden demnach, im Verhältnis zum AggregationServer, beliebig oft als Objekt erstellt, da mit jeder zu aggregieren Knotenstruktur ein neuer AggregationEndpoint benötigt wird. Zudem wird für jedem AggregationEndpoint-Objekt genau eine Aggreator-Instanz erstellt. Die AggregationEndpoint-Instanz nimmt hierbei alle notwendigen Informationen zur Aggregation der Knotenstrukturen, an genau diesem Punkt im Adressraum des aggregierenden Server auf. Die dazugehörige Instanz der Aggregator-Klasse sorgt anschließend für den Verbindungsaufbau zum Quellserver durch Erstellung eines Objektes der Klasse *AggregationClient*. Da pro AggregationEndpoint nur eine Verbindung aufgebaut wird, muss die Aggregator-Instanz hierbei auch nur ein AggregationClient-Objekt erstellen. Zum Abschluss der Initialisierungsphase (Phase zwei) startet die AggregationServer-Instanz den OPC UA Server.

Im Zuge der Aggregation (Phase drei) werden die meisten Prozessschritte durch das Aggregator-Objekt vorgegeben. Die Aggregator-Klasse enthält hierbei Methoden, die den genauen Ablauf der Aggregation eines Knotens definieren und das Zusammenspiel mit Objekten der AggregationClient-Klasse, sowie dem AggregationServer-Objekt koordinieren. Für jeden zu aggregierenden Knoten wird hierbei eine Instanz der Klasse *AggregationNode* erstellt. Diese Objekte dienen vorrangig als Träger der Informationen des ursprünglichen Knotens auf dem Quellserver, indem sie diese in ihren Attributen speichern. Die Attribute der AggregationNode-Instanzen werden hierbei von der jeweiligen AggregationClient-Instanz beschrieben. Im Anschluss reicht dieser die AggregationNode-Objekte an die Aggregator-Instanz weiter, welche nun dem AggregationServer-Objekt anordnet den Knoten entsprechend zu erstellen. Dieser Vorgang geschieht hierbei zyklisch in einer Schleife und wird asynchron ausgeführt. Das Zusammenspiel von AggregationServer, AggregationClient und Aggregator wird hierbei später in Kapitel 4.5 detailliert erläutert und mit den entsprechenden Stellen im Code systematisch verknüpft.

Nach Phase drei, folgt die eigentliche Laufzeit des aggregierenden Servers. Wie bereits erwähnt dient Phase vier hierbei grundlegend der Bereitstellung der OPC UA Services. Diese Aufgabe obliegt ebenfalls der AggregationServer-Instanz. Im Falle einer client-seitigen Service-Anfrage, ermittelt der AggregationServer das zuständigen AggregationEndpoint-Objekt bzw. Aggregator-Objekt, welcher die Anfrage in Zusammenarbeit mit seinem AggregationClient-Objekt an den Quellserver weiterleitet. Hierbei wird zunächst der ursprüngliche Knoten von Aggregator-Instanz ermittelt und anschließend vom AggregationClient-Objekt im Adressraum des Quellserver identifiziert. Im Anschluss kann die Service-Anfrage ausgeführt und die Antwort über den AggregationClient und den Aggregator an den AggregationServer weitergeleitet werden. Im Fall von event-basierten Antworten des Quellserver, wie bspw. bei einer Subscription auf die Änderung von Daten einzelner Knoten über einen bestimmten Zeitraum, muss

der Client kontinuierlich Informationen an die Aggregator-Instanz weiterleiten, um jede Datenänderung des ursprünglichen Knotens auf dem aggregierenden Server bereitstellen zu können.

4.4 Informationsmodell

Wie bereits im vorherigen Kapitel beschrieben, soll zur Konfiguration des aggregierenden OPC UA Servers ein Informationsmodell mit notwendigen Informationen zur Aggregation erstellt werden. Aufgrund dessen beschäftigt sich dieses Kapitel vorrangig mit dem Basis-NodeSet, welches im Rahmen dieser Ausarbeitung definiert wurde und aus welchem das Informationsmodell dem aggregierenden Servers später abgeleitet werden soll. Hierfür wird zunächst erneut der Begriff des Informationsmodelles erläutert, um anschließend das NodeSet grundlegend vorzustellen. Darauf aufbauend können die im NodeSet verwendeten, relevanten Objekt- und Datentypen erläutert und die Anwendung anhand des Use Case MFlex2025 aufgezeigt werden.

Das Informationsmodell eines OPC UA Servers definiert die Struktur- und Knoteninformationen der Knoten auf dem Server. Das Modell gibt somit vor, welche Knotentypen zur Verfügung stehen, wie diese durch Referenzen verknüpft sind und welche Instanzen in Form von Objekten, Variablen und Methoden enthalten sind. Wie in Kapitel 2.3 bereits angesprochen, können Informationsmodelle aus mehreren NodeSets bestehen, die auch Abhängigkeiten untereinander besitzen können (siehe Abb. 2.4). Des weiteren können diese NodeSets mithilfe der OPC UA Specification Part 6 im XML-Format gespeichert und somit weitergereicht werden. Durch die Überführung der Informationen des NodeSet in das XML-Format wird es dem Entwickler ermöglicht, das NodeSet in einen sogenannten OPC UA Modelling-Editor zu importieren, um dort Anpassungen vorzunehmen. Vor Server-Start wird das Informationsmodell vom Server eingelesen und im Adressraum bereitgestellt. Auf die selbe Weise können so im Modelling-Editor weitere NodeSets importiert werden, um bereits definierte Strukturen übernehmen zu können, oder auf ihnen aufzubauen. So wird auch im Fall der Aggregation generischer OPC UA Server im Rahmen dieser Ausarbeitung ein spezielles Aggregation-NodeSet vordefiniert, welches spezifische Objekttypen beinhaltet, um eine Konfiguration, passend zur Software-Implementierung, realisieren zu können. Im Gegensatz zu den in Kapitel 2.4 vorgestellten Konzepten, ergibt sich so der Vorteil, dass der Adressraum des aggregierenden OPC UA Servers direkt vom Nutzer, noch vor Server-Start, detailliert definiert werden kann. Je nach Anwendungszenario können so beliebige Strukturen erstellt und Knotenstrukturen der Quellserver an gewünschter Position eingebaut werden. Durch den manuellen Start der Aggregation in Form einer Client-Anfrage (bspw. ein Methoden-Aufruf, siehe 2.3), kann anschließend zudem gewährleistet werden, dass die Konfiguration der Aggregation auch nach Server-Start vom Nutzer angepasst werden können. Hierzu werden die relevanten Variablen durch einen Client-Zugriff einfach neu beschrieben.

Um die Konfiguration einzelner Aggregationspunkte im Informationsmodell möglichst vereinheitlicht zu gestalten, definiert das Aggregation-NodeSet den *AggregationEndpointType*. Der AggregationEndpoint, welcher vom AggregationEndpointType abgeleitet wird (siehe Kapitel 2.3), markiert hierbei genau die Knoten im Informationsmodell, an denen später eine Aggregation generischer Strukturen anknüpfen soll. Des Weiteren definiert der AggregationEndpointType drei essentielle Variablen, welche Informationen

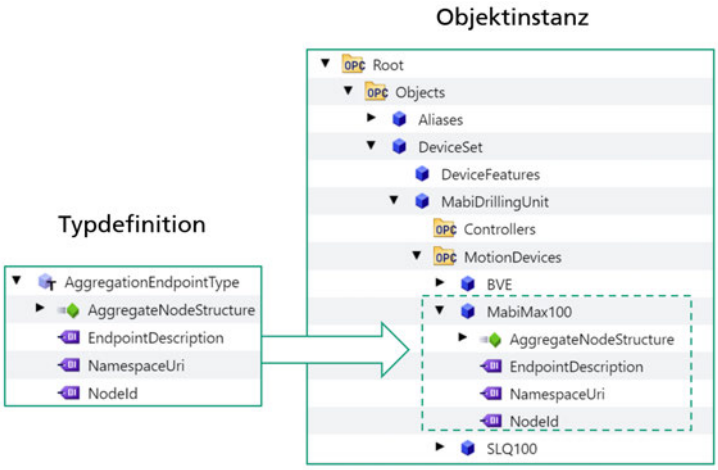


Abbildung 4.7: AggregationEndpointType zur Konfiguration der Aggregation

über den Quellserver bzw. den ersten zu aggregierenden Knoten enthalten. Der Datentyp dieser Variablen spielt hierbei eine essentielle Rolle auf die später näher eingegangen wird. Wie in Abbildung 4.7 zu sehen, wird neben den drei Variablen zudem eine Methode zum Start der jeweiligen Aggregation von der Typdefinition bereitgestellt. Die Abbildung zeigt hierbei das Beispiel einer Konfiguration am Use Case MFlex2025. Neben dem Industrieroboter MabiMax100, wird hier auch die Aggregation der Bohrvorschubeinheit, sowie der Spindel entsprechend angepasst. Im Anschluss kann das Informationsmodell gespeichert und dem AggregationServer zur Verfügung gestellt werden. Auf diese Weise kann die Softwareimplementierung nach Server-Start die benötigten Informationen zum Aufbau von Verbindungen bzw. zur Aggregation der Knotenstrukturen aus dem Adressraum des Servers auslesen.

Wie bereits erwähnt kommt den Datentypen der drei definierten Variablen des AggregationEndpointType eine besondere Bedeutung zuteil. So ist die Variable *EndpointDescription* bspw. vom gleichnamigen Datentyp *EndpointDescription*, welcher in Tabelle 4.3 tabellarisch darstellt ist. Wie der Tabelle zu entnehmen, definiert der Datentyp hierbei eine Struktur aus Sub-Informationen, welche neben dem Namen, sowie der URL des Quellservers auch weitere Informationen, wie bspw. zur Verschlüsselung der Verbindung enthalten. Die Softwareimplementierung kann diese Informationen im Anschluss auslesen, um die Verbindung zum Quellserver entsprechend herstellen zu können.

Neben der EndpointDescription definiert der AggregationEndpointType zudem die Variablen NodeId und NamespaceUri. Die NodeId definiert hierbei den Knoten im Adressraum des Quellservers, welcher als erstes aggregiert werden soll. Von ihm aus werden anschließend alle weiteren Knoten, unter Verwendung der OPC UA Services ermittelt und stückweise aggregiert. Die Variable NamespaceUri definiert einen String, welcher die Kennung des jeweiligen Namensraumes wiedergibt. Betrachtet man nun erneut die Definition der NodeId (siehe Kapitel 2.3), könnte angenommen werden, dass der Identifier bereits den Namensraum klar identifiziert. Wird der Quellserver nun allerdings durch weitere Namensräume erweitert, so kann der Identifier neu vergeben werden, wodurch es zur Aggregation ungewollter Knotenstrukturen kommen kann. Die NamespaceUri hingegen, definiert den Namensraum anhand seiner Bezeichnung, welche auf dem Server einzigartig sein muss. Auf Basis dessen, identifiziert die Softwa-

Tabelle 4.3: OPC UA Spec. Part 4: Services-Endpointdescription [OPC21], Tabelle 135

Name	Type	Description
EndpointDescription	structure	Describes an *Endpoint* for a *Server*.
endpointUrl	String	The URL for the *Endpoint* described.
server	ApplicationDescription	The description for the *Server* that the *Endpoint* belongs to. The *ApplicationDescription* type is defined in 7.2.
serverCertificate	ApplicationInstance Certificate	The *Application Instance Certificate* issued to the *Server*. The *ApplicationInstanceCertificate* type is defined in 7.3.
securityMode	Enum MessageSecurityMode	The type of security to apply to the messages. The type *MessageSecurityMode* type is defined in 7.20. A *SecureChannel* may need to be created even if the *securityMode* is NONE. The exact behaviour depends on the mapping used and is described in the OPC 10000-6.
securityPolicyUri	String	The URI for *SecurityPolicy* to use when securing messages. The set of known URIs and the *SecurityPolicies* associated with them are defined in OPC 10000-7.
userIdentityTokens []	UserTokenPolicy	The user identity tokens that the *Server* will accept. The *Client* shall pass one of the *UserIdentityTokens* in the *ActivateSession* request. The *UserTokenPolicy* type is described in 7.42.
transportProfileUri	String	The URI of the *Transport Profile* supported by the *Endpoint*. OPC 10000-7 defines URIs for the *Transport Profiles*.
securityLevel	Byte	A numeric value that indicates how secure the EndpointDescription is compared to other EndpointDescriptions for the same *Server*. A value of 0 indicates that the *EndpointDescription* is not recommended and is only supported for backward compatibility. A higher value indicates better security.

reimplementierung den Start-Knoten stets anhand der NodeId und zusätzlich durch eine Prüfung der NamespaceUri.

Auf Basis dieser Problematik, definierte die OPC Foundation in Part 4 ihrer Specifikation zu OPC UA, den Datentyp *ExpandedNodeId* ([OPC21], Tabelle 136). Dieser vereint die Informationen der beiden Variablen NodeId und NamespaceUri und stellt daher eine gute Alternative zum herkömmlichen String dar. Wie der Tabelle 4.4 zu entnehmen, definiert die ExpandedNodeId hierbei, ähnlich der EndpointDescription, eine Struktur aus Informationen, die neben dem NamespaceIndex auch die NamespaceUri und den Serverindex enthalten.

Tabelle 4.4: OPC UA Spec. Part 4: Services-ExpandedNodeId [OPC21], Tabelle 136

Name	Type	Description
ExpandedNodeId	structure	The *NodeId* with the namespace expanded to its string representation.
serverIndex	Index	Index that identifies the *Server* that contains the *TargetNode*. This *Server* may be the local *Server* or a remote *Server*. This index is the index of that *Server* in the local *Server*'s *Server* table. The index of the local *Server* in the *Server* table is always 0. All remote *Servers* have indexes greater than 0. The *Server* table is contained in the *Server Object* in the *AddressSpace* (see OPC 10000-3 and OPC 10000-5). The *Client* may read the *Server* table *Variable* to access the description of the target *Server*.
namespaceUri	String	The URI of the namespace. If this parameter is specified then the namespace index is ignored. 5.4 and OPC 10000-12 describes discovery mechanism that can be used to resolve URIs into URLs.
namespaceIndex	Index	The index in the *Server*'s namespace table. This parameter shall be 0 and is ignored in the *Server* if the namespace URI is specified.
identifierType	IdType	Type of the identifier element of the *NodeId*.
identifier	*	The identifier for a *Node* in the *AddressSpace* of an OPC UA *Server* (see *NodeId* definition in OPC 10000-3).

Wie in Kapitel 4.2 bereits beschrieben wird die Python-Bibliothek AsyncUA zur Umsetzung der OPC UA Server Aggregation genutzt. Die im Rahmen von MFlex2025 verwendete Version von AsyncUA beinhaltet hierbei jedoch noch keine Implementierungen des ExpandedNodeId-Datatypes, wodurch die

Handhabung mit Variablen diesen Typs deutlich erschwert wird. Wie Anforderung 8 zu entnehmen, ist die Entwicklung der Software inkl. Dokumentation zudem begrenzt auf 3 Monate (siehe 4.2). Durch diesen Umstand wird bei der Implementierung, welche in dieser Ausarbeitung thematisiert wird, auf die vereinfachte Struktur, bestehend aus zwei Variablen des String-Datatypes, statt einer Variable des ExpandedNodeId-Types zurückgegriffen.

4.5 Prozessstrukturelle Aggregation

Dieses Kapitel beschäftigt sich detailliert mit der Ausarbeitung der Software in Python. Neben dem genauen Ablauf der einzelnen, in Kapitel 4.3 bereits beschriebenen, Prozesse zur Konfiguration und Aggregation, wird hier ebenfalls die Programmstruktur und die damit verbundenen Methoden jeder verwendeten Klasse thematisiert. Dies sorgt so dafür, dass ein Zusammenhang zwischen dem zuvor entwickelten Phasenkonzept und den programmierten Funktionen im Code hergestellt werden kann. Im Zuge dessen werden zudem einzelne, besonders komplexe Abläufe der Softwareimplementierung, anhand des Codes ausführlicher erläutert.

Um einzelne Abläufe detailliert erläutern und in den Gesamtablauf der einzelnen Phasen des Phasenkonzeptes einordnen zu können, werden zur Verbildlichung Ablaufdiagramme genutzt. Wie in Abbildung 4.8 zu sehen, stellt dieser Diagrammtyp hierbei jeden Prozess der durch die Software ausgeführt wird dar und zeigt auf, welcher Klasse die dafür genutzte Methode zuzuordnen ist. Des weiteren wird der Prozess stichpunktartig beschrieben. Die Ablaufdiagramme sind hierbei unterteilt in die einzelnen Phasen, wobei die erste Phase (Konfiguration) nicht zum Programmablauf gehört und bereits in Kapitel 4.4 detailliert erläutert worden ist.

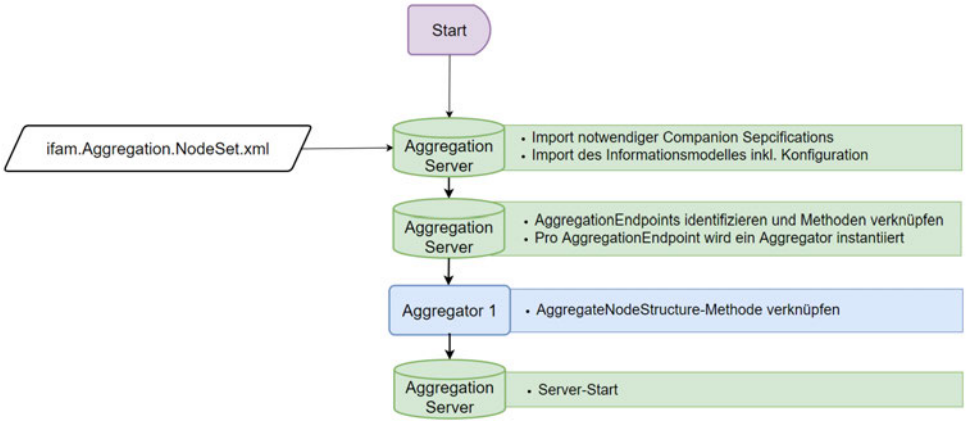


Abbildung 4.8: Konfiguration und Initialisierung des Aggregierenden OPC UA Server

Wie aus der Abbildung 4.8 ersichtlich, beginnt der Ablauf mit dem Start der Softwareimplementierung. Die Konfiguration der relevanten Parameter für den Server selbst, wie bspw. den Namen, die URL und den Endpoint des Servers, ist hierbei im Diagramm nicht enthalten, da Sie bereits vor dem Start der Software zu definieren sind. Dies geschieht indem zunächst eine Instanz der AggregationServer-Klasse erstellt wird. Im Anschluss werden die relevanten Attribute der Instanz direkt im Code beschrieben (siehe Abb. 4.11).

```
21      # Configure Server
22      server.ServerName = "MabiDrillingUnit"
23      server.Endpoint = "opc.tcp://127.0.0.1:4800"
24      server.NamespaceUri = "http://ifam.fraunhofer.de/UA/DI/Aggregation/MFLEX"
25      server.SecurityPolicy = "None"
```

Abbildung 4.9: Definition relevanter Parameter des Servers

Nachdem die AggregationServer-Instanz erstellt und konfiguriert worden ist, folgt der Import relevanter NodeSets. Hierzu zählt das NodeSet zur Konfiguration der Aggregation (ifam.Aggregation.NodeSet.xml), welches zuvor erstellt worden ist. Im Anschluss durchsucht der AggregationServer den Adressraum, um die relevanten AggregationEndpoint-Knoten lokalisieren zu können. Hierbei wird, wie bereits in Kapitel 4.3 beschrieben, für jeden AggregationEndpoint ein Objekt der gleichnamigen Klasse AggregationEndpoint und eine Aggregator-Instanz erstellt. Wie dem Code in Abbildung 4.10 zu entnehmen, geschieht dies indem die Software durch den Adressraum des OPC UA Server iteriert. Begonnen beim obersten Knoten, dem sogenannten *Object*-Knoten, tastet sich die Software hierbei entlang der hierarchischen Referenzen der Knotenstruktur systematisch voran, bis alle Knoten des Adressraumes einmal erfasst worden sind.

```
123     async def getAggrEndpoints(self, node:Node):
124         """
125         Interate through Namespace and detect AggregationPoints
126         """
127         try:
128             typeDef = self.get_node((await node.read_type_definition()))
129             typeDefname = (await typeDef.read_browse_name()).Name
130             if ( typeDefname == "AggregationEndpointType"):
131                 self.AggrPoints.append(node)
132                 _logger.info(f"Typedefinition: {typeDefname} found for Node: {node.nodeid}")
133         except:
134             _logger.info(f"No TypeDefinition found for Node: {node.nodeid}")
135 
136         self._Tracker.append(node)
137         children = await node.get_children(refs=33)
138 
139         for child in children:
140             if child not in self._Tracker:
141                 await self.getAggrEndpoints(node=child)
```

Abbildung 4.10: Lokalisieren der AggregationEndpoins durch Iteration

Zur Umsetzung dieser Iteration in Python, wird der erste Knoten zunächst durch seine statisch festgelegte NodeID vorgegeben. Diese NodeID basiert hierbei auf dem Basis-NodeSet der OPC Foundation und ist demnach für jeden OPC UA Server gleich (siehe 2.3). Im Anschluss kann die Methode *getAggrEndpoints()* der AggregationServer-Klasse zur Iteration auf Basis dieses Start-Knoten ausgeführt werden. Die Methode überprüft hierbei zunächst, ob der Knoten vom AggregationEndpointType abstammt, indem sie die Typdefinition des Knoten abfragt und vergleicht (Abb. 4.10, Zeile 128-134). Sollte hierbei eine Übereinstimmung festgestellt werden, so wird der Knoten einem Array zugeordnet, welches abschließend alle AggregationEndpoints enthalten wird (Abb. 4.10, Zeile 131). Im Anschluss an die Abfrage des Objekttypen, werden die hierarchisch mit diesem Knoten verknüpften Knoten durch die Methode *get_children()* der AggregationClient-Klasse in Zeile 137 identifiziert. Diese werden hierbei als Array aus Objekten der Klasse *Node* vpn der Methode zurückgegeben und bilden später die Grundlage für einen erneuten Durchlauf der Methode *getAggrEndpoints()*, in welchem auch sie darauf überprüft werden, ob sie vom AggregationEndpointtype abstammen 4.10, Zeile 139-141). Auf diese Weise wird die

Methode *getAggrEndpoints()* schlussendlich für jeden Knoten im Adressraum des aggregierenden OPC UA Server durchgeführt. Da die Spezifikationen von OPC UA es nicht verbieten Schleifen in den Knotenstrukturen im Adressraum durch Referenzen zwischen Knoten zu erstellen, besteht die Gefahr das es bei dieser Iteration zu einem ungewollten unendlichen Durchlauf kommt. Die Methode würde dann die gleichen Knoten unendlich oft abfragen. Hierzu ist es unabdingbar, die bereits geprüften Knoten separat zu listen. Jeder geprüfte Knoten wird so in Zeile 136 einem Array zugeordnet. Um die Methode nun explizit nur für neue Knoten auszuführen, wird in Zeile 140 geprüft, ob einer der hierarchisch verknüpften Knoten bereits im Array gespeichert worden ist.

Sind alle AggregationEndpoints identifiziert, kann für jede AggregationEndpoint-Instanz im Array aus Zeile 131 der Abbildung 4.10 eine Aggregator-Instanz erstellt werden. Dies wird hierbei, wie in Abbildung 4.11 ersichtlich, durch eine *For*-Schleife realisiert. Sie iteriert hierbei durch das Array und erstellt die Aggregator-Instanzen. Zudem initialisiert die Methode hierbei jedes Aggregator-Objekt, indem sie die Methode *_init_Aggregator()* der Aggregator-Klasse aufruft. Des weiteren wird jedem Aggregator-Objekt hierbei das jeweilige AggregationEndpoint-Objekt und ein Integer als Indikator übergeben. Dieser Indikator, sogt später dafür, dass die jeweilige Aggregator-Instanz einfacher zu identifizieren ist.

```
118         # Prepare Aggregators for everey AggregationEndpoint
119         for Endpoint in self.AggrEndpoints:
120             ident = self.AggrEndpoints.index(Endpoint)
121             await self._init_Aggregator(Ident=ident, AggrEndpoint=Endpoint)
```

Abbildung 4.11: Instanziierung einer Aggregator-Instanz pro AggregationEndpoint-Instanz

Wie Abbildung 4.8 zu entnehmen ist, verknüpft die Aggregator-Instanz bei der Initialisierung die OPC UA Methode *AggregateNodeStructure* mit einer Methode im Code. Die Verknüpfung geschieht hierbei, indem das Aggregator-Objekt den Knoten der Methode zunächst anhand des AggregationEndpoint-Knoten durch eine hierarchische Referenz identifiziert und anschließend durch eine Methode der Bibliothek AsyncUA mit der Methode *start_aggregation()* der Aggregator-Klasse verbindet. Wenn der Server später gestartet ist und ein Client die Methode AggregateNodeStructure im Adressraum des OPC UA Server ausführt, wird die entsprechend verknüpfte Methode *start_aggregation()* der jeweiligen Aggregator-Instanz durchlaufen. Zum Abschluss der Phase zwei folgt nun der Server-Start.

Nachdem der Server gestartet wurde und somit für Client-Zugriffe zur Verfügung steht, wartet die Implementierung auf den Aufruf der Methode AggregateNodeStructure. Wird diese aufgerufen, so beginnt Phase drei des Phasenkonzeptes indem die jeweilige Aggregator-Instanz die Konfiguration zur Aggregation einliest. Hierbei werden die, in Kapitel 4.4, beschriebenen Parameter aus den Variablen EndpointDescription, NamespaceUri und NodeID aufgenommen und gespeichert. Im Anschluss daran, initialisiert die Aggregator-Instanz ein Objekt der AggregationClient-Klasse.

Wie in Abbildung 4.12 dargestellt, können hierbei mehrere Aggregationen zur selben Zeit gestartet und durchgeführt werden. Durch die objektorientierte Programmierung der Implementierung erstellt jedes Aggregator-Objekt hierbei sein eigenes AggregationClient-Objekt und übergibt die für diese Aggregation spezifischen Parameter. Darauf aufbauend stellt das AggregationClient-Objekt die Verbindung zum Quellserver her. Wie im Code in Abbildung 4.13 zu sehen, geschieht dies durch den Aufruf der beiden

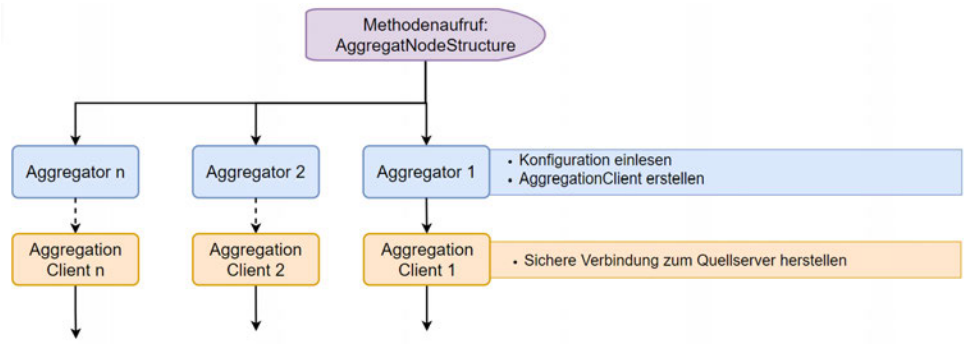


Abbildung 4.12: Verbindungsaufbau und Start der Aggregation

init-Methoden der AggregationClient-Klasse. Die Methode *__init__()* ist hierbei die sogenannte *Constructor*-Methode der Klasse. Sie wird jedes mal automatisch aufgerufen, sobald ein Objekt der Klasse AggregationClient erstellt wird. Wie zu erkennen wird diese Methode jedoch sequenziell ausgeführt, wodurch asynchrone Befehle hier nicht definiert werden dürfen. Aufgrund dessen wird die asynchrone Methode *init()* zusätzlich benötigt, um die Verbindung zum Quellserver herstellen zu können.

```
16      def __init__(self, url: str, timeout: int = 4):
17          super().__init__(url, timeout=timeout)
18          self.AggrObjects = {}
19          self.AggrObjecttypes = {}
20          self.NamespaceArray = None
21          self.ServerName = None
22
23      async def init(self):
24          try:
25              await self.connect()
26              self.NamespaceArray = await self.get_namespace_array()
27          except:
28              _logger.warn(f"No Connection established to Server: {self.server_url}")
29              raise ConnectionError
```

Abbildung 4.13: Initialisierung des AggregationClient-Objekts

Wie der Zeile 17 in Abbildung 4.13 zu entnehmen, wird bei der Constructor-Methode die sogenannte *Super-Klasse* initialisiert. Die Super-Klasse stellt hierbei stets die Klasse dar, von welcher die Kind-Klasse durch die Vererbung abstammt. Im Fall der AggregationClient-Klasse ist dies die *Client*-Klasse der Bibliothek AsyncIO (siehe Abb. 4.6). Sie muss bei Initialisierung des AggregationClient-Objektes ebenfalls initialisiert werden. Ihr wird hierbei zudem die URL des Quellserver übergeben, um den Verbindungsaufbau anschließend realisieren zu können. Da die Aggregation nur bei bestehender Verbindung zum Quellserver durchgeführt werden kann, wird der Versuch den Quellserver erstmalig zu erreichen mithilfe einer sogenannten Ausnahmebehandlung durchgeführt. Wie den Zeilen 24 bis 28 der asynchronen Methode *init()* zu entnehmen, wird die Ausnahmebehandlung durch die Schlüsselwörter *try* und *except* eingeleitet. Innerhalb dieser Umgebung versucht die AggergationClient-Instanz in Zeile 25 der Abbildung 4.13 eine Verbindung zum Quellserver aufzubauen. Gelingt dies nicht, weil z.B der Quellserver nicht am selben Netzwerk angeschlossen ist, wird eine Ausnahme (engl. *exception*) generiert. Diese Ausnahme kann anschließend durch die Ausnahmebehandlung aufgegriffen und für weitere Befehle genutzt werden. In diesem Fall, wird bei fehlender Verbindung zum Quellserver ein Verbindungsfeh-

ler (Zeile 29) ausgegeben, welcher die Aggregation abbricht und den Nutzer darüber informiert (Zeile 28).

Konnte die Verbindung zum Quellserver jedoch erfolgreich hergestellt werden, kann mit der Aggregation begonnen werden. Um den Server während der Aggregation nicht zu blockieren, wird hierfür, das in Kapitel 4.2 bereits beschriebene, Konzept der asynchronen Tasks verwendet (siehe Abb. 4.14). Dem Event-Loop wird somit eine Task übergeben, auf dessen Ausführung nicht gewartet werden muss. Die Schleife zur Aggregation generischer Knotenstrukturen läuft somit nahezu vollkommen getrennt vom eigentlichen Server, jedoch mit der Bedingung, dass kein weiterer Thread hierfür genutzt werden muss.

```
72          # Create Aggregation Task
73          await asyncio.create_task(self._aggregate_nodestruct())
```

Abbildung 4.14: Erstellung der Task zur asynchronen Aggregation

Wie dem Code in Abbildung 4.14 zu entnehmen wird bei Erstellung einer Task lediglich die auszuführende Methode übergeben. Die Anweisung darf hierbei nicht mithilfe des Schlüsselwortes *await* gekennzeichnet werden, da andernfalls an dieser Stelle auf die Ausführung der Methode gewartet wird.

Bevor mit der Aggregation der Knotenstrukturen nun begonnen werden kann, müssen zunächst die Namensräume des Quellserver aggregiert werden (Namensräume des OPC UA Adressraumes, siehe Abb. 2.4). Da hierbei genau genommen nur die Namensräume hinzugefügt werden müssten, welche Knoten enthalten die später aggregiert werden sollen, müsste jeder aggregierte Knoten auf seinen Namensraum hin untersucht werden. Um einen Knoten jedoch nach der in Kapitel 4.3 vorgestellten Methodik zu aggregieren, muss der relevante Namensraum bereits auf dem aggregierenden Server existieren. Um den Entwicklungsaufwand an dieser Stelle bewusst zu beschränken, werden mit dieser Implementierung zunächst alle Namensräume aggregiert. Hierfür liest das AggregationClient-Objekt das NamespaceArray des Quellserver, welches alle Namensräume eines OPC UA Server auflistet, aus und übergibt die Informationen an die Aggregator-Instanz. Im Anschluss kann dieser die Informationen mit dem NamespaceArray des aggregierenden Server vergleichen und die fehlenden Namensräume ergänzen. Um aggregierte Namensräume später besser identifizieren zu können, wird diesen bei Erstellung im Adressraum des aggregierenden Server der Suffix “Aggregated“ hinzugefügt. Weiterführend werden die hinzugefügten Namensräume mit den ursprünglichen Namensräumen auf dem Quellserver durch ein Python-Dictionary verknüpft. Somit lässt sich später genau nachvollziehen, welcher Namensraum von welchem Quellserver übernommen wurde. Das Dictionary erlaubt hierbei eine Zuordnung zweier Objekte über das Prinzip von Schlüssel-Wert-Paarungen. Ist eines der beiden Objekte demnach bekannt, kann das jeweils andere durch das Dictionary ausgegeben werden.

Sind die Namensräume abgeglichen, startet die Implementierung mit der Aggregation der Knotenstrukturen. Wie in Kapitel 2.3 bereits beschrieben, wird jedes Objekt im OPC UA Adressraum von einem Objekttypen abgeleitet. Ebenso verhält es sich mit Variablen und Referenzen. Um diese Strukturen neben den instantiierten Knoten ebenfalls im Adressraum des aggregierenden Server abzubilden und bei der Implementierung möglichst effizient vorzugehen, wurde der Aggregator-Klasse eine Methode hinzugefügt, die bei der Aggregation einzelner Knoten zwischen deren Knotentypen unterscheiden kann. Je

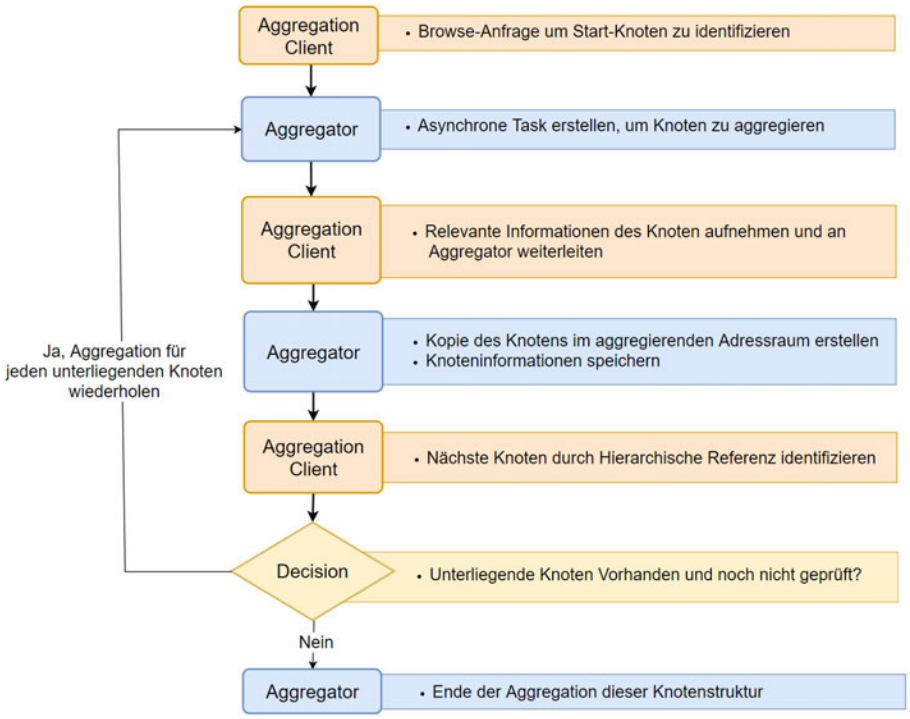


Abbildung 4.15: Prozessschleife zur Aggregation generischer Knotenstrukturen

nach Typ, kann die Implementierung einen angepassten Prozess zur Aggregation des Knoten durchlaufen. Wie in Abbildung 4.15 darstellt, wird dies anhand einer Schleife gelöst. Ähnlich dem dem Schema aus Abbildung 4.10, iteriert die Software hierbei entlang der hierarchischen Referenzen durch den Adressraum des Quellserver. Je nachdem welcher Start-Knoten hierbei gewählt wird, werden zunächst die Typdefinitionen und anschließend die Instanzen aggregiert (siehe Anhang A.4).

Die Schleife beginnt hierbei indem das AggregationClient-Objekt den Start-Knoten auf dem Quellserver identifiziert. Für die Typdefinitionen ist dies der Knoten mit der Bezeichnung *Types*, welcher eine, durch das Basis-NodeSet der OPC Foundation vorgegebene, statische NodeID besitzt. Von ihm ausgehend, sind alle Typdefinitionen im Adressraum eines OPC UA Server hierarchisch referenziert. Für die Aggregation der spezifischen Objekte, wird die Konfiguration der Aggregation im Adressraum des aggregierenden Server hinzugezogen, welche zu Beginn in Form eines Informationsmodelles importiert oder durch Client-Zugriffe angepasst wurde (siehe Kapitel 4.3 und 4.4). Anhand der dort definierten NodeID des jeweiligen AggregationEndpoints wird der Start-Knoten zur Aggregation der Instanzen auf dem Quellserver lokalisiert. Ist der Start-Knoten gefunden, kann die Iteration beginnen.

Um einen Knoten im Adressraum des aggregierenden Server erstellen zu können, muss dieser direkt von Beginn an einem bereits existierenden Knoten durch eine Referenz zugeordnet werden. Bei der Aggregation eines Knoten aus dem Adressraum des Quellserver, muss daher zuvor ein entsprechend übergeordneter Knoten auf dem aggregierenden Server bekannt sein. Beim Start der Aggregation wird hierfür der jeweilige AggregationEndpoint gewählt. Um die Knoten im Code eindeutig zu benennen, wird hierbei auf die Nomenklatur aus Kapitel 2.4 zurückgegriffen.

Wie in Abbildung 4.15 zu sehen, wird nachdem der Start-Knoten identifiziert worden ist, eine asynchrone Task zu Aggregation des Knoten gestartet. Im Anschluss daran, liest die AggregationClient-Instanz die relevanten Informationen (Attribute, Referenzen, Werte etc.) des Knoten auf dem Quellserver und gibt diese gebündelt an das Aggregator-Objekt weiter. Hierfür erstellt die AggregationClient-Instanz ein Objekt der AggregationNode-Klasse. Dieses dient hierbei als Speicher der Informationen zur Aggregation des Knoten. Die AggregationClient-Instanz führt hierbei eine Reihe von Abfragen zum betreffenden Knoten im Adressraum des Quellserver durch und speichert die erhaltenen Informationen in den Attributen des AggregationNode-Objektes (siehe Anhang A.3). Zu den relevanten Informationen gehören hierbei unter anderem die Attribute (Beschreibung, Name etc.) und Referenzen des Knoten. Hat die Aggretor-Instanz das AggregationNode-Objekt erhalten, beginnt diese die Informationen des Objektes zu nutzen, um eine Kopie des ursprünglichen Knotens im Adressraum des aggregierenden Server zu erstellen. Hierbei wird zunächst durch eine Abfrage überprüft, ob der Knoten bereits auf dem aggregierenden Server vorhanden ist. Ist dies nicht der Fall, wir die Methode *_aggregate_node()* der Aggregator-Klasse aufgerufen, welche im Anhang A.5 ganzheitlich abgebildet ist. Die Methode differenziert zwischen den jeweiligen Knotentypen und weist das AggregationServer-Objekt an, den entsprechenden Knoten an der entsprechenden Stelle zu erstellen.

```
283     async def _transform_nodeId(self, nodeId:NodeId):
284         """
285         Create unique NodeId to aggregate Node to Server Adressspace
286         Current Transformation-rule: NodeIdentifier(String) = AggregatorIdent:Identifier
287         return NodeId
288         """
289 
290         NewIndex = self.NamespaceMap[nodeId.NamespaceIndex]
291         NewIdent = f"{self.Ident}:{nodeId.Identifier}"
292         NewNodeId = NodeId(Identifier=NewIdent,
293                            NamespaceIndex=NewIndex)
294         return NewNodeId
```

Abbildung 4.16: Transformation der NodeID

Da bei der Erstellung der Kopie durch die Methode *_aggregate_node()* auch die NodeID festgelegt werden muss, kann es hier zu Konflikten auf dem aggregierenden Server kommen, da die ursprüngliche NodeID des Knoten auf dem Quellserver hier bereits vergeben sein könnte. Aufgrund dessen, dass nach Spezifikation Part 3 der OPC Foundation, jede NodeID einzigartig im Adressraum eines OPC UA Server sein muss, findet hier die Methode *_transform_nodeId()* Anwendung (siehe Abb. 4.16). Diese generiert auf Basis der ursprünglichen NodeID eine neue, einzigartige NodeID und nutzt hierfür das Prinzip einer statischen Regel zur Transformation. Wie in Abbildung 4.16 zu erkennen, wird der Identifier der NodeID hierbei durch einen String ersetzt, welcher sich aus dem Indikator des jeweiligen Aggregator-Objektes und dem ursprünglichen Identifier zusammensetzt.

Ist die Kopie im Adressraum des aggregierenden Server erstellt, wird diese zunächst durch ein Python-Dictionary mit dem ursprünglichen Knoten des Quellserver verknüpft. So kann später nachvollzogen werden, wo der jeweilige Knoten auf dem Quellserver zu finden ist. Im Anschluss ermittelt die Instanz des AggregationClient die hierarchisch verknüpften Knoten des ursprünglichen Knotens auf dem Quellserver, da für diese nun ebenfalls die Aggregation zu starten ist (Abb. 4.17, Zeile 207). Da es auch hier,

ähnlich wie bei der Methode aus 4.10, zu unendlichen Schleifen beim iterieren kommen kann, müssen auch hier alle, bereits aggregierten Knoten, gelistet werden.

```
209             # Track already visited Nodes to avoid looping
210             self.Tracker.append(NodeRH)
211
212             # Call for all hierarchical of aggregated Node
213             if(ChildrenRH):
214                 for ChildRH in ChildrenRH:
215                     if ChildRH not in self.Tracker and NewNodeLH is not None:
216                         asyncio.create_task(self._browse_nodestruct(
217                             ParentLH=NewNodeLH, NodeRH=ChildRH))
218                     else:
219                         return
220             else:
221                 return
222         else:
223             print("Iteration Failure or End reached")
```

Abbildung 4.17: Iteration zur Aggregation hierarchisch verknüpfter Knoten

Wie Zeile 210 aus Abbildung 4.17 zu entnehmen, geschieht dies indem die Knoten einem Array hinzugefügt werden. In Zeile 215 wird dann die Aggregation ausschließlich für noch nicht im Array gespeicherte Knoten gestartet, indem eine *IF*-Abfrage durchgeführt wird. Zudem wird in Zeile 213 geprüft, ob das AggregationClient-Objekt hierarchisch verknüpfte Knoten finden konnte. Sollte dies nicht der Fall sein, endet die Aggregation dieses Zweiges der Knotenstruktur an dieser Stelle.

Sind jedoch hierarchisch verknüpfte Knoten gefunden worden, wird die Aggregation jedes Knoten mithilfe einer For-Schleife gestartet. Auch hierbei wird eine neue asynchrone Task erstellt, um die Aggregation mehrerer Knoten nahezu parallel durchführen zu können. Um die Strukturen des Quellserver im Adressraum des aggregierenden Server exakt gleich abbilden zu können, ist hierbei jedoch darauf zu achten, dass der gerade erstellte Knoten als neuer linksseitiger Knoten für die nächste Aggregation übergeben wird (siehe Abb. 4.17, Zeile 216/217). Nur so kann die hierarchische Struktur der ursprünglichen Knoten übernommen werden. Sind alle Schleifen zur Aggregation durchlaufen und die relevanten Knotenstrukturen, Attribute und Referenzen erstellt, gilt Phase drei des Phasenkonzeptes aus Abbildung 4.5 als abgeschlossen.

In der letzten Phase, “Run“, wird nun die Handhabung von OPC UA Service-Anfragen thematisiert. Nachdem die Knotenstrukturen aggregiert worden, befindet sich der Server in einer Art Leerlauf, in welchem er auf OPC UA Client-Zugriffe oder eine erneute Aggregation (Phase drei) wartet. Beginnt ein solcher Zugriff, bspw. in Form einer OPC UA Service-Anfrage für eine Subsciption bestimmter Knoten, wird auf dem aggregierenden Server ein Event ausgelöst. Als Folge des Event, wird eine Methode der Klasse AggregationServer aufgerufen, welche die Anfrage bearbeiten soll. Damit dieser Prozess durchlaufen werden kann, muss jedoch bereits in Phase zwei, der Initialisierung des Server, ein sogenanntes *Callback* eingerichtet werden. Als Callback wird hierbei grundlegend jede Funktion bezeichnet, welche eine weitere Funktion als Parameter zugewiesen bekommt. Wie in Darstellung 4.18 zu sehen, bietet die Bibliothek AsyncIO zu Handhabung von Service-Events eine dedizierte Callback-Methode an.

```
167     def _create_callbacks(self):
168         self.subscribe_server_callback(CallbackType.ItemSubscriptionCreated, self.create_monitored_items)
169         self.subscribe_server_callback(CallbackType.ItemSubscriptionDeleted, self.delete_monitored_items)
```

Abbildung 4.18: Erstellung der Server-Callbacks

Hierfür wird der Methode *subscribe_server_callback()* zunächst die Art des Events übergeben, um anschließend die Funktion zu definieren, welche ausgeführt werden soll, wenn das Event eintritt. Im Fall der Subscription-Anfrage wäre dies das Event *ItemSubscriptionCreated*, welches die Funktion *create_monitored_items()* auslöst. Sollte die Subscription wieder beendet werden, so folgt die Methode *delete_monitored_items()*.

Um eine Subsciption-Anfrage auf aggregierte Knoten umsetzen zu können, ist eine entsprechende Anfrage auf dem Quellserver zu platzieren. Hierzu wird das jeweilige Aggregator-Objekt bzw. AggregationClient-Objekt benötigt, welches den Knoten zuvor aggregiert hat. Wie in Abbildung 4.19 zu sehen, identifiziert die AggregationServer-Instanz, nachdem das Event ausgelöst wurde, zunächst das relevante Aggregator-Objekt (siehe Anhang A.6, Zeile 180-186). Hierzu prüft sie jedes Aggregator-Objekt welches bei den Aggregationen erstellt wurde daraufhin, ob dieses für die Aggregation des relevanten Knoten verantwortlich war. Dies geschieht, indem die AggregationServer-Instanz durch die Python-Dictionaries der Aggregator-Objekte iteriert und nach dem Knoten sucht. Hat sie ihn gefunden, kann die AggregationServer-Instanz eine asynchrone Task erstellen, um die Subsciption auf dem Quellserver einzuleiten.

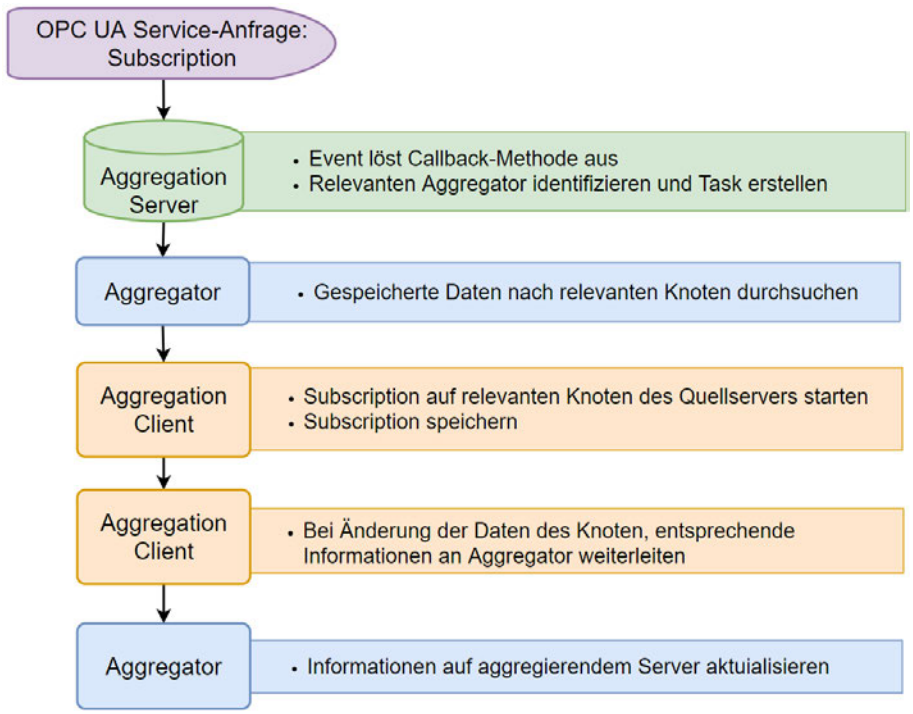


Abbildung 4.19: Handhabung von OPC UA Service Anfragen am Beispiel einer Subscription

Um die Subscription auf dem Quellserver zu erstellen, wird innerhalb der Methode *create_aggr_subscription()* der Klasse AggregationServer zunächst eine Instanz der *SubscriptionHandler*-Klasse erstellt (Abb. 4.20, Zeile 190). Diese Klasse beinhaltet hierbei Methoden zur Handhabung der Events bei Änderung von Daten eines Knoten. Im Fall der Aggregation generischer Knotenstrukturen bedeutet dies, dass ein SubscriptionHandler-Objekt dafür zuständig ist, geänderte Daten vom Quellserver an den aggregierenden Server weiterzuleiten, damit dieser die Informationen in seinem Adressraum aktualisieren kann. Im Anschluss an die Erstellung des SubscriptionHandler-Objektes wird in Zeile 192 der Abbildung 4.20, die Subscription auf dem Quellserver erstellt. Hierzu wird die Methode *create_subscription()* des AggregationClient-Objektes ausgeführt und ein Objekt der Klasse *Subscription* aus der Bibliothek AsyncUA instantiiert. Der Methode wird hierbei das SubscriptionHandler-Objekt, sowie eine statische Periode übergeben. Die Periode definiert hierbei das Abtastintervall, in welchem das SubscriptionHandler-Objekt auf aktualisierte Informationen im Adressraum des Quellserver prüfen soll. Um die Subsciption abschließend zu starten und die zu überwachenden Knoten des Quellserver anzugeben, wird in Zeile 194 die Methode *subscribe_data_change()* der Subscription-Klasse ausgeführt.

```
188     async def create_aggr_subscription(self, period, Aggregator:Aggregator, NodeRH:Node, ItemId):
189         # Create SubscriptionHandler
190         handler = SubricptionHandler(server=self, NodeRH=NodeRH, ItemId=ItemId)
191         # Create Subscription
192         sub = await Aggregator.AggrClient.create_subscription(period=period, handler=handler)
193         print(f"Subscribe Datachange for Node {NodeRH.nodeid}")
194         await sub.subscribe_data_change(nodes=NodeRH)
195         #Wait till end of subscription
196         while ItemId in self.SubscriptedNodes:
197             print("Sleep 1")
198             await asyncio.sleep(1)
199         await sub.delete()
```

Abbildung 4.20: Erstellung der Server-Callbacks

Ist die Subscription gestartet, begibt sich die Methode *create_aggr_subscription()* in eine *While*-Schleife, in welcher sie verweilt, bis die Subscription des jeweiligen Knoten auf dem aggregierenden Server gelöscht wird (Abb. 4.20, Zeile 196-198). Erst dann wird auch die Subscription auf dem Quellserver beendet. Der Auslöser zum beenden der Subscriptions ist hierbei das, in Abbildung 4.18 bereits dargestellte, Event *ItemSubscriptionDeleted*. Dieses wird erzeugt, nachdem der Client die Service-Anfrage auf dem aggregierenden Server löscht. Bei einer Vielzahl von gleichzeitigen Subscriptions muss jedoch sichergestellt werden, dass die richtige Subscription, passend zum jeweiligen Knoten, beendet wird. Dies geschieht, indem eine sogenannte *ItemId* infolge des Events *ItemSubscriptionCreated* zur Liste *SubscriptedNodes* hinzugefügt wird. Die ItemId wird hierbei für jeden Knoten einzigartig vergeben. Wenn nun die Subscription des Knoten im aggregierenden Namensraum beendet wird, wird die jeweilige ItemId aus der Liste entfernt. Hierdurch trifft das Kriterium für die While-Schleife aus Zeile 196 nicht mehr zu und die Subscription zum Knoten auf dem Quellserver wird beendet (Abb. 4.20, Zeile 199).

5 Evaluation und Ausblick

In diesem Kapitel wird die zuvor beschriebene Implementierung der Aggregation generischer OPC UA Server anhand der in Kapitel 4.1 definierten Anforderungen an die Software evaluiert. Des weiteren wird die Aggregation am Use Case MFlex2025 getestet und ein Ausblick über mögliche Erweiterungen oder Verbesserungen der Lösung thematisiert.

Wie dem Product-Backlog zu entnehmen, soll die Software grundlegend die Knotenstrukturen anderer OPC UA Server aggregieren (Anhang Tabelle A.1, Anforderung 2). Hierzu muss zunächst durch Implementierung sichergestellt werden, das entsprechender OPC UA Server zur Abbildung der aggregierten Knotenstrukturen für den Client-Zugriff zur Verfügung steht (Anforderung 1). Hierfür wurde die Klasse AggregationServer, welche im Klassenkonzept in Abbildung 4.6 dargestellt ist, von der Server-Klasse der Bibliothek AsyncUA abgeleitet. Wie auch die Eltern-Klasse beinhaltet demnach auch die AggregationServer-Klasse die notwendigen Funktionalitäten um einen konformen OPC UA Server bereitzustellen. Um die Aggregation prozessspezifisch konfigurieren zu können, wurde weiterführend ein Konzept zur Konfiguration des aggregierenden OPC UA Server entwickelt. Das Konzept beinhaltet hierbei die Konfiguration durch ein Informationsmodell, welches AggregationEndpoints definiert und notwendige Parameter zur Aggregation definiert. Diese zeitsparende Variante der Konfiguration bietet dem Anwender hierbei zudem auch die Möglichkeit, Änderungen nach Start des OPC UA Server direkt im Adressraum durch einen Client-Zugriff durchzuführen. Dies sorgt für maximale Flexibilität bei der Konfiguration der Software und vermindert den Arbeitsaufwand um ein Vielfaches (Anforderung 6). Aufbauend auf der Konfiguration der Anwendung, wurde die Softwareimplementierung in Python mithilfe der Bibliothek AsyncUA, welche auf der Grundkomponente AsyncIO basiert, ausgearbeitet (Anforderung 9). Die Ausarbeitung wurde hierbei in Kapitel 4 detailliert beschrieben und erläutert. Des weiteren wurde der erstellte Code zum besseren Verständig stellenweise kommentiert (Anforderung 10).

Um die Anwendung zu testen wurde die Anlage des Use Case MFlex2025, welche in Kapitel 3.2 beschrieben ist, am Institut der Fraunhofer Gesellschaft in Stade genutzt. Hierbei wurden die OPC UA Server des MabiMax100, der BVE, sowie der Spindel SLQ100 mit der Implementierung aggregiert. Um die Aggregation auf die Anlage abzustimmen, wurde zunächst das in Abbildung 5.1 dargestellte Informationsmodell mit den OPC UA Modelling-Editor SiOME von Siemens erstellt. Das NodeSet basiert hierbei auf dem Aggregation-NodeSet, welches den AggregationEndpointType definiert. Wie der Abbildung zu entnehmen wurde zunächst ein übergeordneter Knoten, mit der Bezeichnung “MabiDrillingUnit“ erstellt. Dieser ist vom *MotionDeviceSystemType* abgeleitet, welcher in der Robotics-Spezifikation der OPC Foundation definiert wurde [OPC19].

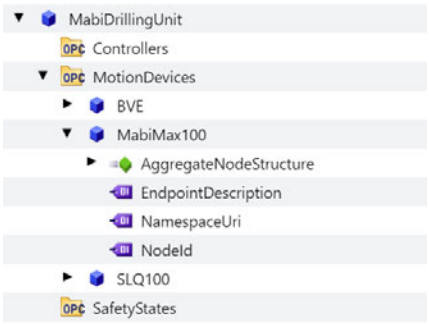


Abbildung 5.1: Informationsmodell zur Aggregation im Rahmen von MFlex2025

Das MotionDeviceSystem beinhaltet hierbei Knoten des *FolderType*, die das System in Komponenten zur Bewegung und Steuerung, sowie Sicherheitsfunktionalitäten unterteilen. Den Bewegungskomponenten werden hierbei die drei zu aggregierenden Feldgeräte zugewiesen. Da jedes Gerät einen generischen OPC UA Server besitzt, dessen Knotenstrukturen hier abgebildet werden sollen, werden an dieser Stelle Instanzen des AggregationEndpointType für jedes Gerät definiert. Nachdem die notwendigen Parameter ausgefüllt wurden, kann das NodeSet exportiert und für die Anwendung bereitgestellt werden.

Um die Software nun zur Nutzung vorzubereiten, wird lediglich ein Python-Skript benötigt. Wie Abbildung 5.2 zu entnehmen, wird hier zunächst in Zeile 13 eine Instanz der AggregationServer-Klasse erstellt. Im Anschluss werden die relevanten Parameter des Server in den Attributen des AggregationServer-Objektesin definiert (Zeilen 16 bis 19). Wie zu sehen zählen hierzu der Name des OPC UA Server, sowie dessen erster Namensraum und den Endpoint unter welchem er später für Client erreichbar ist. Weiterführend wurde die NodeSet-Datei mit dem zuvor erstellten Informationsmodell in den Zeilen 22 und 23 lokalisiert. Dies geschieht hierbei mithilfe der *OS*-Systemklasse, welche standartmäßig in Python integriert ist und hilfreiche Methoden zur Definition eines Dateipfades im System bereitstellt. Der Pfad wird im Anschluss in Zeile 26 an das Server-Objekt übergeben, wobei hier die *init()*-Methode der AggregationServer-Klasse aufgerufen wird. Wie bereits in Kapitel 4.5 beschrieben, sorgt diese Methode im Anschluss für den import der relevanten Informationsmodelle und bereitet die Aggregation vor.

```
12  # Create AggregationServer
13  server = AggregationServer()
14
15  # Configure Server Parameter
16  server.ServerName = "MabiDrillingUnit"
17  server.Endpoint = "opc.tcp://127.0.0.1:4800"
18  server.NamespaceUri = "http://ifam.fraunhofer.de/UA/DI/Aggregation/MFLEX"
19  server.SecurityPolicy = "None"
20
21  # Lokalize NodeSet of Aggregation-Configuration
22  thisFile = os.path.dirname(os.path.abspath(__file__))
23  MflexNodeSet = os.path.join(thisFile,"Ifam.Fraunhofer.MFLEX_Aggregation.NodeSet2.MabiSqlBve.xml")
24
25  # Init Server
26  await server.init(MflexNodeSet)
```

Abbildung 5.2: Nutzung der Implementierung zur Erstellung eines aggregierenden OPC UA Server

Nachdem der Server gestartet wurde, konnte ein Client-Zugriff mithilfe eines OPC UA Client erfolgreich durchgeführt werden. Weiterführend wurde im Anschluss die Aggregation der drei Feldgeräte, durch den Aufruf der AggregateNodeStructure-Methoden, angestoßen. Die Aggregation der Knotenstrukturen verlief hierbei problemlos und erbrachte das in Abbildung A.7 dargestellte Resultat.

Im Anschluss an die Aggregation wurden die Knotenstrukturen zunächst auf Vollständigkeit und Konformität gegenüber der Spezifikationen überprüft. Zudem konnte hier festgestellt werden, dass die Namensräume aller Quellserver erfolgreich aggregiert und die hinzugefügten Knoten den Namensräumen korrekt zugeordnet werden konnten (Anforderung 4). Ebenfalls wurden alle Typdefinitionen von Objekten der Quellserver vollständig im Adressraum des aggregierenden Server abgebildet (Anforderung 2). Um daraufhin die Implementierung der Handhabung von OPC UA Service-Anfragen zu testen, wurde eine Subscription auf bereits aggregierte Knoten erstellt. Die Daten der jeweiligen Knoten wurden hierbei zyklisch aktualisiert und konnten vom Client erfolgreich gelesen werden. Das hierbei verwendete Abtastintervall von 500 Aktualisierungen pro Minute wurde dahingehend als vollkommen ausreichend eingestuft (Anforderungen 3 und 5).

Um die Performance der Software zu testen, wurde die Anzahl der im Fall von MFlex2025 aggregierten Knoten mithilfe eines Python-Skriptes ermittelt. Diese beläuft sich hierbei auf rund 430 Knoten. Des weiteren wurde die Dauer des Aggregationsprozesses für diesen Fall ebenfalls aufgenommen. Auf Basis dieser beiden Werte konnte eine gemittelte Dauer von 0,01 Sekunden pro Knoten ermittelt werden. Da es sich beim Use Case MFlex2025 um ein Forschungsprojekt des Fraunhofer IFAM handelt, soll hier zwar vorrangig die Machbarkeit der Anwendung beurteilt werden, jedoch ist an dieser Stelle festzustellen, dass die kurze Zeitperiode des gesamten Aggregationsprozesses für eine deutlich höher Performance der Anwendung spricht, als erwartet wurde (Anforderung 11). Der Sicherheit im Bezug auf die Kommunikation zwischen Client und Server kommt im Fall von MFlex2025 ebenfalls nahezu keine Bedeutung zuteil, da die Anlage lediglich innerhalb von Testläufen intern genutzt wird. Auf Basis dessen und durch die zeitliche Begrenzung der Ausarbeitung, wurde hier auf eine gesonderte Implementierung von verschlüsselten Verbindungen verzichtet.

Wie der Evaluation, sowie den Tests der mobilen Bohrroboter-Anlage zu entnehmen, funktioniert die Aggregation generischer Knotenstrukturen durch den hier entwickelten, aggregierenden OPC UA Server bereits sehr gut. Dennoch sind eine Vielzahl von weiteren Funktionalitäten des Kommunikationsprotokolls OPC UA künftig zu ergänzen. Hier ist bspw. die Handhabung von OPC UA Service-Anfragen zu nennen. Da bisher lediglich Anfragen in Form von Subscriptions unterstützt werden, gibt es hier ein großes Potenzial die Implementierung durch weitere Service-Typen zu ergänzen. So könnten bspw. Methoden-Aufrufe oder DataAccess-Anfragen durch eine ähnliche Implementierung, möglicherweise sogar unter Verwendung bereits erstellter Funktionen, anschließend hinzugefügt werden. Des weiteren könnte die Performance der Anwendung gesteigert werden, indem statt Python die Programmiersprache C verwendet wird (siehe Kapitel 4.2). Dies würde allerdings einen erheblichen Mehraufwand mit sich führen, da der bereits erstellte Code hierbei grundlegend neu programmiert werden müsste. Das Konzept der Aggregation könnte hierbei jedoch übernommen werden.

Weiterführend könnte die Implementierung zudem durch Mechanismen ergänzt werden, die eine Filtrierung oder Umrechnung der Daten einzelner Informationen bereitstellen. Dies wäre insbesondere dann von Vorteil, wenn die Software mehrere Feldgeräte bzw. Anlagen aggregiert, da hier oft gleichbedeutende Variablen verknüpft werden können. Hierzu kann zudem die OPC UA Spezifikation Part 13 der OPC Foundation genutzt werden. Sie definiert bereits sogenannte Aggregates, welche die Zusammenführung, Filtrierung und Umrechnung von Datenströmen im Adressraum des Server unterstützen. Kombiniert mit der, in dieser Ausarbeitung entwickelten, Aggregation generischer Knotenstrukturen könnte so eine höhere Informationsebene auf den aggregierenden OPC UA Server abgebildet werden.

Abschluss

Der aktuelle Trend der Industrie zeichnet sich insbesondere durch die Digitalisierung und Vernetzung, möglichst wandlungsfähiger Systeme zur intelligenten Produktion aus. Unternehmen der Industrie 4.0 müssen demnach zunehmend flexibel und modular aufgebaut sein. Oft bedarf dies einer Umstrukturierung von Arbeitsprozessen, sowie der Weiterentwicklung bereits bestehender Komponenten. Je modularer die Produktion hierbei gestaltet wird, desto mehr Schnittstellen zwischen Anlagen und Feldgeräten werden benötigt. Im Zuge der Digitalisierung wird die klassische Automatisierungspyramide daher ersetzt, durch ein Netzwerk aus vertikal und horizontal verbundenen Systemen. Dies sorgt zumeist ebenfalls für eine grundlegenden Umstrukturierung der IT hinter den Anlagensystemen. Flexible Kommunikationsprotokolle, wie OPC UA, werden hierbei vermehrt zur vereinheitlichten Kommunikation eingesetzt. Durch die stetig wachsende Anzahl an integrierten Komponenten, steigt die Auslastung gewöhnlicher Netzwerke der Fertigung jedoch enorm an. Als Folge daraus kann es zu hohen Wartungsintervallen und im schlechtesten Fall auch zum Versagen der Netzwerke kommen.

Das in dieser Ausarbeitung entwickelte und implementierte Konzept zur Aggregation generischer Knotenstrukturen diverser OPC UA Server schafft hierbei Abhilfe. Durch die Einführung zentraler Integrationsplattformen, können so Netzwerkverbindungen zur Kommunikation zwischen Systemen der Produktion ressourcenschonend eingesetzt werden. Am Beispiel vom Projekt MFlex2025 des Fraunhofer IFAM wurde so die Implementierung einer solchen Integrationsplattform, in Form eines aggregierenden OPC UA Server, umgesetzt. Mithilfe asynchroner Prozesse und auf Basis bereits bestehender Lösungsansätze konnten konzeptionelle Abläufe zur Verknüpfung mehrerer OPC UA Server realisiert und das Ziel einer minimalen Anzahl von Client-Server-Verbindungen zur Kommunikation erreicht werden.

Literaturverzeichnis

[Bil18] Meik Billman. Welche kriterien müssen industrie-4.0-produkte 2019 erfüllen? https://www.zvei.org/fileadmin/user_upload/Presse_und_Medien/Publikationen/2018/November/Welche_Kriterien_muessen_I-4.0-Produkte_erfuellen_2019/ZVEI__LF_Welche_Kriterien_muessen_I-4.0-Produkte_2019.pdf, 2018. Besucht: 11-01-2022.

[BMB16] BMBF. Industrie 4.0. https://www.bmbf.de/bmbf/de/forschung/digitale-wirtschaft-und-gesellschaft/industrie-4-0/industrie-4-0.html, 2016. Besucht: 10-01-2022.

[BMB21] BMBF. Hightech-strategie 2025. https://www.bmbf.de/bmbf/de/forschung/hightech-strategie-2025/hightech-strategie-2025_node.html, 2021. Besucht: 10-01-2022.

[BMW20] BMWK. Luftfahrtforschungsprogramm (lufo). https://www.bmwi.de/Redaktion/DE/Artikel/Technologie/luftfahrttechnologien-02.html, 2020. Besucht: 15-03-2022.

[GBB+14] Daniel Grosmann, Markus Bregulla, Suprateek Banerjee, Dirk Schulz, and Roland Braun. Opc ua server aggregation — the foundation for an internet of portals. Published in: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), 2014.

[H. 16] H. Greinke im Auftrag des BMWK. *Bekanntmachung zur Förderung von Forschungs- und Technologievorhaben im Rahmen des fünften nationalen zivilen Luftfahrtforschungsprogramms Dritter Programmaufruf*. Bundenanzeiger, 2016.

[Hei22] Andreas Heine. opcua-asyncio. https://github.com/FreeOpcUa/opcua-asyncio, 2022. LGPL-3.0 Lizenz.

[Hop17] Stefan Hoppe. There is no industrie 4.0 without opc ua. https://opcconnect.opcfoundation.org/2017/06/there-is-no-industrie-4-0-without-opc-ua/, 2017. Besucht: 11-01-2022.

[Jam07] James Robertson, Suzanne Robertson. *Volmere - Requirements Specification Template*. Atlantic Systems Guild Limited, 2007.

[Kos15] Gunther Koschnick. Das referenzarchitekturmodell rami 4.0 und die industrie 4.0-komponente. https://www.zvei.org/themen/industrie-40/das-referenzarchitekturmodell-rami-40-und-die-industrie-40-komponente, 2015. Besucht: 11-01-2022.

[OPC16] Opc 10000-3: Opc unified architecture - part 3: Adress space model. https://reference.opcfoundation.org/v105/Core/docs/Part3/#5, 2016.

[OPC19] Opc unified architecture for robotics. https://opcfoundation.org/developer-tools/specifications-opc-ua-information-models/opc-unified-architecture-for-robotics/, 2019.

[OPC21] Opc 10000-4: Opc unified architecture - part 4: Services. https://reference.opcfoundation.org/v104/Core/docs/Part4/#5, 2021.

[Pet15] Dr. Thieß Petersen. *Globalisierung, Digitalisierung und Einkommensungleichheit*. 2015. Bertelsmann Stiftung, ISSN: 2191-2459.

[PH09] Arnd Poetzsch-Heffter. *Konzepte objektorientierter Programmierung: Mit einer Einführung in Java (eXamen.press)*. Springer, 2009.

[Pla18a] Plattform Industrie 4.0. Das referenzarchitekturmodell industrie 4.0 (rami 4.0). https://www.plattform-i40.de/IP/Redaktion/EN/Downloads/Publikation/rami40-an-introduction.pdf?__blob=publicationFile&v=7, 2018. Besucht: 11-01-2022.

[Pla18b] Plattform Industrie 4.0. Hierarchie in der industrie 3.0. https://www.plattform-i40.de/IP/Redaktion/DE/Infografiken/hierarchie-in-der-industrie-3-0.html, 2018. Besucht: 24-02-2022.

[Pla18c] Plattform Industrie 4.0. Hierarchie in der industrie 4.0. https://www.plattform-i40.de/IP/Redaktion/EN/Infographics/hierarchy-in-industrie-4-0.html, 2018. Besucht: 24-02-2022.

[Pla20] Plattform Industrie 4.0. *Verwaltungsschale in der Praxis*. Bundesministerium für Wirtschaft und Klimaschutz (BMWK), 2020. Publiziertes Diskussionspapier.

[Poe18] Dr.-Ing. Harald Poetter. Cyber physical systems. https://www.izm.fraunhofer.de/de/trendthemen/cyber-physical-systems.html, 2018. Besucht: 10-01-2022.

[RS16] Armin Roth and David Siepmann. *Einführung und Umsetzung von Industrie 4.0: Grundlagen, Vorgehensmodell und use cases Aus der praxis*. Springer Gabler, 2016. ISBN 978-3-662-48504-0.

[SMA+18] Miriam Schleipen, Henning Mersch, Jouni Aro, Heikki Tahvanainen, Daniel Pagnozzi, Uslaender Thomas, Julius Pfrommer, Robert Henßen, Nadia Scandelli, Jan Bajorat, and et al. *Praxishandbuch OPC UA Grundlagen - Implementierung - Nachruestung - Praxisbeispiele*. Vogel Business Media, 1 edition, 2018.

[STE+16] Ilkka Seilonen, Tomi Tuovinen, Joona Elovaara, Ian Tuomi, and Timo Oksanen. *Aggregating OPC UA servers for monitoring manufacturing systems and mobile work machines*. 2016.

A Anhang

Tabelle A.1: Lastenheft auf Basis der Anforderungsanalyse nach Volere [Jam07]

Anforderungsanalyse und Lastenheft nach Volere-Template						
Id	Anforderungstyp	Beschreibung	Begründung	Abnahmekriterium	Konflikte/Machbarkeit	Priorität (1-10)
1	9. Anforderungen an Funktionen und Daten des Produktes	Die Software soll einen OPC UA Server für Client-Zugriffe bereitstellen	Um aggregierte Informationen bereitstellen zu können, soll die Software einen OPC UA-Server beinhalten, welcher einen Client-Zugriff ermöglicht	Test der Erreichbarkeit innerhalb des Netzwerkes durch einen Client-Zugriff	--	10
2	9. Anforderungen an Funktionen und Daten des Produktes	Der Server soll Knotenstrukturen aggregierter OPC UA-Server abbilden	Zur Realisierung einer OPC UA Server Aggregation müss der Aggregationsserver Knotenstrukturen der Quellserver adaptieren und Abbilden können	Die Konfigurierten Knotenstrukturen sind auf dem Server durch einen Browse-Request verfügbar	--	9
3	9. Anforderungen an Funktionen und Daten des Produktes	Die Software soll die Abfrage von Daten aggregierter OPC UA-Server ermöglichen	Um aggregierte Knotenstrukturen funktional nutzbar zu machen, sollen Attribute und Daten aggregierter Knoten auf dem Aggregationsserver verfügbar gemacht werden	Abfrage der Daten mittels OPC UA Service-Anfrage	--	8
4	4. Vorgegebene Randbedingungen für das Projekt	Der Server soll konform zu den Spezifikationen der OPC Foundation aufgebaut sein	Zur reibungslosen Implementierung in ein Netzwerk nach den Standarts der I4.0 und zur Kommunikation mit anderen OPC UA-Servern muss der aggregierende Server die Standarts der OPC Foundation strikt beachten	Aufbau des Adressraumes untersuchen	--	10
5	9. Anforderungen an Funktionen und Daten des Produktes	Die Software soll die Bereitstellung von OPC UA-Services sicherstellen	Um Client-Anfrgen für OPC UA-Services zu realisieren, muss die Software diese weiterleiten und Rückmeldungen auf dem aggregierenden OPC UA Server bereitstellen	Test der Services durch Client-Anfragen	--	7
6	11. Benutzbarkeitsanforderungen	Die Konfiguration der Software soll für den Nutzer mit geringem Aufwand verbunden sein	Zu nutzerfreundlichen Anwendung der Software soll die Konfiguration der zu aggregierenden Server und Knotenstrukturen so zeit- und kosteneffizient wie möglich gestaltet sein	Vergleich der Lösung mit anderen Konzepten	--	7
7	9. Anforderungen an Funktionen und Daten des Produktes	Namensräume aggregierter OPC UA-Server sollen durch die Software bereitgestellt werden	Zur Zuordnung aggregierter Knotenstrukturen müssen die ürsprünglichen Namensräume bei der Aggregation berücksichtig werden	Test durch Client-Zugriff	--	9
8	4. Vorgegebene Randbedingungen für das Projekt	Der Arbeitsumfang, sowie die Dauer der Entwicklung soll begrenzt sein	Die Bearbeitungszeit ist begrenzt auf 3 Monate. Der Arbeitsumfang soll durch die Analyse und Prüfung relevanter Anforderungen abgesteckt werden	Anforderungsanalyse und fristgerechte Abgabe der Ausarbeitung	--	10
9	13. Operationale und Umfeldanforderungen	Das Design der Software soll nach Gesichtspunkten der objektorientierten Programmierung aufgebaut sein	Durch die variable Anzahl an Servern, Namensräumen und Knoten, sowie zur leichten Pflege und Modifizierung der Software, soll eine objektorientierte Programmiersprache verwendet werden	Klassen- und Methodenkonzept, sowie Polymorphie und Terminologie darstellen	--	8
10	11. Benutzbarkeitsanforderungen	Die Softwareimplementierung soll kommentiert und dokumentiert werden	Durch Kommentare und Beschreibungen im Code und der Ausarbeitung, können einzelne Prozesse und Strukturen künftig einfacher interpretiert und identifiziert werden	Beschreibung von Klassen und Methoden im Code, sowie die schriftliche Ausarbeitung	--	7
11	12. Sicherheitsanforderungen	Der Server soll die Sicherheitsstandarts aggregierter Server übernehmen	Zum sicheren Informationsaustausch soll der Server die von den aggregierten Servern verwendeten Verschlüsslungsmethoden und Nutzerbeschränkungen adaptieren	Client-Zugriff über verschlüsselte Verbindung	--	2
12	12. Performance Anforderungen	Die Aggregation soll performant und automatisch durchgeführt werden	Um eine effiziente Verwendung der Software gewährleisten zu können, muss die Aggregation vollautomatisch durchgeführt werden und soll hierbei möglichst wenig Zeit in Anspruch nehmen	Automatische Aggregation der relevanten Informationen (Kann durch Auslöser angestoßen werden). Dauer des Aggregationsprozesses pro Knoten messen	Performance-Verbesserungen oft sehr zeitintensiv. Machbarkeit daher beschränkt durch Bearbeitungsdauer	3

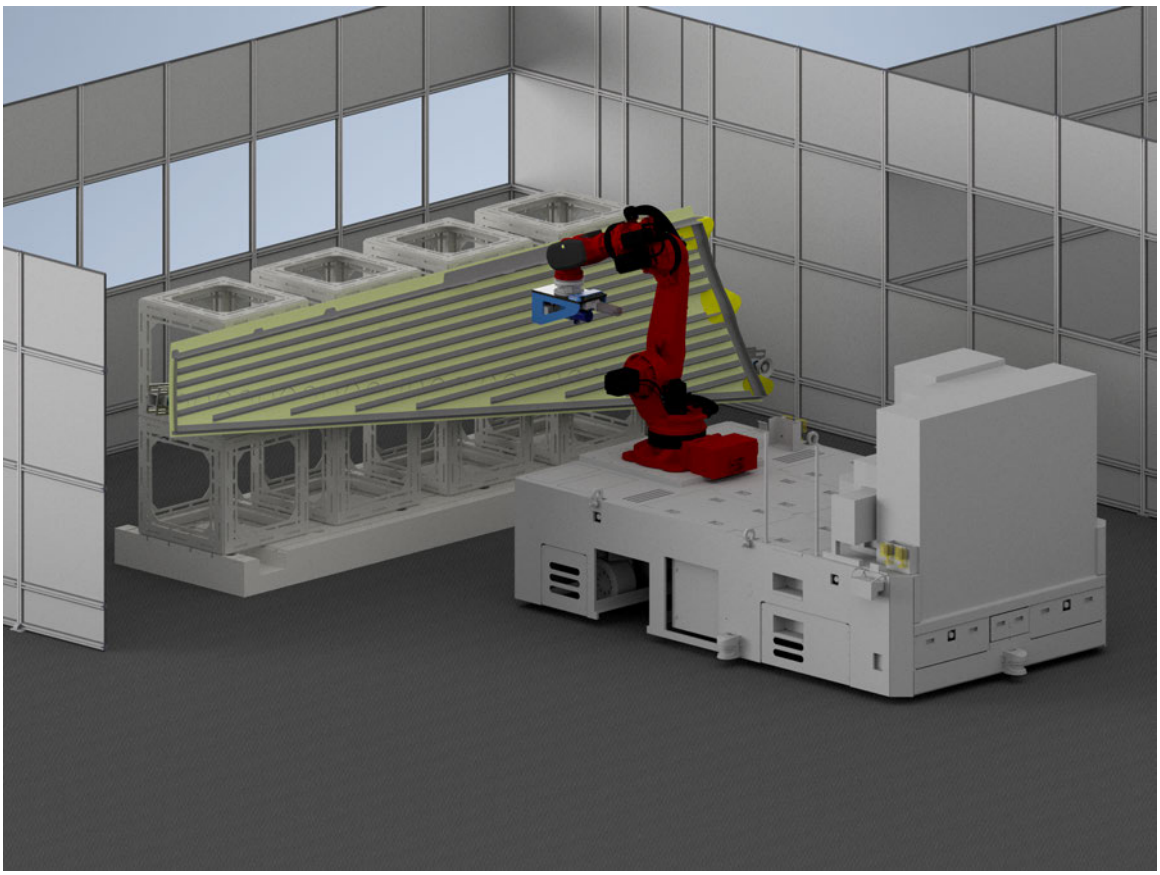


Abbildung A.1: Rendering des physischen Demonstrators

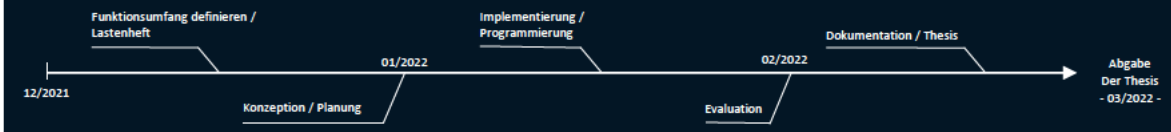
Fraunhofer IFAM

Vertikale Integration in der Industrie 4.0 mithilfe einer service-orientierten Informationsarchitektur

Open **P**lattform **C**ommunications **U**nified **A**rchitecture

Ziel der Wissenschaftlichen Ausarbeitung:

Ziel der Forschungsarbeit ist eine Untersuchung, inwiefern die automatische Aggregation mehrerer Feldgeräte mit generischen OPC UA Servern am Beispiel eines prozessspezifischen Bohrrobotersystems zu realisieren ist



OPC UA

OPC UA ist eine standardisierte, service-orientierte Informationsarchitektur, welche einen hersteller- und plattformunabhängigen, sicheren Datenaustausch einzelner Systeme gewährleisten soll. Der Standard wurde von der OPC Foundation im Jahr 2006 als Nachfolger des bereits etablierten OPC Standards eingeführt und gilt bereits jetzt, unter anderem im Rahmen des Zukunftsprojektes „Industrie 4.0" des Bundesministeriums für Bildung und Forschung, als wegweisende Kommunikationsarchitektur im Hinblick auf die Digitalisierung der industriellen Produktion.

Durch eine stetig erweiterte Multipart-Spezifikation soll OPC UA somit künftig dafür sorgen, dass Produktionssysteme unterschiedlichster Funktions- und Informationsebenen eine generische Kommunikationsschnittstelle bieten und somit möglichst schnell in bestehende Produktionsstrukturen integriert werden können. Dies sorgt für ein hohes Maß an Modularität und steigert die Wandlungsfähigkeit von Produktionsanalgen

Abb. 1 :Automatisierungspyramide

Aggregation

Um eine entsprechend großflächige, vertikale Integration unterschiedlichster Systeme innerhalb eines Produktions-standortes zu erreichen und dabei ressourcenschonende Netzwerke zu nutzen, ist es notwendig eine Vielzahl von separaten Client-Server-Verbindungen zu den OPC UA Servern einzelner System herzustellen. Dies sorgt für einen stetig ansteigenden Datenverkehr, Unübersichtlichkeit und hohe Lizenzgebühren.

Abb. 2 : Servernetzwerk, aufgeteilt nach Informationsebenen

Ein Aggregationsserver kann hierbei Datenströme, unterschiedlicher Subsysteme aggregieren und an einem *Serverendpoint* für den Client-Zugriff bereitstellen. Der Server sorgt hierbei für die Verbindung zu den Quellservern und koordiniert zudem die Handhabung von OPC UA Service-Anfrage. Hierdurch kann die Anzahl der benötigten Verbindungen auf ein Minimum reduziert und die Auslastung des Netzwerks niedrig gehalten werden. Des weiteren kann der Aggregationsserver Funktionalitäten enthalten, welche bspw. das Filtern oder Zusammenfassen von Daten erlauben, um diese im Anschluss in SCADA oder MES –Systemen zur Produktionssteuerung, -optimierung und –überwachung zu nutzen.

MFlex2025

Das Fraunhofer Institut für Fertigungstechnik und angewandte Materialforschung (IFAM) in Stade befasst sich im Projekt „MFlex2025" mit der Erstellung eines flexiblen Robotersystems, zur Bearbeitung von Flugzeugkomponenten. So soll bei diesem Projekt einen Industrieroboter, welcher in Kombination mit einer verfahrbaren Plattform einen deutlich vergrößerten Arbeitsraum besitzt, Bohrungen an einem Seitenleitwerk einbringen. Hierbei liegt der Fokus unter anderem auf dem Zusammenspiel der einzelnen Systeme (Roboter, mobile Plattform, Bohrvorschubeinheit, Navigationssysteme etc.), welche mittels OPC UA, standardisiert miteinander kommunizieren sollen. Die einzelnen Subsysteme werden hierbei in Kooperation mit unterschiedlichen Herstellern der Automatisierungsbranche wie bspw. FFT, 3D.aero und Airbus gestaltet und abschließend als vollwertiges, mobiles Bohrsystem zusammengefügt und getestet.

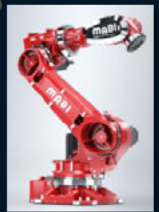
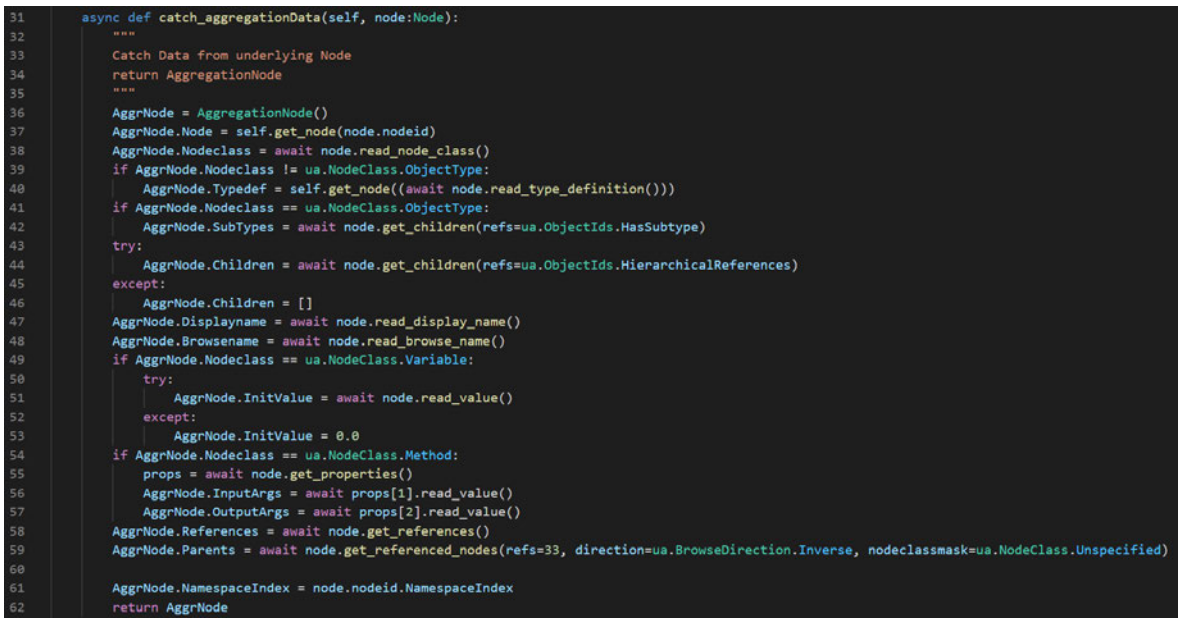


Abb. 3 : MabiMax100 der Firma Mabi

Autor: Nico Töpfer | 03/2022 | Exposé im Rahmen der Bachelor-Thesis

1) https://www.tuvsud.com/en/digital-service/industry-40-enablement-programme
2) https://automotive-nordwest.de/member/fraunhofer-ifam-e-v/
3) https://opcfoundation.org/about/opc-technologies/opc-ua/
4) https://www.mabi-robotic.com/robotic/overview/

Abbildung A.2: Expose im Rahmen der Bachelorarbeit



```python
async def catch_aggregationData(self, node:Node):
    """
    Catch Data from underlying Node
    return AggregationNode
    """
    AggrNode = AggregationNode()
    AggrNode.Node = self.get_node(node.nodeid)
    AggrNode.Nodeclass = await node.read_node_class()
    if AggrNode.Nodeclass != ua.NodeClass.ObjectType:
        AggrNode.Typedef = self.get_node((await node.read_type_definition()))
    if AggrNode.Nodeclass == ua.NodeClass.ObjectType:
        AggrNode.SubTypes = await node.get_children(refs=ua.ObjectIds.HasSubtype)
    try:
        AggrNode.Children = await node.get_children(refs=ua.ObjectIds.HierarchicalReferences)
    except:
        AggrNode.Children = []
    AggrNode.Displayname = await node.read_display_name()
    AggrNode.Browsename = await node.read_browse_name()
    if AggrNode.Nodeclass == ua.NodeClass.Variable:
        try:
            AggrNode.InitValue = await node.read_value()
        except:
            AggrNode.InitValue = 0.0
    if AggrNode.Nodeclass == ua.NodeClass.Method:
        props = await node.get_properties()
        AggrNode.InputArgs = await props[1].read_value()
        AggrNode.OutputArgs = await props[2].read_value()
    AggrNode.References = await node.get_references()
    AggrNode.Parents = await node.get_referenced_nodes(refs=33, direction=ua.BrowseDirection.Inverse, nodeclassmask=ua.NodeClass.Unspecified)

    AggrNode.NamespaceIndex = node.nodeid.NamespaceIndex
    return AggrNode
```

Abbildung A.3: Lesen und speichern relevanter Informationen des zu aggregierenden Knoten

```python
async def _browse_nodestruct(self, ParentLH: Node, NodeRH: Node):
    """
    Browse underlying servers Nodestructure
    RH : Node on underlying Server
    LH : Node on Aggregating Server
    """

    if NodeRH and (ParentLH != NodeRH or ParentLH != 0):
        try:
            AggrNodeRH = await self.AggrClient.catch_aggregationData(node=NodeRH)

        except:
            await self.except_AggrNodeDataMissing(node=NodeRH)

        if (await NodeRH.read_node_class()) == ua.NodeClass.ObjectType:
            # Create Objecttype
            try:
                if (await self._check_node_existence(ObjTypeNodeRH=AggrNodeRH)) == False:
                    NewNodeLH = await self._aggregate_node(ParentNode=ParentLH, AggrNode=AggrNodeRH)
                else:
                    node_id = NodeId(Identifier=AggrNodeRH.Node.nodeid.Identifier, NamespaceIndex=self.NamespaceMap[AggrNodeRH.NamespaceIndex])
                    NewNodeLH = self.AggrServer.get_node(node_id)
            except:
                await self.except_FailedToAggregateNode(node=NodeRH)

        else:
            # Create Object, Var or Method
            try:
                if (await self._check_node_existence(ObjTypeNodeRH=AggrNodeRH)) == False:
                    NewNodeLH = await self._aggregate_node(ParentNode=ParentLH.nodeid, AggrNode=AggrNodeRH)
                else:
                    node_id = NodeId(Identifier=AggrNodeRH.Node.nodeid.Identifier, NamespaceIndex=self.NamespaceMap[AggrNodeRH.NamespaceIndex])
                    NewNodeLH = self.AggrServer.get_node(node_id)

            except:
                NewNodeLH = None
                await self.except_FailedToAggregateNode(node=NodeRH)

        # Catch hierarchical childre-nodes
        ChildrenRH = AggrNodeRH.Children

        # Track already visited Nodes to avoid looping
        self.Tracker.append(NodeRH)

        # Call for all children of aggregated Node
        if(ChildrenRH):
            for ChildRH in ChildrenRH:
                if ChildRH not in self.Tracker and NewNodeLH is not None:
                    asyncio.create_task(self._browse_nodestruct(
                        ParentLH=NewNodeLH, NodeRH=ChildRH))
                else:
                    return
        else:
            return
    else:
        print("Iteration Failure or End reached")
```

Abbildung A.4: Iteration zu Aggregation generischer Knotenstrukturen

```python
    async def _aggregate_node(self, ParentNodeLH: NodeId, AggrNode: AggregationNode):
        """
        Add Node to Aggregating-Server by NodeClass
        Args:
            - ParentNode (LH)
            - AggrNode: AggregationNode
        """
        Parent = self.AggrServer.get_node(ParentNodeLH)

        # Create unique NodeId
        NewNodeId = await self._transform_nodeId(nodeId=AggrNode.Node.nodeid)
        if(AggrNode.Nodeclass == ua.NodeClass.Object):
            # Create new Object
            new_node = await Parent.add_object(nodeid=NewNodeId, bname=AggrNode.Browsename,
            instantiate_optional=False, objecttype=ua.ObjectIds.BaseObjectType)

            # Add to dict
            self.AggrPoint.AggregatedNodes[AggrNode] = new_node
            print("New ObjectNode created")

        elif(AggrNode.Nodeclass == ua.NodeClass.Variable):
            # Create new Variable
            new_node = await Parent.add_variable(nodeid=NewNodeId, bname=AggrNode.Browsename, val=AggrNode.InitValue)
            await new_node.set_writable(True)

            # Add to dict
            self.AggrPoint.AggregatedNodes[AggrNode] = new_node
            print("New Variable created")

        elif(AggrNode.Nodeclass == ua.NodeClass.Method):
            # Create new Method
            new_node = await Parent.add_method(NewNodeId, AggrNode.Browsename, self._method_call)
            print("New Method created")
            return new_node #Cause no references needed

        elif(AggrNode.Nodeclass == ua.NodeClass.ObjectType):
            new_node = await Parent.add_object_type(NewNodeId, AggrNode.Browsename)
        else:
            print("No new Node created")
            return False

        # Add References
        if AggrNode.Nodeclass != ua.NodeClass.ObjectType:
            await self._add_references(NewNode=new_node, AggrNode=AggrNode)
        AggrNode.Aggregator = self

        # Add Node to Dict
        self.NodeMap[new_node.nodeid]=AggrNode.Node.nodeid
        return new_node
```

Abbildung A.5: Aggregation eines Knotens

```python
    def _create_callbacks(self):
        self.subscribe_server_callback(CallbackType.ItemSubscriptionCreated, self.create_monitored_items)
        self.subscribe_server_callback(CallbackType.ItemSubscriptionDeleted, self.delete_monitored_items)

    async def create_monitored_items(self, event, dispatcher):
        for idx in range(len(event.response_params)):
            if event.response_params[idx].StatusCode.is_good():
                NodeToMonitor = event.request_params.ItemsToCreate[idx].ItemToMonitor.NodeId

                monitored_item_id = event.request_params.SubscriptionId
                self.SubscriptedNodes[monitored_item_id] = NodeToMonitor

                #Cause of unique NodeId, only one sub is created per node
                for Aggr in self.Aggregator:
                    NodeRH = await Aggr.get_NodeRH(NodeLH=NodeToMonitor)
                    if NodeRH != None:
                        print("Start create_aggr_subscription Task")
                        asyncio.create_task(self.create_aggr_subscription(500,Aggregator=Aggr, NodeRH=NodeRH, ItemId=monitored_item_id))
                    else:
                        print("Node is not an aggregated Node or Node on underlying Server not found. No Node Subscription started")

    async def create_aggr_subscription(self, period, Aggregator:Aggregator, NodeRH:Node, ItemId):
        print("Create SubHandler")
        handler = SubricptionHandler(server=self, NodeRH=NodeRH, ItemId=ItemId)
        print("Create Subscription")
        sub = await Aggregator.AggrClient.create_subscription(period=period, handler=handler)
        print(f"Subscribe Datachange for Node {NodeRH.nodeid}")
        await sub.subscribe_data_change(nodes=NodeRH)
        #Wait til end of subscription
        while ItemId in self.SubscriptedNodes:
            print("Sleep 1")
            await asyncio.sleep(1)
        await sub.delete()
```

Abbildung A.6: Handhabung einer OPC UA Service-Anfrage in Form einer Subscription

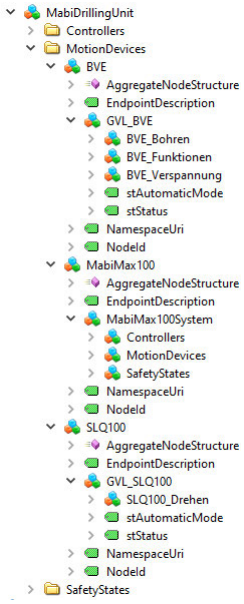


Abbildung A.7: Aggregation im Use Case MFlex2025

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Dieses Blatt, mit der folgenden Erklärung, ist nach Fertigstellung der Abschlussarbeit durch den Studierenden auszufüllen und jeweils mit Originalunterschrift als letztes Blatt in das Prüfungsexemplar der Abschlussarbeit einzubinden.
Eine unrichtig abgegebene Erklärung kann -auch nachträglich- zur Ungültigkeit des Studienabschlusses führen.

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

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

- die folgende Aussage ist bei Gruppenarbeiten auszufüllen und entfällt bei Einzelarbeiten -

Die Kennzeichnung der von mir erstellten und verantworteten Teile der ist erfolgt durch:

[redacted]

_____ _____ _____
Ort Datum Unterschrift im Original