Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Entwicklung einer Edge-Lösung zur Virtualisierung von MQTT-basierten IoT-Sensoren

von

Nicolas Mehlei

dem Fachbereich IV – Wirtschaftswissenschaften II –
der Hochschule für Technik und Wirtschaft Berlin vorgelegte Masterarbeit
zur Erlangung des akademischen Grades
Master of Science (M.Sc.)
im Studiengang
Angewandte Informatik

Berlin, 21. September 2018

Prüfungskommission

Gutachter:	Prof. Dr. Alexander Huhn	Hochschule für Technik und Wirtschaft Berlin
	Prof. Dr.-Ing. Thomas Schwotzer	Hochschule für Technik und Wirtschaft Berlin

Danksagung

An dieser Stelle möchte ich mich bei allen Menschen bedanken, welche mich bei meiner Masterarbeit unterstützt haben.

Herrn Prof. Dr.-Ing. Thomas Schwotzer und Herrn Prof. Dr. Alexander Huhn danke ich für die Übernahme der Betreuung, sowie der sehr hilfreichen Beratung in einer Vielzahl von Treffen.

Dank gilt auch meinem Arbeitgeber, der Homesk GmbH, für deren Verständnis während der Bearbeitungszeit sowie meiner Familie und Freunden für das leidige Korrekturlesen.

Nicolas Mehlei

Berlin, im September 2018

Kurzbeschreibung

Die vorliegende Masterarbeit hat das Ziel, eine Lösung zu erarbeiten, welche in der Lage ist, miteinander kommunizierende „Internet of Things“-Geräte voneinander zu entkoppeln. Mittels dieser Entkoppelung ist es möglich, die Kommunikation der Geräte und somit deren Verwendung zu vereinfachen sowie Einfluss auf die ausgetauschten Daten zu haben. Diese Daten können hierbei aggregiert, projiziert als auch schlichtweg modifiziert werden, um anschließend als eigene Daten eines so genannten *Virtual Devices* auf einem MQTT-Broker veröffentlicht zu werden. Das hierdurch entstehende System ist somit in der Lage, IoT-Geräte zu virtualisieren und dementsprechend von den vorhandenen Sensoren und Aktoren zu abstrahieren. Der Fokus der zu erarbeitenden Lösung liegt hierbei auf der Erweiterbarkeit und Konfigurierbarkeit, um als Plattform für aufbauende Systeme dienen zu können.

Abstract

This master thesis aims to develop a solution which allows to decouple communicating „Internet of Things“-based devices. This decoupling allows to simplify device communication and usage as well as to influence the exchanged data. Exchanged data can then be aggregated, projected or simply modified to be published to a MQTT broker as data of a so-called *virtual device*. Effectively, this allows to virtualize IoT devices and thus abstract from existing sensors and actors. Focus of the solution lies in extensibility as well as configurability to allow the solution to potentially act as a platform for future projects.

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Motivation 1

1.2. Ziel der Arbeit 2

1.3. Gliederung der Arbeit 3

2. Grundlagen **5**

2.1. Messaging 5

2.1.1. Messaging Patterns 5

2.1.2. Zustellungsgarantien 7

2.2. Machine-to-Machine 8

2.3. Internet of Things 8

2.4. Edge Computing 8

2.5. MQTT 9

2.6. Strategy-Pattern 11

2.7. Regelmaschine 11

2.8. Skalierung 12

2.9. JarvisSuite 13

2.9.1. Umgebungssensor 14

2.9.2. Tür und Fenstersensor 15

2.9.3. LED-Strip-Controller 15

3. Analyse **16**

3.1. Funktionale Anforderungen 16

3.1.1. Werteaggregation 16

3.1.2. Werteprojektion 19

3.1.3. Kommando-Verarbeitung 21

3.2. Nichtfunktionale Anforderungen 23
3.2.1. Plattform 24
3.2.2. Zielgruppe 24
3.2.3. Konfigurierbarkeit 24
3.2.4. Erweiterbarkeit 24
3.2.5. Geräteunterstützung 25
3.2.6. Performanz 25
3.2.7. Skalierbarkeit 26
3.2.8. Verfügbarkeit 27
3.2.9. Datensicherheit 27

4. Lösungsansätze **29**
4.1. Regelmaschine 29
4.2. Strategy-Pattern 30
4.3. Gegenüberstellung 31

5. Entwurf **32**
5.1. Aufbau 32
5.2. Sensoren und Aktoren 33
5.2.1. Temperatursensor 33
5.3. Virtual Device 34
5.4. Virtual Value 34
5.4.1. Datenempfang 35
5.4.2. Wertegenerierung 40
5.4.3. Gruppierung 41
5.4.4. Generierung 42
5.4.5. Packaging 44
5.4.6. Veröffentlichung 45
5.5. Virtual Function 45
5.5.1. Aufrufziele 46
5.5.2. Aufrufgenerierung 46
5.5.3. Asynchrone Verarbeitung 48
5.6. Virtual Device Host 49
5.6.1. Konfiguration 49

5.7. Monitoring 50
5.7.1. Veröffentlichung 50
5.7.2. Kalkulation 52
5.7.3. Toleranzzeitraum 53
5.8. Kaskadierung 53
5.8.1. Beispiel 54

6. Implementierung **56**
6.1. Verwendeter MQTT-Broker 56
6.2. Integration als Hintergrunddienst 56
6.3. JSON-Format 57
6.4. Code-Styles 58
6.4.1. Formatierung 58
6.4.2. Design Patterns 59
6.4.3. Konventionen 59
6.5. Temperatursensor 60

7. Test **61**
7.1. Testaufbau 61
7.2. Funktionale Testszenarien 61
7.2.1. Szenario 1: Float-Aggregation 62
7.2.2. Szenario 2: Boolean-Aggregation 62
7.2.3. Szenario 3: Boolean-Aggregation (On-Demand) 62
7.2.4. Szenario 4: Verschachteltes JSON bzw. Hierarchie 63
7.2.5. Szenario 5: Kommando-Weiterleitung 63
7.2.6. Szenario 6: Kommando-Weiterleitung mit Ersetzung 63
7.2.7. Szenario 7: Kaskadierung 63
7.2.8. Szenario 8: Monitoring 64
7.3. Nicht-funktionale Testszenarien 64
7.3.1. Erweiterbarkeit 64
7.3.2. Verfügbarkeit 64

8. Ergebnisbewertung **66**
8.1. Funktionale Ergebnisbewertung 66

8.2. Nichtfunktionale Ergebnisbewertung 66
8.2.1. Konfigurierbarkeit 66
8.2.2. Erweiterbarkeit 66
8.2.3. Geräteunterstützung 67
8.2.4. Skalierbarkeit 67
8.2.5. Verfügbarkeit 67
8.3. Auswertung 68

9. Zusammenfassung **69**

10. Ausblick **70**
10.1. Auto-Konfiguration durch Integration von Ontologien 70
10.2. Live-Konfiguration 71
10.3. Persistenz 72
10.3.1. Ansatz 72

Literaturverzeichnis **74**

Abbildungsverzeichnis **78**

Tabellenverzeichnis **79**

Abkürzungen **80**

Glossar **81**

A. Anhang **82**
A.1. Inhalt der beiliegenden CD 82

1. Einleitung

1.1. Motivation

Seit dem Beginn der IoT-Ära wurden über die Jahre immer mehr IoT-Geräte verfügbar, wodurch inzwischen eine breit gefächerte Landschaft an vielfältig einsetzbaren IoT-Geräten existiert. Ein Gerät kann hierbei bspw. ein Sensor oder ein Aktor sein. Ein Sensor zeichnet sich durch die Werte aus, die er zur Verfügung stellt, wohingegen ein Aktor Kommandos empfangen und anhand derer agieren kann. Selbstverständlich sind auch jedwede Mischformen daraus möglich.

Ein Nachteil an dieser großen Vielfalt an Geräten ist die Vielzahl an unterschiedlichen Formaten und Protokollen, mittels derer eine Kommunikation erfolgen kann, sowie eine große Menge an zueinander inkompatiblen Standards. Dadurch wird die Kommunikation mit solchen Geräten unnötig erschwert, da der Aufwand des Zugriffs mit der Menge der unterstützten Geräte steigt.

Des Weiteren gibt es durch die verteilte Natur einer solch vernetzten Umgebung inhärente Problematiken, welche ein andere Geräte verwendendes Gerät behandeln muss. Ein Beispiel hierfür ist die Ausfallsicherheit: Je mehr IoT-Geräte verwendet werden, desto höher ist die Chance, dass eine verwendete Ressource temporär oder dauerhaft nicht zur Verfügung steht und somit die Funktionalität der zugreifenden Komponente oder des zugreifenden Geräts eingeschränkt sein kann.

Viele dieser Probleme bzw. deren Komplexität sind begründet in einer engen Bindung zwischen den spezifischen Geräten. Wie solch ein Zugriff schematisch aussehen würde bei einem Gerät (hier: Consumer), welches auf vier Geräte zugreift, ist in Abbildung 1.1 zu sehen.

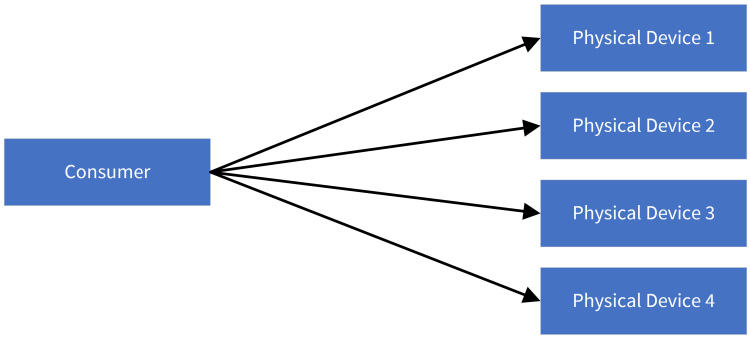

Abbildung 1.1.: Klassischer Zugriff

1.2. Ziel der Arbeit

Allgemein formuliert bieten IoT-Geräte bis zu zwei Arten an Schnittstellen: Werte und Kommandos. Wo ein Sensor hierbei meist eher Werte anbietet, wird ein Aktor meist auf den Empfang von Kommandos horchen. Eine Möglichkeit, diese enge Bindung zwischen Geräten zu entkoppeln, ist, die Punkt-zu-Punkt-Verbindung zwischen ihnen zu unterbrechen und die Kommunikation stattdessen über eine Zwischenschicht zu realisieren.

Der Vorteil solch einer Zwischenschicht ist die Möglichkeit zur Anpassung der Dateninhalte, welche zwischen den Geräten ausgetauscht werden. Der Nutzen solcher Anpassungen ist sehr vom Anwendungsfall abhängig, aber bspw. könnten mehrere Sensoren oder mehrere Aktoren logisch zu jeweils einem Gerät zusammengefasst werden. Die Verwendung eines solchen einzelnen zusammengefassten Gerätes ist einfacher als die Verwendung mehrerer einzelner Geräte. Des Weiteren ist es möglich, basierend auf den aggregierten Daten zusätzliche synthetische Werte bereitzustellen, bspw. Durchschnitts- oder interpolierte Werte.

> Unter dem Begriff *Virtualisierung* wird in der Informatik grundsätzlich eine Abbildung logischer Ressourcen auf physische Ressourcen (Hardware) verstanden. Soweit wie möglich soll ein Benutzer logischer Ressourcen keinen Unterschied zu den darunter liegenden physischen Ressourcen feststellen können. [23, S. 17]

Zitat 1.1.: Definition von Virtualisierung

Basierend auf der Definition in Zitat 1.1 und der zuvor genannten Beschreibung kann solch eine Zwischenschicht somit eine Virtualisierung eines Gerätes (bzw. Devices) genannt werden, also ein *Virtual Device*. Wie dies den Zugriff verändern würde ist in Abbildung 1.2 dargestellt.

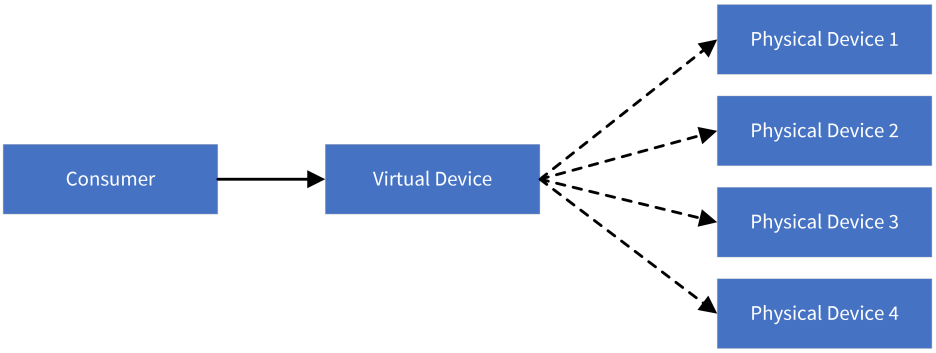

Abbildung 1.2.: Ansatz für entkoppelten Zugriff

Solch eine Virtualisierung von Geräten sollte möglichst keine spürbare Latenz mit sich bringen, daher sollte sich eine entsprechende Lösung nach Möglichkeit physisch so nah an den restlichen Geräten wie möglich befinden. Demnach bietet es sich an, eine Lösung im Rahmen des Edge Computings zu suchen, bei dem ein zentrales Ziel die Positionierung von berechnenden Komponenten in der direkten Umgebung von datengenerierenden Geräten ist. Diese Virtual Devices können sich somit in einem so genannten „Virtual Device Host“ befinden, welches sich im Edge Network befindet und sich dementsprechend sehr nahe an den Geräten befindet.

Diese Arbeit soll die Erarbeitung einer solchen Lösung darstellen und dokumentieren.

1.3. Gliederung der Arbeit

Diese Arbeit unterteilt sich in mehrere Kapitel. Zunächst werden im Kapitel *Grundlagen* Themen erläutert, welche von den nachfolgenden Kapiteln als theoretisches Wissen vorausgesetzt werden.

Im Rahmen der *Analyse* werden dann die funktionalen und nicht-funktionalen Anforderungen ermittelt, welche an die angestrebte Lösung gestellt werden.

In *Lösungsansätze* werden zwei Umsetzungsvarianten gegenübergestellt und abgewägt, welche hiervon besser für die Umsetzung geeignet ist. Anschließend wird dieser Ansatz im Rahmen des Kapitels *Entwurf* weiter ausgeführt und die zu entwickelnde Lösung konzipiert.

Das Kapitel *Implementierung* wird darauf eingehen, welche Herausforderungen sich während der Umsetzung ergaben und wie diese gelöst werden konnten. Diese Implementation wird im anschließenden Kapitel *Test* auf dessen korrekte Funktionalität getestet und in *Ergebnisbewertung* überprüft, ob die in *Analyse* aufgestellten Anforderungen von der Lösung erfüllt werden. Über diesen Ablauf und dessen Ergebnisse wird im Kapitel *Zusammenfassung* ein Resümee gezogen.

Abschließend gibt das Kapitel *Ausblick* einen Überblick über ausgewählte Thematiken, welche nicht mehr im Rahmen dieser Arbeit miteinbezogen werden konnten.

2. Grundlagen

Dieses Kapitel beschäftigt sich mit den theoretischen Grundlagen, welche zum Verständnis der in dieser Arbeit verwendeten Technologien erforderlich sind und dementsprechend für diese Arbeit vorausgesetzt werden.

2.1. Messaging

Messaging ist ein Integrationsschema, mit dem es möglich ist, mehrere Anwendungen oder Komponenten miteinander entkoppelt zu verbinden. Diese Kommunikation ist hierbei *Technologieagnostisch*, da die verwendete Technologie oder Programmiersprache zur Durchführung der Kommunikation irrelevant ist, so lange beide Seiten das Format der Nachricht verstehen können. [8, Kap. 2]

Die Kommunikation via Messaging ist hierbei inhärent asynchron, wodurch die Kommunikationsteilnehmer (also Sender und Empfänger) nicht zwingend zum gleichen Zeitpunkt aktiv, verbunden und funktionsfähig sein müssen. Des Weiteren beeinflussen sich die Kommunikationsteilnehmer nicht gegenseitig, falls einer von beiden Leistungsprobleme hat. [8, Kap. 2]

Ist mehr Kontrolle über die Zustellung der Nachrichten erforderlich, z.B. für eine Lastverteilung, so kann ein Nachrichtenrouter in den Nachrichtenfluss integriert werden. Hierdurch ist eine noch stärkere Entkoppelung möglich als es Messaging ohnehin bereits bietet, da Nachrichten auf dem Übertragungsweg transformiert und somit deren Inhalte oder deren Format an die Anforderungen des Empfängers angepasst werden können. [8, Kap. 3]

2.1.1. Messaging Patterns

Eine Messaging-basierte Kommunikation kann auf Basis unterschiedlicher *Messaging Patterns* erfolgen. Zwei elementare Messaging Patterns sind hierbei *Warteschlangen* und

Publish-Subscribe. [8, Kap. 6]

Warteschlangen (englisch: Queues) können angewendet werden, um zwei Komponenten in ihrem Ablauf voneinander zu entkoppeln. Mittels einer in der Warteschlange gespeicherten Nachricht kann bspw. ein Datenpaket hinterlegt werden, womit eine zweite Komponente asynchron eine Verarbeitung durchführen kann. Die sendenden Komponenten werden hierbei *Producer* und die empfangenen Komponenten als *Consumer* bezeichnet, wie in Abbildung 2.1 demonstriert. [5]

Abbildung 2.1.: Messaging Pattern: Warteschlangen

Das *Publish-Subscribe* Messaging Pattern basiert auf dem „Observer Pattern“ und kann verwendet werden, um das Routing von Nachrichten an mehrere Empfänger (hier: Subscriber) zu organisieren. Sendende Komponenten (hier: Publisher) schicken hierzu ihre Nachrichten an so genannte *Topics*. Subscriber können wiederum diese Topics abonnieren und erhalten fortan eine Kopie jeder Nachricht, welche an dieses Topic gesendet wird. Dieser Aufbau ist in Abbildung 2.2 veranschaulicht. [8, Kap. 6]

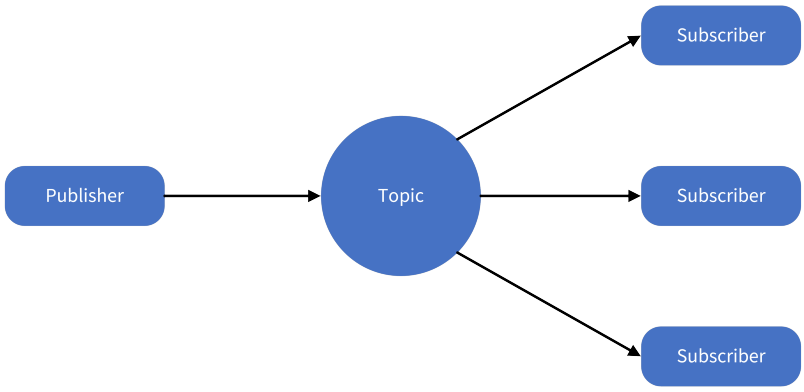

Abbildung 2.2.: Messaging Pattern: Publish-Subscribe

2.1.2. Zustellungsgarantien

Nachrichten können mittels verschiedener Zustellungsgarantien übertragen werden.

Bei einer *at-most-once*-basierten Zustellung erfolgt eine optimistische Zustellung: Eine Nachricht wird versucht zuzustellen und es wird davon ausgegangen, dass dies in den meisten Fällen gelingt. Tritt hierbei jedoch ein Problem auf oder ist der sendenden Anwendung nicht bekannt ob die Übertragung wirklich gelang (bspw. weil die verwendete Infrastruktur diese Informationen nicht zur Verfügung stellen kann), so wird diese Nachricht verworfen. Dies stellt sicher, dass eine Nachricht nicht mehrfach zugestellt und somit fälschlicherweise mehrfach verarbeitet wird, kann jedoch zu nicht zugestellten Nachrichten führen. [4]

Im Kontrast zu der at-most-once-basierten Zustellung bietet die *at-least-once*-basierte Zustellung eine höhere Gewissheit, dass eine Nachricht zugestellt und verarbeitet wird. Diese Zustellungsmethode erfordert es, dass nach einer erfolgreichen Verarbeitung einer Nachricht diese explizit gelöscht oder markiert wird, vgl. eines 2-Phase-Commits. Erfolgt dies nicht, bspw. weil die Verarbeitung fehlschlug, so wird die Nachricht erneut zugestellt. Somit reduziert sich die Chance, dass eine Nachricht nicht verarbeitet wird, jedoch kann es hierbei dazu kommen, dass eine Nachricht mehrfach verarbeitet wird. Dies kann passieren, wenn die empfangende Anwendung während der Verarbeitung, jedoch vor der Löschung der Nachricht abstürzt. Abbildung 2.3 veranschaulicht den Ablauf der at-most-once- sowie der at-least-once-basierten Zustellung. [4]

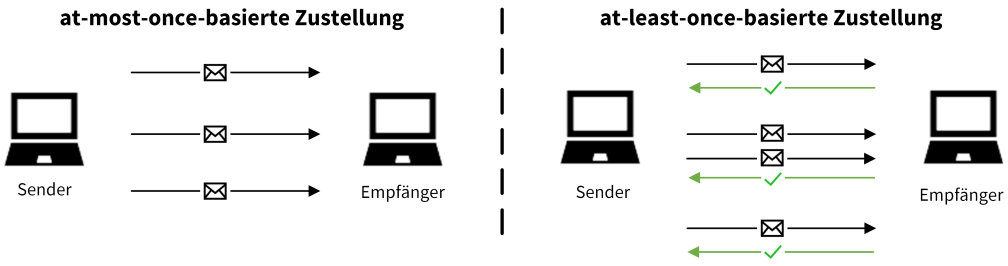

Abbildung 2.3.: Zustellungsgarantien

Eine *exactly-once*-basierte Zustellung ist sehr selten und erfordert die Integration von Client-seitiger Logik, welche bei einer at-least-once-basierten Zustellung garantiert, dass

mehrfach zugestellte Nachrichten dedupliziert und somit nicht mehrfach verarbeitet werden.

2.2. Machine-to-Machine

Das Konzept von Machine-to-Machine (kurz: M2M) beschreibt die Kommunikation zweier Maschinen oder Geräte, welche ohne direkte Teilnahme eines Benutzers miteinander kommunizieren. Diese Kommunikation kann drahtlos oder kabelgebunden erfolgen. [7, Kap. 1]

Hierbei gibt es verschiedene Kommunikationsprotokolle, welche mitunter (je nach verwendeter Hardware) darauf spezialisiert sind, mit geringen Hardwareressourcen auszukommen, z.B. bei *constrained devices*, also Geräten mit limitierender Hardware. Die Kommunikation erfolgt in M2M-Szenarien zwar oft über IP-Netzwerke, jedoch ist dies nicht vorgegeben und kann auch auf komplett andere Art und Weise stattfinden. [1]

2.3. Internet of Things

Das Thema des *Internet of Things* (kurz: IoT) beschreibt, wie Geräte des für Menschen üblichen Gebrauchs (z.B. ein Kühlschrank, Spiegel oder TV) miteinander vernetzt werden können um somit die Funktionalität und Komfort für den Benutzer zu erhöhen. Der Begriff wurde 1999 von Kevin Ashton geprägt, jedoch gab es das Thema bereits zuvor, wenn auch nicht unter einem einzelnen Begriff vereint. [18, Kap. 1] [24]

Prinzipiell erweitert IoT das Prinzip von M2M, also der direkten Kommunikation mehrerer Geräte untereinander, beschränkt diese jedoch üblicherweise auf IP-basierte Kommunikation und bezieht hierbei das Internet in diese Kommunikation mit ein. [1]

2.4. Edge Computing

Cloud Computing, inzwischen ein sehr bekanntes Prinzip, beschreibt die Anbindung von IT-Infrastruktur und Rechenkomponenten über das Internet. Eine typische Herausforderung bei der Anwendung von Cloud Computing kann jedoch der Umgang mit begrenzter Netzwerkbandbreite und -Latenz zwischen Client und Server sein.

Edge Computing beschreibt hingegen den Prozess, eine Rechenkomponente nah an datengenerierende Komponenten zu positionieren. In Kombination mit Cloud Computing bedeutet dies, dass zwischen den IoT-Geräten und der Cloud eine weitere zum IoT-Gerät lokale Rechenkomponente hinzugefügt wird, welche die Daten vom IoT-Gerät empfängt oder anfragt, diese potenziell aufbereitet und anschließend in die Cloud überträgt. Dies kann sinnvoll sein, wenn die IoT-Geräte selbst nicht in der Lage sind diese Übertragung selbst zu erledigen, z.B. durch inkompatible Kommunikationsprotokolle (bspw. Funk), wenn deren Rechenkapazität oder Übertragungsbandbreite hierfür nicht ausreicht oder aber wenn eine Nachbereitung der Daten erfolgen soll, welche die Rechen- oder Speicherkapazität des IoT-Geräts überschreiten würde. Ein Gerät in einer Edge-Umgebung hat mitunter keine Verbindung geschweige denn eine direkte Kenntnis von der Cloud. Dies kann sich positiv auf eine ggf. knappe Leistung oder Bandbreite der Internetverbindung auswirken. [9, Kap. 2]

Der Vergleich mit Cloud Computing ist in Abbildung 2.4 dargestellt.

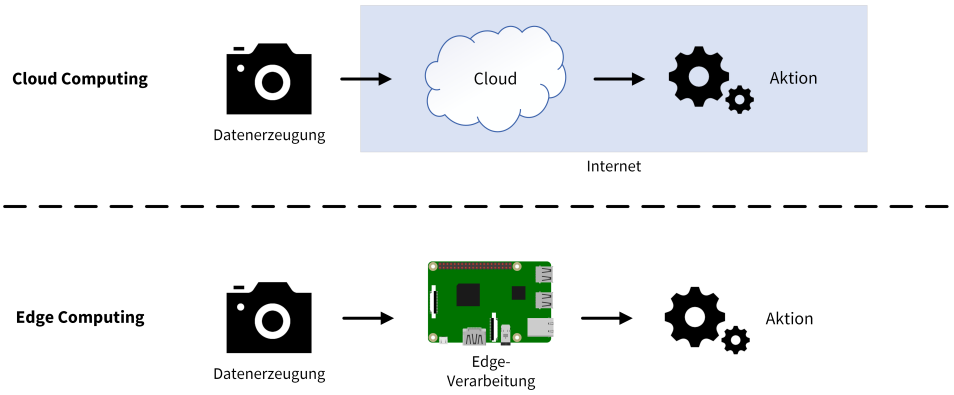

Abbildung 2.4.: Edge-Computing Vergleich

2.5. MQTT

Das MQ Telemetry Transport Protokoll (kurz: MQTT, vormals Message Queue Telemetry Transport) ist ein leichtgewichtiges M2M-Messaging-Protokoll. Es wurde spezifisch für die IoT-Welt konzipiert und basiert auf dem Publish-Subscribe Messaging Pattern. [17]

Durch das sparsame Protokoll mit geringem Nachrichtenoverhead eignet sich das Protokoll

insbesondere für begrenzte Geräte (bspw. constrained device) und/oder schwache Netzwerkanbindungen.

Der typische Aufbau mitsamt einem Broker, einem Publisher und mehreren Consumern kann der Abbildung 2.5 entnommen werden. MQTT erlaubt auch mehrere Broker, z.B. zur Lastverteilung, Unterteilung/Segmentierung oder für einen gestaffelten Aufbau (Kaskadierung).

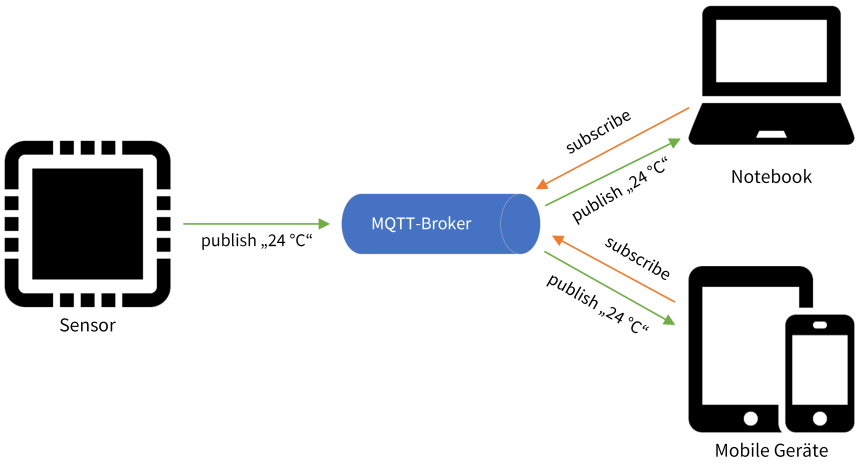

Abbildung 2.5.: MQTT-Aufbau

Inzwischen gibt es für jede gebräuchliche Programmiersprache Bibliotheken, mit denen eine MQTT-basierte Kommunikation erfolgen kann. Darüber hinaus sind mehrere verschiedene MQTT-Broker-Implementationen verfügbar, sowohl Open-Source als auch kommerziell unterstützt, welche unterschiedliche Anforderungen erfüllen können.

Wie beim Publish-Subscribe Messaging Pattern üblich, können Nachrichten durch das Abonnieren von Topics empfangen werden. Abonnements von Topics können entweder auf spezifische Topics oder aber mittels Wildcards abgeschlossen werden. Wildcards ermöglichen es, Teile der Topic-Bezeichnung dynamisch zu definieren, sodass ein einzelnes Abonnement bei mehreren effektiven Topic-Bezeichnungen greifen kann.

Für die Zustellung der Nachrichten unterstützt MQTT alle drei üblichen Zustellungsgarantien (Quality-of-Service-Level): at-most-once, at-least-once und exactly-once. [4]

2.6. Strategy-Pattern

Das Strategy-Pattern ist ein Design-Pattern, bei dem ein Algorithmus zur Lösung eines spezifischen Problems in einer leicht austauschbaren, gekapselten Form integriert werden kann. Die Wahl der zu verwendenden Strategie kann dann bspw. zur Laufzeit ausgewählt werden. Alternativ kann das Pattern auch als „Policy-Pattern“ bezeichnet werden. [6, S. 79-80]

In objektorientierten Programmiersprachen wird für solch eine Implementation oftmals Vererbung eingesetzt, bei der eine abstrakte Klasse die Schnittstelle vorgibt und die jeweiligen Strategien diese implementieren. Abbildung 2.6 zeigt hierfür ein Beispiel, bei dem drei verschiedene Implementierungen zum Formatieren von Daten zur Verfügung stehen und von einer Klasse (hier: Consumer) verwendet werden können. [6, S. 383-389]

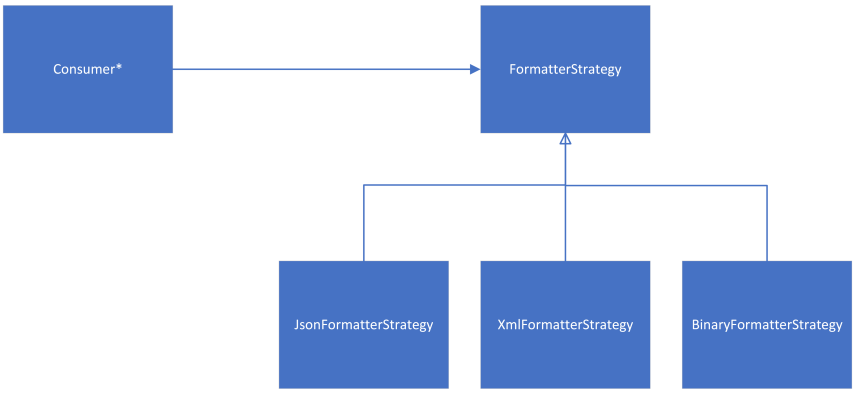

Abbildung 2.6.: Strategy-Pattern

2.7. Regelmaschine

Eine Regelmaschine (auch Regelinterpreter oder englisch „rule engine“ genannt) agiert zur Laufzeit anhand von definierten Geschäftsregeln und kann anhand einer Auswertung der Bedingungen von den Geschäftsregeln bedingt deren definierte Logik ausführen. Somit ist es eine Variante einer deklarativen Problemlösung. [20, S. 228-231]

Geschäftsregeln bestehen üblicherweise aus Bedingungen, welche zu Aktionen führen. Ein Beispiel für eine solche Geschäftsregel könnte z.B. sein: ≪Ist die Job-Bezeichnung des Kun-

den „Student“, so wird ihm bei der Rechnungserstellung automatisch ein Rabatt von 10 % gewährt». Beliebt sind auch Konstrukte aus den drei Bestandteilen Auslöser, Bedingungen und Aktionen, wie dies Abbildung 2.7 veranschaulicht.

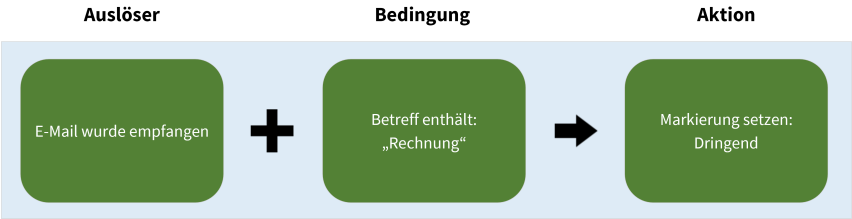

Abbildung 2.7.: Regelbeispiel

Die Definition solcher Abläufe durch ein Regelsystem kann den Vorteil haben, dass die zugrundeliegende Logik nicht im Code fest verankert ist, sondern extern verwaltet bzw. definiert werden kann. Des Weiteren kann eine solche Definition in einem vereinfachten Format vorliegen, sodass diese auch von Nicht-Entwicklern gelesen oder sogar verändert werden kann. [20, S. 228-231]

Je nach Art der Implementation kann solch eine Regelverarbeitung nicht-triviale zusätzliche Latenz mit sich bringen, weswegen die Verarbeitung mittels Regelmaschinen für Performanzsensitive Szenarien nachteilig sein kann. Des Weiteren kann solch ein Regelsystem je nach Implementation eine hohe Komplexität sowie zusätzliche Wartungsanforderungen mit sich bringen. [20, S. 228-234]

2.8. Skalierung

Unter Skalierung versteht man die Möglichkeit, die Leistung eines Systems durch Hinzufügen von Ressourcen oder Recheneinheiten zu verbessern. Dies ist notwendig, um ein System an steigende Anforderungen (wie z.B. steigende Zugriffszahlen) anzupassen. Skalierbarkeit ist somit wichtig, um auch bei steigenden Anforderungen in der produktiven Nutzung eine entsprechende Leistung garantieren zu können. [21, S. 26-27] [10] [20, S. 298-300]

Skalierung wird üblicherweise unterschieden in *Vertikale Skalierung* sowie *Horizontale Ska-*

lierung.

Vertikale Skalierung, auch „scale-up“ genannt, bezeichnet das Hinzufügen von Ressourcen zu einer bestehenden Recheneinheit, damit diese in der Lage ist, mehr Arbeitslast zu verrichten.

Als Alternative oder Ergänzung hierzu kann ebenso *Horizontale Skalierung* eingesetzt werden, auch „scale-out“ genannt. Hierbei werden weitere Recheneinheiten hinzugefügt, um die Arbeitslast auf diese zu verteilen.

Eine beispielhafte Darstellung der beiden Skalierungsarten sind in Abbildung 2.8 zu sehen.

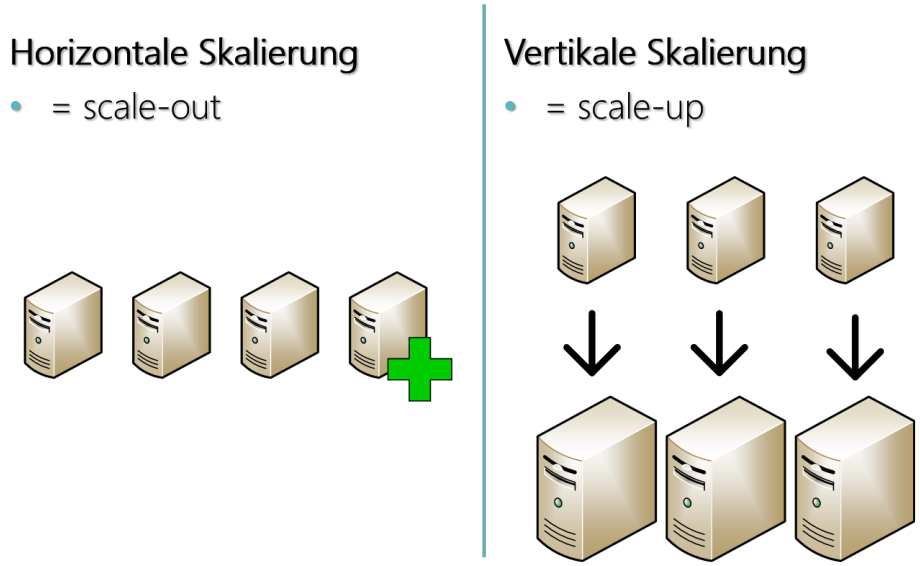

Abbildung 2.8.: Skalierungsarten

2.9. JarvisSuite

JarvisSuite bezeichnet das kombinierte Erzeugnis mehrerer durchgeführter Smart Home Projekte des Autors dieser Arbeit. Es begann in [13] als Meta-Lichtsteuerungssystem, mit dessen Hilfe unterschiedlichste Lichtsteuerungssysteme (wie z.B. Philips Hue) über eine Tablet-Oberfläche angesteuert werden konnten. Dies wurde im Rahmen von [14] um angepasste und physisch konstruierte Lichtschalter erweitert, welche mittels einer Touchfläche bedient werden konnten.

Als Forschungsprojekt wurde es in [15] weitergeführt und um ein Regelsystem ergänzt, mit dem der Benutzer das Verhalten seines Smart Homes in einfacher Art und Weise an seine eigenen Bedürfnisse anpassen kann.

Die JarvisSuite-Komponentenlandschaft besteht aus einer Vielzahl an unterschiedlichen Sensoren, Aktoren und Steuerungskomponenten. Diese sind via MQTT angebunden und kommunizieren auf Basis von offenen Formaten, wie bspw. JSON. [13, S. 9]

2.9.1. Umgebungssensor

Der Umgebungssensor wurde konzipiert, um in jedem Raum montiert zu werden und fortan verschiedene Werte des Raumes dauerhaft zu erfassen. Er basiert auf einem ESP8266 Mikrocontroller mit integrierter Wireless LAN Konnektivität und enthält die folgenden Sensorkomponenten: [15, S. 27-28]

- Passiver-Infrarot-Sensor (PIR)
- Temperatursensor
- Luftfeuchtigkeitssensor
- Barometer

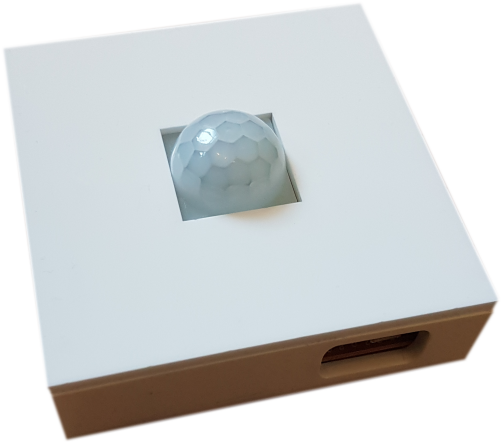

Abbildung 2.9.: Umgebungssensor

Die Bewegungsdaten des PIR-Sensors werden bei einer erfolgten Erkennung einer Bewegung übermittelt, wohingegen die restlichen Umgebungsdaten in einem Zyklus von 60 Sekunden übertragen werden. Der Sensor im fertigen Gehäuse ist in Abbildung 2.9 dargestellt.

2.9.2. Tür und Fenstersensor

Die Tür- und Fenstersensoren der *JarvisSuite* basieren ebenfalls auf einem ESP8266 und verwenden Magnetkontakte (sogenannte Reed-Switches) um den Geöffnet-Zustand von Türen und Fenstern zu erkennen. Die beiden Arten unterscheiden sich nur minimal in der Gehäuseform, da Türen und Fenster durch die baulichen Differenzen eine unterschiedliche Bauweise voraussetzen. Der Geöffnet-Zustand wird stets nach einer Änderung als auch nach einem erfolgten Stromverlust an das System übertragen. Ein verbauter Fenstersensor ist in Abbildung 2.10 dargestellt. [15, S. 13, S. 31-32]

Abbildung 2.10.: Fenstersensor

2.9.3. LED-Strip-Controller

Der LED-Strip-Controller fungiert als Aktor und ermöglicht es, LED-Leisten einzubinden und ansteuerbar zu machen. Die Farbe, Lichttemperatur und Intensität des zu aktivierenden Lichts kann hierbei in einer MQTT-Nachricht spezifiziert und somit vom Sender ferngesteuert definiert werden. Ein LED-Strip-Controller mitsamt einer montierten LED-Leiste ist in Abbildung 2.11 dargestellt. [13, S. 7]

Abbildung 2.11.: LED-Strip-Controller mitsamt angebundener LED-Lichtleiste

3. Analyse

In diesem Kapitel sollen die Anforderungen für eine Lösung erarbeitet werden, wie sie in der Einleitung beschrieben wurde. Diese werden nachfolgend in funktionale sowie nicht-funktionale Anforderungen unterteilt und näher erläutert.

3.1. Funktionale Anforderungen

Basierend auf den üblichen Szenarien von Sensoren und Aktoren, also dem zur Verfügung stellen von Werten und Funktionen, ergeben sich mehrere Funktionen, welche benötigt werden, um aus den Daten physischer Geräte ein virtuelles Gerät erschaffen zu können, das dem späteren Benutzer einen Mehrwert geben kann.

Der Umgang mit Werten kann unterteilt werden in die Aggregation mehrerer Werte in einen kombinierten Wert, sowie die Projektion von Werten, also z.B. die Umwandlung in ein anderes Format.

Für die Verarbeitung von Kommandos ist hingegen Funktionalität vonnöten, welche empfangene Kommandos an physische Aktoren weiterleiten kann.

Nachfolgend wird im Rahmen von Beispielen direkt von Sensoren und Aktoren gesprochen, um das Verständnis der Aussagen nicht durch zu generische Begriffe zu erschweren. Es kann sich hierbei jedoch stets auch um Mischformen handeln. Des Weiteren wird nachfolgend die Lösung, welche die Virtual Devices berechnen wird, als *Virtual Device Host* bezeichnet.

3.1.1. Werteaggregation

Eine wichtige Funktion ist die Aggregation mehrerer Ausgabedaten von Geräten zu einem kombinierten Wert. Die hierfür verwendete Art der Kalkulation sollte dabei flexibel sein:

Es sollte bspw. möglich sein, einen Mittelwert oder einen Minimal-/Maximalwert zu bestimmen, als auch bspw. eine Anwendungsfall-spezifische Kalkulation durchzuführen.

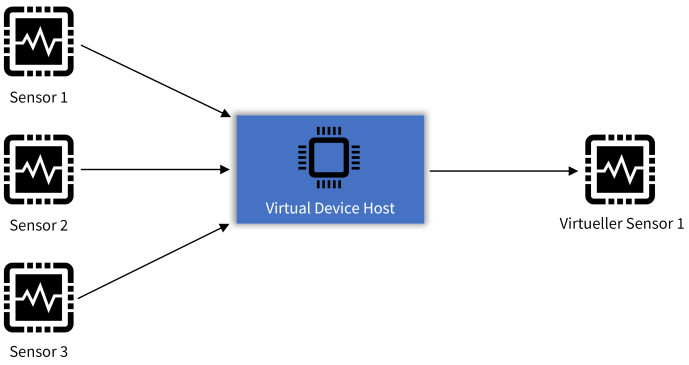

Abbildung 3.1.: Werteaggregation

Die Vorteile einer solchen Aggregation sind vielfältig:

Genauigkeit

Im Kontext von Sensoren kann durch die Kombination der Daten mehrerer Sensoren die Gesamtgenauigkeit verbessert werden. Trotz Kalibrierung von Sensoren haben diese mitunter Abweichungen in ihren Werten, sowohl von Produkt zu Produkt als auch von Stück zu Stück. Wie aus den Testergebnissen in [19] und [2] ersichtlich, gibt es bei den Messungen üblicher Temperatursensor-Komponenten klar erkennbare Abweichungen. Würde man demnach, wie in Abbildung 3.1 dargestellt, die Werte von mehreren Temperatursensoren unterschiedlicher Bauart miteinander kombinieren und bspw. hieraus den Durchschnitt ermitteln (optional mit Anpassungen falls eine spezifische Sensorart einen konstanten „Offset“ aufweist), so ist anzunehmen, dass dieses Ergebnis sich näher an der wirklichen Temperatur befindet, als es ein einzelner der Sensoren hätte messen können.

Entkoppelung

Unterbricht man die direkte Kommunikation zwischen Verbraucher und den zugegriffenen Geräten und setzt ein virtuelles Gerät zwischen diese, so werden diese voneinander entkoppelt. Der Verbraucher muss fortan nicht mehr über die Menge und die individuellen Eigenschaften jedes verwendeten abstrahierten Geräts Bescheid wissen, sondern benötigt nur alle Informationen über das virtualisierte Gerät. Theoretisch können sich die abstrahierten Geräte grundlegend ändern (bzw. ausgetauscht werden), ohne dass dies eine Veränderung

von Seiten des Verbrauchers erforderlich macht. Da es üblicherweise aufwändiger ist, mehrere Geräte anzusprechen, entfällt hierbei auch Last auf Seiten des Verbrauchers. Logisch gesehen wird hierbei aus der direkten Verbindung zwischen Verbraucher und den Geräten stattdessen eine transitive Verbindung. Abbildung 3.2 zeigt eine solche Entkoppelung am Beispiel von mehreren Sensoren.

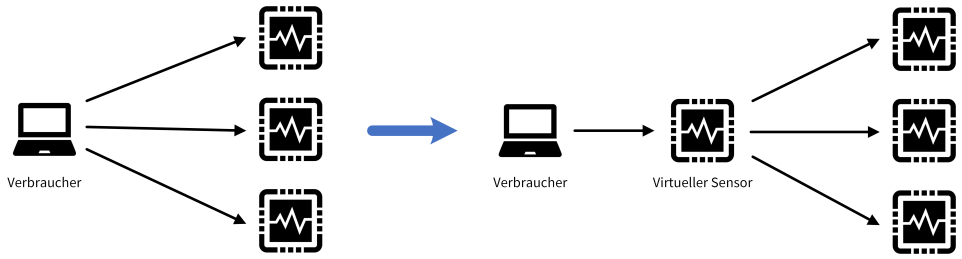

Abbildung 3.2.: Sensor-Entkoppelung

Ausfallsicherheit

Ein einzelner Sensor ist für sich gesehen zunächst ein Single-Point-of-Failure (kurz: SPOF). Fällt dieser Sensor aus, so hat ggf. der jeweilige Verbraucher keinen Zugriff auf weitere Daten. Um die Ausfallsicherheit von Systemen zu verbessern, gilt es, SPOFs zu vermeiden.

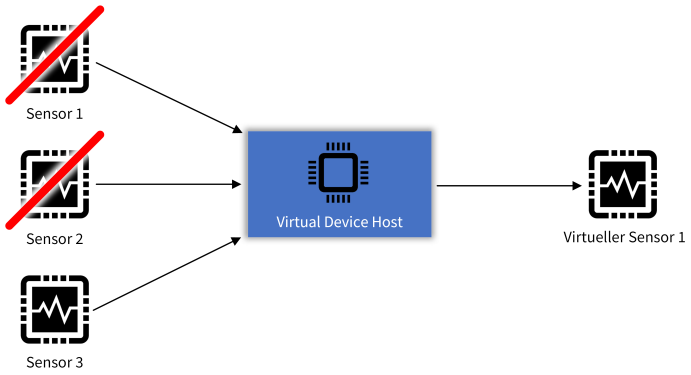

Abbildung 3.3.: Ausfall einer Teilmenge von Sensoren

Durch die Aggregation der Daten mehrerer Sensoren ist es möglich, den Ausfall eines oder mehrerer Sensoren zu verkraften, ohne die Funktionalität des Systems sowie die Integrität der Daten zu gefährden. Abbildung 3.3 veranschaulicht, wie auch nach einem Ausfall einer Teilmenge der verwendeten Sensoren eine Funktionalität weiter ermöglicht werden kann.

Mitunter wird hier jedoch über die Dauer des Ausfalls die Genauigkeit der aggregierten Daten reduziert.

Ausfalltransparenz

Ein potenzieller Nachteil der Entkoppelung ist, dass ein Verbraucher es nicht mehr mitbekommt, sollte ein vom verwendeten virtuellen Gerät abstrahiertes Gerät ausfallen und somit unter Umständen die Aktualität, Genauigkeit oder ggf. sogar die Verfügbarkeit der Werte des virtuellen Geräts beeinträchtigen. Während das Szenario im vorigen Abschnitt nur den Ausfall einer Teilmenge der abstrahierten Geräte zeigte, so stellt Abbildung 3.4 nun einen Ausfall aller Geräte da. Spätestens in diesem Falle sollte der Verbraucher Kenntnis von der Problemsituation haben können.

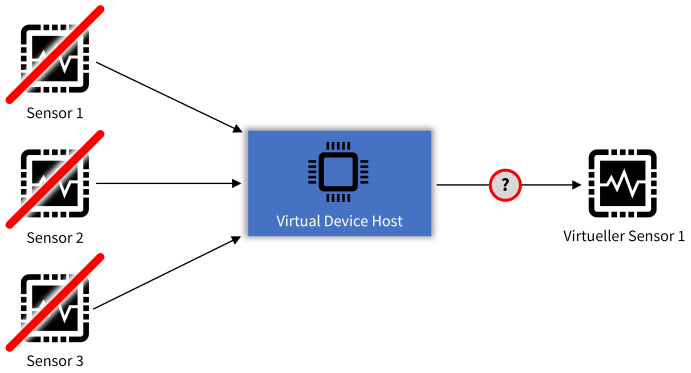

Abbildung 3.4.: Ausfall aller Sensoren

3.1.2. Werteprojektion

In vielen Fällen kann es sinnvoll sein, die Daten der abstrahierten Geräte in einem anderen Format darzustellen, als diese ursprünglich verschickt wurden. Diese Funktionalität wird im Kontext dieser Arbeit „Projektion“ genannt, basierend auf der Definition von Projektion als ≪Andersdarstellung≫ in [16].

Auch die Werteprojektion bietet mehrere Vorteile:

Kombinierte Darstellung

Ein möglicher Anwendungsfall für solch eine Projektion ist die kombinierte Darstellung verschiedener andersartiger Daten von mehreren Geräten zu einem gebündelten Paket. So ist es bspw. möglich, einen jeweils einzelnen Temperatur-, Luftfeuchtigkeits- und Luftdrucksensor als virtuell-kombinierten Umgebungssensor darzustellen. Abbildung 3.5 zeigt, wie solch eine Kombinierung schematisch aussehen würde.

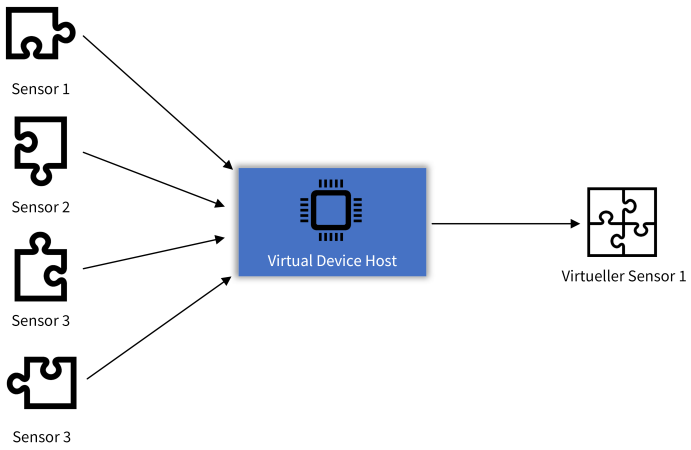

Abbildung 3.5.: Beispiel einer kombinierten Darstellung

Oftmals ist auch die Kombination aus Aggregation und Projektion sinnvoll. Hierdurch wird z.B. ermöglicht, einen Pool aus Sensoren unterschiedlicher Bauart, also ein virtueller Sensor, in einem gemeinsamen Format darzustellen und so den Verbraucher vor der Komplexität der unterschiedlichen Formate zu entlasten.

Kompatibilität

Durch Projektion kann die Kompatibilität in vielerlei Art verbessert werden.

Ein Anwendungsfall wäre bspw. der Wunsch einer Verwendung dreier verschiedener Sensorvarianten (welche jeweils nur XML, JSON und Binär als Format anbieten) mit einem Verbraucher, welcher ausschließlich JSON interpretieren kann. Mittels einer Projektion könnten hierbei die Sensoren gegenüber dem Verbraucher als drei JSON-basierte Sensoren abgebildet werden, so dass er diese problemlos verwenden kann. Dies ist in Abbildung 3.6 veranschaulicht.

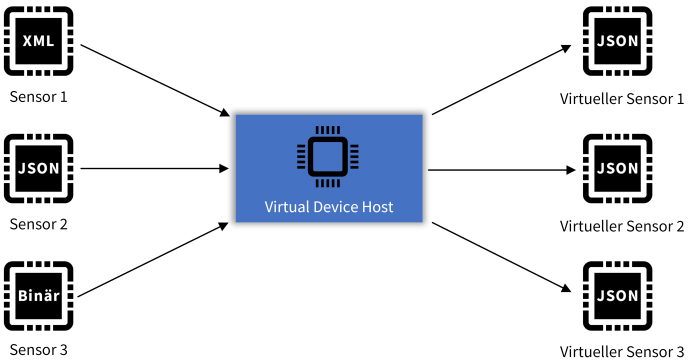

Abbildung 3.6.: Beispiel einer Projektion von Sensoren

Ein ähnliches Szenario wäre die Verwendung mehrerer Revisionen derselben Sensorart, welche jedoch ihre Ausgabe unterschiedlich durchführen und so den Zugriff erschweren. Möglicherweise besteht kein Herstellersupport mehr oder technische Hindernisse bestehen um diese vonseiten ihrer Firmware aneinander anzugleichen. Auch hierbei kann dies mittels Projektion gelöst werden.

Zusammenfassend lässt sich sagen, dass die Kompatibilität sowohl im Sinne der Aufwärts- als auch Abwärtskompatibilität mittels Projektion verbessert werden kann.

3.1.3. Kommando-Verarbeitung

Zusätzlich zu der Virtualisierung eines Sensors, inklusive der Aggregation und Projektion von deren Daten, ist für eine bidirektionale Kommunikation auch eine umgekehrte Verarbeitung vonnöten, also die Verarbeitung von Kommandos welche an Aktoren gesendet werden können.

Hinter jedem virtuellen Aktor können sich – wie bei den Sensoren – potenziell ein oder mehrere physische Aktoren befinden. Bei dem Eingang einer Nachricht an den virtuellen Aktor muss somit entschieden werden, ob und wie dies an die jeweiligen physischen Aktoren weitergeleitet werden soll. Der schematische Aufbau hiervon ist in Abbildung 3.7 dargestellt.

Routing

Ein sehr simpler Anwendungsfall ist das so genannte Routing, also bspw. die Weiterleitung bzw. das Kopieren des Kommandos an ein oder mehrere weitere Geräte bzw. auf

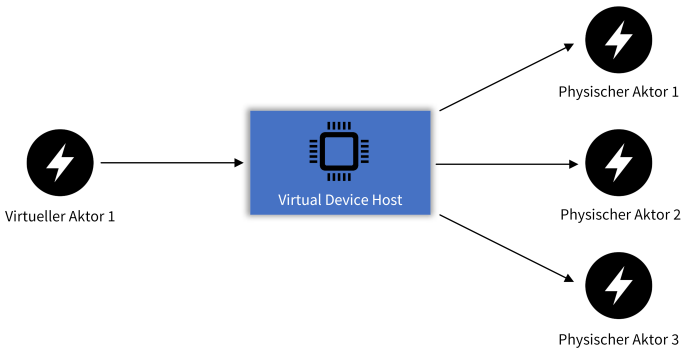

Abbildung 3.7.: Kommando-Verarbeitung

benutzerdefinierte MQTT-Topics.

Projektion

Wie schon bei der Verarbeitung der Werte so kann auch bei der Kommandoverarbeitung eine Projektion von Vorteil sein.

So kann auch hier der Inhalt oder das Format der Kommandos angepasst werden bevor es vom Empfänger oder von den Empfängern empfangen wird. Theoretisch kann sogar bereits das im vorigen Abschnitt beschriebene Routing zu einer Projektion auf logischer Ebene führen, da durch das Weiterleiten bzw. Kopieren auf ein anderes Topic ggf. bereits eine semantische Veränderung durchgeführt wird.

Entkoppelung

Auch bei Kommandos kann eine Entkoppelung vorteilhaft sein. Sendet ein Gerät stattdessen Kommandos an einen virtuellen Aktor, so kann sich die Identität und Menge des bzw. der Empfänger ändern, ohne dass das Gerät hiervon Kenntnis haben muss.

Kompatibilität

Ein Anwendungsfall zur Erhöhung der Kompatibilität wäre bspw. eine virtuelle Lampe, welche bei Empfang einer Aktivier-Nachricht (z.B. von einem Lichtschalter-Gerät) mehrere physische Lampen unterschiedlicher Bauart aktivieren soll, die jeweils auf unterschiedliche Kommandoformate horchen. Der virtuelle Aktor kann sich um die Umwandlung in die jeweiligen Formate kümmern und so die Kompatibilität zu den unterschiedlichen Lampen gewährleisten. Abbildung 3.8 zeigt eine schematische Darstellung dieses Beispiels.

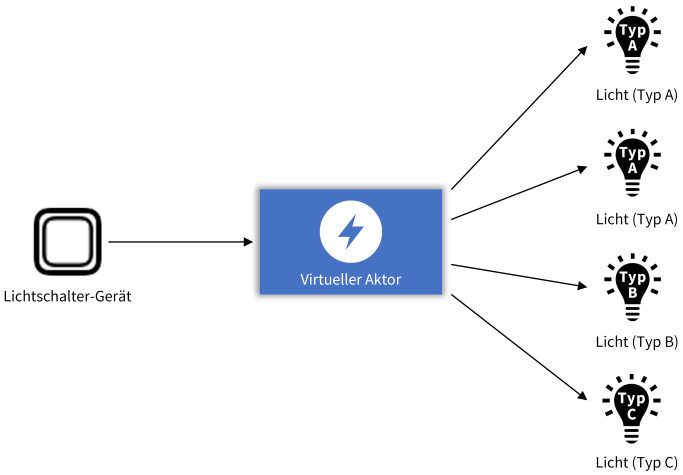

Abbildung 3.8.: Kommando-Projektion für erhöhte Kompatibilität

Lastverteilung

Es ist nicht zwingend vonnöten, dass alle physischen Aktoren an jedem Zugriff des virtuellen Aktors beteiligt sind. Beispielsweise kann das Ziel aus der Menge der abstrahierten physischen Aktoren zufällig ausgewählt werden. Dies käme einer Lastverteilung nach dem *round-robin*-Verfahren gleich, wie es in Abbildung 3.9 veranschaulicht ist.

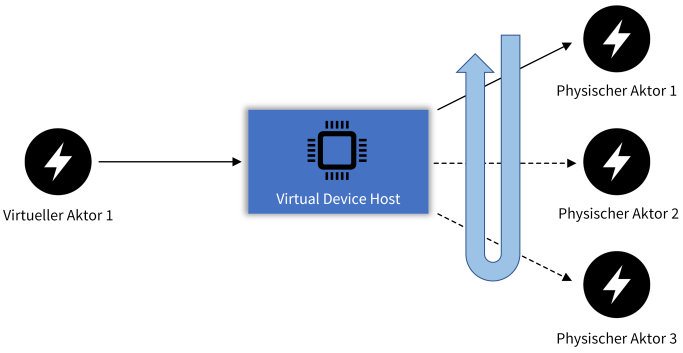

Abbildung 3.9.: Lastverteilung verschiedener physischer Aktoren

3.2. Nichtfunktionale Anforderungen

Um eine weitreichende und angenehme User Experience, sowie die Möglichkeit zur Pflege und Erweiterung der Software zu bieten, müssen folgende nicht-funktionale Anforderungen beachtet werden.

3.2.1. Plattform

Als Entwicklungsbasis für die skizzierte Lösung soll Python in der Version 3 genutzt werden. Für die Hardware wurde die Verwendung eines Raspberry Pi festgelegt. Diese beiden Entscheidungen wurden von externer Quelle für diese Arbeit bestimmt.

3.2.2. Zielgruppe

Die Zielgruppe ist hierbei definiert als Ingenieure, welche die Lösung in Rahmen von größeren Projekten einsetzen möchten. Dementsprechend kann ein gewisses technisches Verständnis vorausgesetzt werden.

3.2.3. Konfigurierbarkeit

Die zu entwickelnde Lösung soll als Framework für eine Vielzahl von Anwendungsfällen und Geräte-Kombinationen fungieren. Da somit die genauen Anforderungen für die spätere Verwendung nur sehr unklar definiert werden können, bedarf es einem hohen Grad an Konfigurierbarkeit, um nicht bereits während der Entwicklungszeit zuviele potenziell später aufkommenden Anwendungsfälle auszuschließen.

3.2.4. Erweiterbarkeit

Die Konfigurierbarkeit einer Anwendung ist jedoch inhärent begrenzt durch die Anwendungsfälle, welche vom Entwickler vorausgesehen und vorgesehen wurden. Wird dieser Bereich verlassen, so muss die Lösung dahingehend erweiterbar sein.

Die Lösung soll dementsprechend durch Schnittstellen oder Modularisierung in der Lage sein, neue oder erweiterte Funktionalitäten durch den späteren Benutzer hinzugefügt zu bekommen, um an neue oder geänderte Anforderungen angepasst werden zu können.

Für den Fall, dass dies nicht ausreicht und von weiteren Entwicklern ein tiefergehenderer Eingriff in den Sourcecode benötigt wird, soll der Sourcecode einen entsprechend guten Stil und leichte Lesbarkeit besitzen. Hierbei ist die Verwendung global genutzter Patterns und sinnvoller Sourcecode-Design-Prinzipien ratsam.

Die einfache Anwendung soll jedoch nur sehr geringe Python-Kenntnisse erfordern, z.B.

zum Erstellen von Konfigurationen. Die Erweiterung des Systems darf jedoch nicht-triviale Python-Kenntnisse vorraussetzen.

3.2.5. Geräteunterstützung

Da zum Entwicklungszeitpunkt noch nicht feststeht, für welche Geräte (Sensoren, Aktoren, etc.) die Lösung später angewendet werden soll, ist eine sehr offene Geräteunterstützung sehr wichtig. Dies kann durch eine hohe Erweiterbarkeit gelöst werden, indem Geräte später durch Anpassungen kompatibel gemacht werden können. Diese Erweiterbarkeit bezieht sich hierbei auf verwendete Formate, da für das zu verwendende Kommunikationsprotokoll auf MQTT fokussiert werden soll.

3.2.6. Performanz

Da die zu entwickelnde Lösung zwischen den kommunizierenden Geräten steht, entscheidet diese maßgeblich über die Performanz und Latenz des Zugriffs.

Eine Erhöhung der Latenz ist unumgänglich, da Nachrichten von der zu entwickelnden Lösung zunächst empfangen und anschließend neu versendet werden müssen, damit deren Inhalte verändert werden können. Werden Operationen basierend auf den empfangenen Werten ausgeführt (bspw. Wertaggregierung basierend auf mehreren Datenpaketen unterschiedlicher Quellen), so kann sich hierbei die Latenz aller betroffenen Pakete auf den jeweils höchsten Latenzwert der verwendeten Nachrichten verschlechtern. Dieser Latenz-Overhead soll jedoch nach Möglichkeit minimiert werden, um möglichst keine Anwendungsszenarien durch die Erhöhung der Latenz auszuschließen.

Für eine ausreichende Übertragungsgeschwindigkeit muss sichergestellt werden, dass die verwendeten Netzwerkschnittstellen und Netzwerkinfrastruktur den aufkommenden Datendurchsatz übertragen können, da sonst ein Engpass und somit eine Verringerung der zur Verfügung stehenden Performanz die Folge ist. Hier kann die Vorgabe des Raspberry Pi als verwendete Hardware zu einem begrenzenden Faktor werden, jedoch kann dies für einen Prototypen vernachlässigt werden.

3.2.7. Skalierbarkeit

Skalierbarkeit ist wichtig um auch bei steigenden Anforderungen in der produktiven Nutzung eine entsprechende Leistung garantieren zu können. Skalierbarkeit wird üblicherweise unterschieden in *Vertikale Skalierung* sowie *Horizontale Skalierung*. Eine genauere Erklärung hierzu befindet sich im Abschnitt 2.8.

Vertikale Skalierbarkeit

Eine *Vertikale Skalierung* kann durch das Erhöhen der Leistung der Maschine durchgeführt werden. Hierbei ist es zunächst wichtig zu ermitteln, an welcher Stelle (Prozessor, Arbeitsspeicher und Netzwerkbandbreite) mehr Leistung vonnöten ist. Im Rahmen des verwendeten Raspberry Pi ist eine vertikale Skalierung jedoch nur sehr begrenzt möglich, nämlich nur durch das Hinzufügen von mehr oder leistungsstärkeren Netzwerkschnittstellen sowie ein Wechsel auf potenziell zu einem späteren Zeitpunkt erhältliche neuere Raspberry Pi Versionen. Des Weiteren dürfte es theoretisch möglich sein, die entwickelte Lösung auf einem Gerät zu betreiben welches nicht auf einem Raspberry Pi basiert, jedoch überschreitet dies die Zielsetzung dieser Arbeit. Alternativ steht die horizontale Skalierung zur Verfügung.

Horizontale Skalierbarkeit

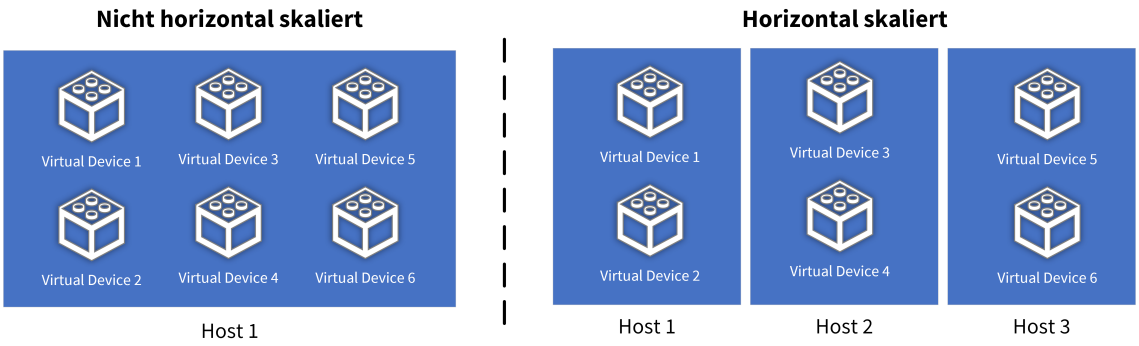

Abbildung 3.10.: Horizontale Skalierung

Eine *Horizontale Skalierung* soll durch Partitionierung der Virtual Devices erfolgen können. Da Virtual Devices als Konsistenzgrenze fungieren – und somit untereinander nichts teilen – macht es funktionell keinen Unterschied ob 300 Virtual Devices auf einem Host betrieben werden oder jeweils 100 auf drei verschiedenen Hosts, bis auf dass im letzteren Fall die

Performanz höchstwahrscheinlich höher sein wird. Beide Varianten sind in Abbildung 3.10 gegenübergestellt.

3.2.8. Verfügbarkeit

Da die zu entwickelnde Lösung später für die Übermittlung der Daten vieler Geräte zuständig sein wird, ist eine ausreichende Verfügbarkeit vonnöten. Fällt der Virtual Device Host aus, so sind potenziell auch alle dahinter stehenden Geräte unerreichbar bzw. erhalten alle darauf zugreifenden Geräte keine neuen Daten und sind somit wahrscheinlich in deren Funktionalität schwer beeinträchtigt.

Um eine hohe Verfügbarkeit zu ermöglichen wäre eine Ausfallsicherheit, ggf. sogar eine Vermeidung von Single-Point-of-Failures, sinnvoll. Solch eine Ausfallsicherheit wird dadurch erschwert, dass eine Erweiterung mittels Benutzer-Code möglich sein soll, also Source-Code, welcher vom späteren Benutzer geschrieben wird. Hierbei sind die externen Entwickler in der Pflicht, deren Code ausreichend zu testen um nicht die Stabilität der restlichen Anwendung zu gefährden. Wenn möglich soll die prototypische Anwendung jedoch zumindest leichte Fehler abfangen können.

Da es sich hierbei jedoch um einen Prototypen handelt, wird dieses Thema zwar angeschnitten, jedoch würde eine Implementation von Hochverfügbarkeit den Rahmen dieser Arbeit überschreiten.

Des Weiteren ist es für eine akzeptable Verfügbarkeit sehr sinnvoll, wenn das System eine Logging-Funktionalität besitzt. Hierdurch kann entweder im Fehlerfall oder im Rahmen der normalen Anwendung der Benutzer sich wichtige weitere Informationen beschaffen. Dies wird insbesondere dadurch unterstrichen, dass die Anwendung als Konsolen- bzw. Hintergrundanwendung implementiert werden soll.

3.2.9. Datensicherheit

Die Daten der prototypischen Anwendung können in zwei Bereiche unterteilt werden.

Die *Konfigurationsdaten* werden vom Benutzer angelegt und sollen von der Anwendung nicht verändert werden.

Die *Laufzeitdaten*, welche bspw. zum Erstellen der synthetischen Werte verwendet werden, müssen nicht persistent gespeichert werden. Hierbei handelt es sich um transiente Daten, welche nur eine begrenzte Lebenszeit besitzen und nach einem Neustart der Anwendung erneut gesammelt werden können.

Eine gesonderte Form der Datensicherheit für diese Daten, z.B. eine Persistierung oder ein Backup, ist demnach nicht erforderlich.

4. Lösungsansätze

Innerhalb dieses Kapitels sollen zwei Lösungsansätze beschrieben und bewertet werden um die in den Kapiteln *Einleitung* und *Analyse* beschriebene Anwendung umzusetzen.

4.1. Regelmaschine

Eine Regelmaschine, bzw. genauer gesagt ein Regelsystem, hat den großen Vorteil, dass es eine gute Konfigurierbarkeit durch die Definition von Regeln mitbringt. Diese Regeln bestehen üblicherweise aus fertigen Bausteinen, welche miteinander kombiniert und bis zu einem gewissen Grad konfiguriert werden können. Solche Bausteine könnten sowohl direkt anwendbare Rechenoperationen als auch Logikkonstrukte (wie z.B. Fallunterscheidungen etc.) sein.

Eine Konfigurierbarkeit könnte z.B. durch die Definition der verwendeten MQTT-Topics für die Ein- und Ausgabe sowie die darauf anzuwendende (Rechen-)Operation erfolgen, wie in Abbildung 4.1 demonstriert.

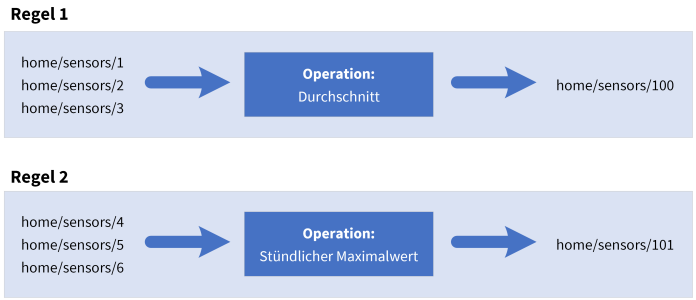

Abbildung 4.1.: Skizzierung des Regelmaschinen-Lösungsansatzes

Durch die Verwendung fertiger Bausteine kann solch eine Konfigurierbarkeit sehr anwenderfreundlich sein.

Eine Erweiterbarkeit ist in Regelsystemen meist nur im Rahmen der Weiterentwicklung des Gesamtsystems vorgesehen, weswegen insbesondere sehr spezielle Anwendungsszenarien hierbei potenziell nicht abgedeckt werden könnten.

Des Weiteren gelten Regelsysteme gemeinhin als nicht besonders schnell. Es ist jedoch fraglich, ob dies in diesem Falle relevant ist, da in den meisten Fällen weder IoT-Geräte sonderlich leistungsstark noch MQTT-Kommunikation für besonders latenzsensitive Kommunikation geeignet ist. [20, S. 228-234]

4.2. Strategy-Pattern

Das Strategy-Pattern ist eines der bekannteren Design Patterns für die Implementation von Austauschbarkeit. Eine Strategy wird stets für eine bestimmte Aufgabe definiert, wie z.B. die Berechnung eines aggregierten Wertes. Diese Strategy kann dann auf unterschiedliche Art und Weise implementiert werden, wobei jede Implementierung einen alternativen Ansatz verfolgt. Durch das Hinzufügen einer Strategy-Implementation ist somit eine Erweiterbarkeit möglich. Abbildung 4.2 zeigt solch einen Aufbau.

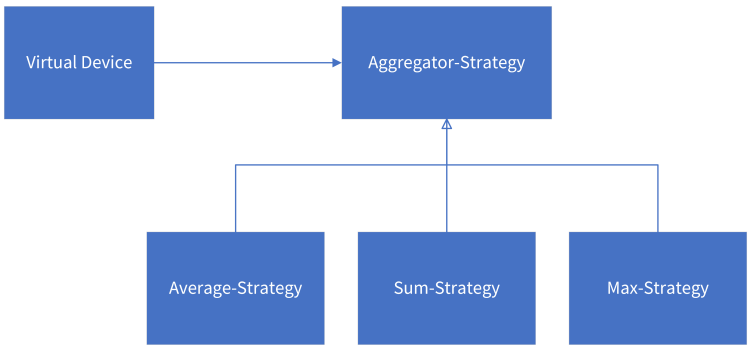

Abbildung 4.2.: Skizzierung des Strategy-Pattern-Lösungsansatzes

Da Design Patterns sich primär an Entwickler richten und sich auf der Ebene von Source-Code befinden, ist deren Verwendung durch den höheren Komplexitätsgrad komplizierter als die Verwendung fertiger Regelbausteine und bedarf in den meisten Fällen Programmierkenntnisse. Durch die Integration von Code seitens des späteren Benutzers ergibt sich jedoch ein höherer Grad an Flexibilität. Da das Strategy-Pattern sehr allgemein gehalten ist, spricht nichts dagegen dies an mehreren Stellen im Anwendungsdesign vorzusehen und somit diese hohe Erweiterbarkeit in mehreren Aspekten der Anwendung anzubieten.

4.3. Gegenüberstellung

Im direkten Vergleich beider Lösungsansätze ist das Strategy-Pattern der effizientere Ansatz, da es einen höheren Grad an Anpassbarkeit bietet und nicht auf einen festen Pool an Bausteinen begrenzt ist. Diese Flexibilität wird mit einer höheren Hürde zur Anpassung erkauft, da eine Erweiterbarkeit zwingend Programmierkenntnisse erfordert. Da bei der definierten Zielgruppe jedoch entsprechende Programmierkenntnisse vorausgesetzt werden können, ist dies vollkommen akzeptabel.

5. Entwurf

Das Kapitel Entwurf befasst sich mit der Umsetzung der in der *Analyse* herausgefundenen Anforderungen mittels des in *Lösungsansätze* ausgewählten Ansatzes in eine prototypisch anwendbare Softwareanwendung. Der Schwerpunkt der zu entwickelnden Lösung befasst sich mit dem korrekten Umgang von Daten.

5.1. Aufbau

Der Aufbau des Gesamtsystems besteht aus dem Virtual Device Host, mindestens einem MQTT-Broker und den verwendeten Sensoren und Aktoren. Es können jedoch auch mehrere MQTT-Broker in potenziell unterschiedlichen Netzwerken verwendet werden, sollte hierfür Bedarf bestehen. Diese Verbindungen zu MQTT-Brokern werden im Kontext dieser Arbeit als *Broker-Connections* bezeichnet. Jede Broker-Connection hat eine eindeutige Kennung, um bei der späteren Verwendung die Definition zu erleichtern, welche Komponente wann welche Broker-Connection verwenden soll. Der Virtual Device Host ist hierbei eine Konsolenanwendung, welche als Hintergrundanwendung betrieben werden kann.

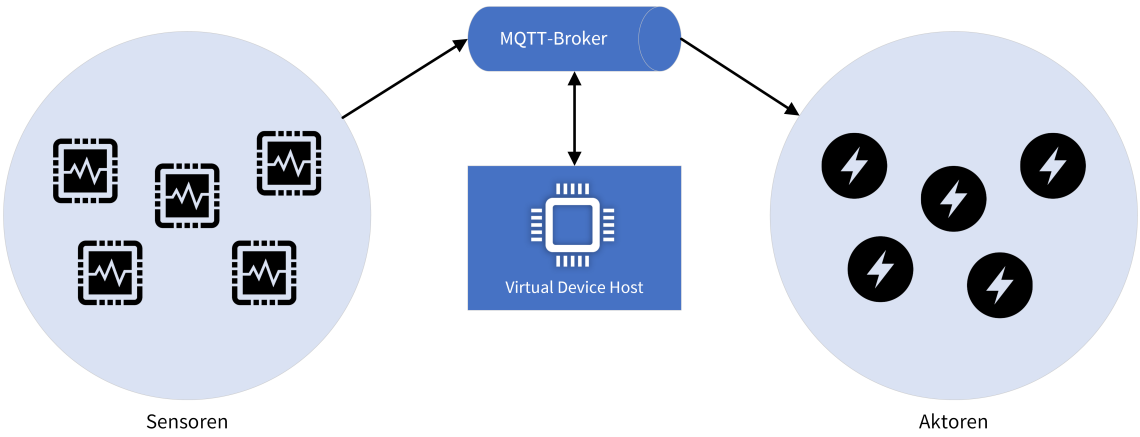

Abbildung 5.1.: Aufbau

5.2. Sensoren und Aktoren

Bevor basierend auf Sensordaten agiert sowie Aktoren gesteuert werden können, bedarf es zunächst der Komponenten, welche exemplarisch verwendet werden sollen. Hierfür sollen leicht abgewandelte Komponenten des aus früheren Arbeiten entstandenen Projekts *JarvisSuite* verwendet werden.

Die hiervon übernommenen Komponenten sind der Umgebungssensor, welcher u.a. die Temperatur eines Raums messen kann, sowie der LED-Strip-Controller, welcher als Aktor eine LED-Leiste ansteuern kann. Diese Komponenten wurden gewählt, da sowohl Temperatur als auch Licht sehr einfach zu greifende Thematiken und somit leicht nachvollziehbar sind.

5.2.1. Temperatursensor

Zusätzlich zu dem Umgebungssensor und LED-Strip-Controller des JarvisSuite-Projekts soll im Rahmen dieser Arbeit noch ein sehr simpler Sensor für die Demonstration verwendet werden, da es für die Werteaggregation und Werteprojektion von Vorteil ist, wenn mehrere Geräte angewendet werden können.

Basierend auf einer ähnlichen Bauweise wie die anderen Komponenten, ESP8266 als zentraler Mikroprozessor, wird dieser mit der Temperatursensor-Komponente DS18B20 bestückt, um als Temperatursensor fungieren zu können. Der DS18B20 wurde gewählt, da er kostengünstig beschaffbar ist, mittels seiner digitalen 1-Wire-Schnittstelle einfach zu verwenden ist sowie laut [2] vergleichsweise verlässliche Werte liefert.

Anders als die anderen verwendeten Sensoren soll dieser jedoch seine Daten nicht im JSON-Format übertragen, sondern nur die puren (minimal aufbereiteten) Fließkommazahl-Werte als MQTT-Nachricht übertragen. Dieser Unterschied zu den anderen Sensorkomponenten soll später aufzeigen, wie Werte in unterschiedlichen Formaten als Grundlage für die Berechnung der aggregierten Sensorwerte verwendet werden können.

5.3. Virtual Device

Der Kern der zu entwickelnden Lösung ist das so genannte *Virtual Device*. Es ist ein Container für die Funktionalität, welche den Zugriff auf reale Werte und Funktionen abstrahiert. Jedes Virtual Device ist hierbei eine natürliche Konsistenzgrenze und ist dementsprechend in sich bündig, hat also keine direkten Relationen zu anderen Virtual Devices, sondern kommuniziert ausschließlich via MQTT. Dies wurde absichtlich so gewählt, um Virtual Devices leichter abgrenzbar und skalierbar zu machen. Der grobe Kommunikationsverlauf des Virtual Devices ist in Abbildung 5.2 dargestellt.

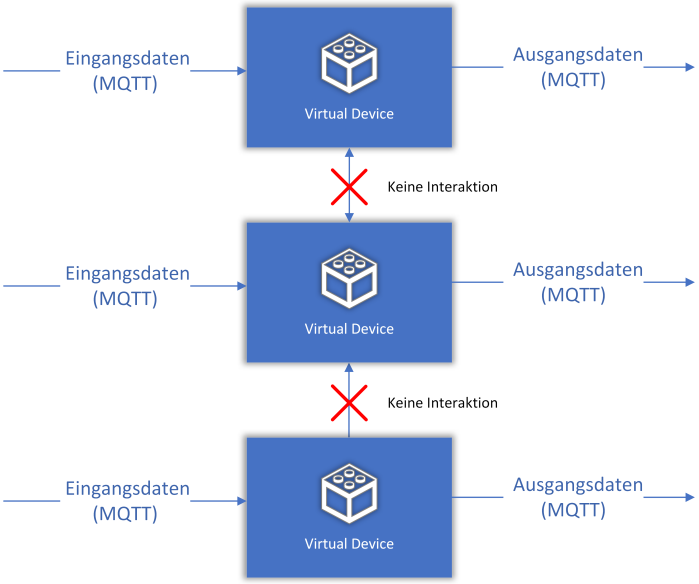

Abbildung 5.2.: Kommunikationsverlauf von Virtual Devices

Die Kernfunktionalität des Virtual Devices unterteilt sich in *Virtual Values* und *Virtual Functions*. Diese werden in den nachfolgenden Abschnitten näher erläutert.

5.4. Virtual Value

Virtual Values ermöglichen die in der Analyse spezifizierten Funktionalität der Wertaggregation und Wertprojektion. Um dies zu ermöglichen, müssen zunächst die Daten empfangen, extrahiert und interpretiert werden, um anschließend die gewünschten Werte zu generieren.

5.4.1. Datenempfang

Für den Empfang von Daten gibt es so genannte *Input-Data-Sources* (kurz: IDS), welche basierend auf einer spezifizierten MQTT-Verbindung und einem definierten MQTT-Topic auf eingehende Daten horchen und diese Snapshot-artig in einer Warteschlange zur weiteren Verwendung speichern. Jede Input-Data-Source-Instanz ist hierbei nur für den Empfang von Daten eines Geräts gedacht, da dies sonst Anomalien in der Weiterverarbeitung hervorrufen kann. Eine solche mögliche Anomalie wäre die Verfälschung der Interpretation der Daten eines Sensors, wenn ein weiterer Sensor dauerhaft auf demselben Topic höhere oder niedrigere Werte sendet.

Subscribe-Verhalten

Der Empfang von Daten eines MQTT-Brokers basiert Protokoll-bedingt auf Subscriptions. Solche Subscriptions können für ein spezifisches Topic oder mittels Wildcards offener bzw. generischer erstellt werden. Für die Verwaltung dieser Subscriptions sind die Broker-Connection-Instanzen zuständig. Jede Broker-Connection-Instanz (wovon es im Regelfall nur ein bis zwei geben dürfte, je nach Netzwerkaufbau) besitzt hierfür eine Empfangsschleife, welche Nachrichten vom MQTT-Broker empfängt und diese daraufhin lokal auswertet um zu entscheiden, welche Virtual Devices über die neuen Nachrichten benachrichtigt werden müssen. Da die zu entwickelnde Anwendung, je nach Anwendungsfall und Konfiguration, potenziell auf Hunderte oder sogar Tausende an Topics agieren wird, kommt jedoch die Frage auf, wie der Umgang mit den Subscriptions stattfinden soll.

Hierfür wurden drei Ansätze in Betracht gezogen:

Der *erste Ansatz* besteht aus einer einzelnen Wildcard-Subscription pro vorhandener Broker-Connection. Bei diesem Ansatz wäre sichergestellt, dass alle relevanten Nachrichten an die lokalen Virtual Devices weitergereicht werden können. Hierfür braucht die Broker-Connection auch keine komplette Liste aller verwendeten Topics, sondern muss nur bei jeder empfangenen Nachricht entscheiden können ob und zu welchen Virtual Devices diese weitergereicht werden muss. Der offensichtliche Nachteil ist jedoch, dass alle über den oder die MQTT-Broker veröffentlichten Nachrichten lokal empfangen und zumindest deren jeweiliges Topic ausgewertet werden muss. Je nach Nachrichtenaufkommen und Netz-

werkanbindung könnte hierbei ein Engpass entstehen.

Der *zweite Ansatz* basiert darauf, jedes einzelne vom Virtual Device verwendete Topic als explizite Subscription zu verwenden. Hierdurch wird sichergestellt, dass die Anwendung ausschließlich Nachrichten empfängt, welche auf abonnierten (und somit relevanten) Topics veröffentlicht wurden, wodurch die lokale Verarbeitungslast und die Auslastung der Netzwerkanbindung minimiert werden würde. Durch diesen Ansatz kann es potenziell zu Hunderten oder Tausenden an Subscriptions kommen. Es konnte keine verlässliche Quelle gefunden werden, die ein Maximum oder eine empfohlene Obergrenze an Subscriptions empfiehlt, jedoch ist es durchaus möglich, dass dieser Wert mit diesem Ansatz überschritten werden würde.

Der *dritte Ansatz* erweitert den zweiten Ansatz um einen Optimierungsalgorithmus. Anstatt aus jedem verwendeten Topic eine explizite Subscription zu generieren wird zuvor versucht, ähnliche Topics mittels Wildcards zu generalisieren und somit deren Anzahl zu reduzieren. Hierbei könnte bspw. aus einer Menge von zehn Topics, in deren Bezeichnung sich nur eine ID oder Nummer ändert, eine einzelne Subscription generiert werden, welche diese sich wechselnde Nummer durch eine Wildcard ersetzt und somit alle zehn Topics abdecken kann. Wie im ersten Ansatz ist auch hier anzunehmen, dass die Subscriptions auch Topics miteinbeziehen, welche für keine der vorhandenen Virtual Devices relevant sind und deren Nachrichten sofort wieder verworfen werden müssen, jedoch dürfte dies durch die durchgeführte Optimierung nur in sehr geringen Ausmaß der Fall sein.

Die drei Ansätze sind in Abbildung 5.3 gegenübergestellt.

Für den Prototypen wurde der erste Ansatz gewählt, da er der mit geringstem Aufwand zu implementierende ist und keine direkten Nachteile außer einem möglichen Leistungsengpass besitzt, was im Rahmen eines Prototyps vernachlässigbar ist. Bei steigenden Leistungsansprüchen könnte zukünftig hier jedoch auf einen anderen Ansatz gewechselt werden.

Da der verwendete Ansatz einen tiefgreifenden Einfluss auf die Anwendungsarchitektur hat, wurde hier nicht auf eine Auswahlmöglichkeit seitens des Benutzers gesetzt. Zudem

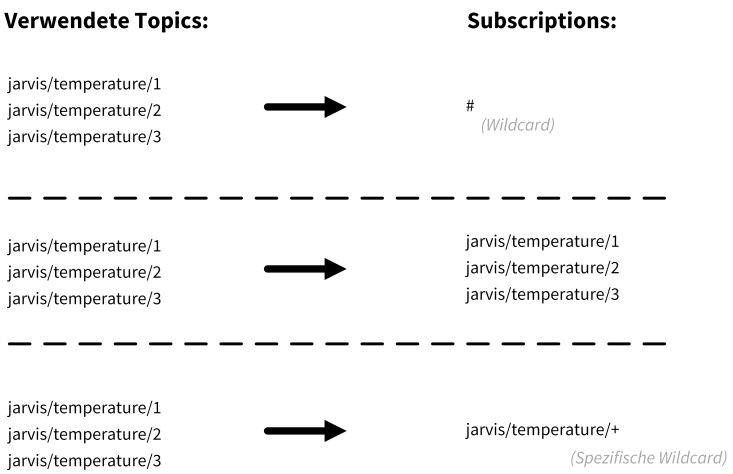

Abbildung 5.3.: Vergleich der Subscribe-Verhaltensansätze

kann nicht erwartet werden, dass der Benutzer die Implikationen der unterschiedlichen Ansätze nachvollziehen kann.

Wertentgegennahme

Eingehende Daten können in einer Vielzahl von Formaten empfangen werden. Dies schließt sowohl gängige Formate, wie z.B. XML und JSON, als auch potenzielle anwendungsspezifische oder proprietäre Formaten ein. Um trotz dieser Ungewissheit eine durchgehende Möglichkeit des korrekten Empfangs zu ermöglichen, wurde hier auf das Strategy-Pattern gesetzt, um die Parsing-Methode der eingehenden Daten austauschbar, erweiterbar und anpassbar zu gestalten.

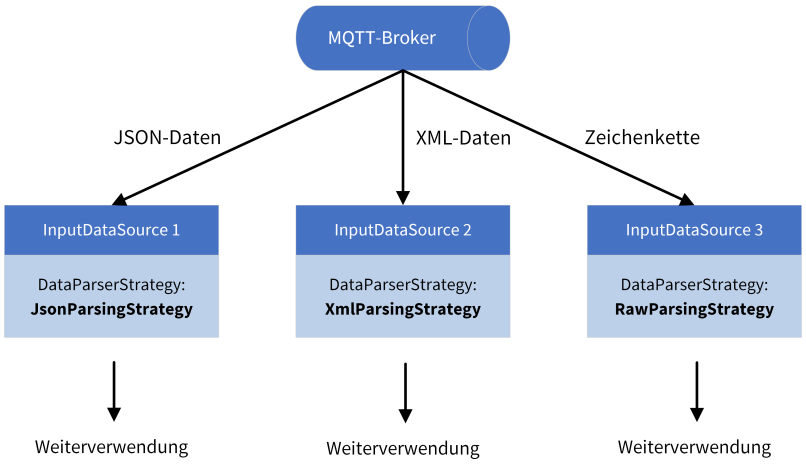

Abbildung 5.4.: Wertentgegennahme unterschiedlicher Formate

Für den Prototyp sind Strategien für XML und JSON sowie für den Empfang von rohen Werten (bspw. der Empfang eines puren Integer-Wertes oder eines Strings) implementiert worden. Den Datenfluss über diese Strategien ist in Abbildung 5.4 dargestellt.

Wertadressierung

Nachdem die eingegangenen Daten empfangen werden konnten, ist es nun erforderlich, die benötigte Information daraus zu extrahieren. Da es sich hierbei selten um atomare Daten handelt, sondern sich meist mehrere Daten in einem Datenpaket befinden, muss angegeben werden welcher Teil benötigt wird. Handelt es sich bspw. um JSON-Daten, so kann dies mittels eines *Pfads* definiert werden.

```
{
  environmentSensorIdentifier: 12,
  temperature: 23.7,
  humidity: 57,
  pressure: 1004.52
}
```

Abbildung 5.5.: Beispielnachricht von Umgebungssensor

Am Beispiel des simplen nicht-hierarchischen JSON-Objekts von Abbildung 5.5 wäre der Pfad, um an den Temperaturwert zu kommen, der Name des Attributs, also „temperature“. Dieser Prozess ist in Abbildung 5.6 dargestellt.

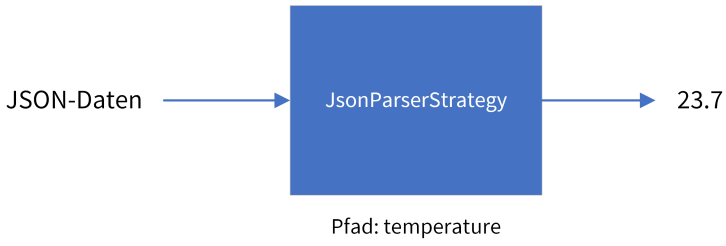

Abbildung 5.6.: Beispiel einer Wertadressierung

In komplexeren Beispielen kann es jedoch erforderlich werden eine Hierarchie mittels des Pfads zu „durchlaufen“ um einen tieferliegenderen Wert anzugeben. Die JSON-Spezifikation erlaubt bspw. sowohl verschachtelte Objekte als auch Arrays, wodurch eine Hierarchie von theoretisch unbegrenzter Tiefe möglich ist. [22, S. 5-6]

```
{
  device-id: 1337,
  maintenance-info: {
        contact-name: "Max Mustermann",
        phone-number: "555-0137-26"
  },
  sensors: [
    {
      environmentSensorIdentifier: 12,
      temperature: 23.7,
      humidity: 57,
      pressure: 1004.52
    }
  ],
  timestamp: "2012-04-23T18:25:43.511Z"
}
```

Abbildung 5.7.: Beispielnachricht von komplexem Umgebungssensor

Abbildung 5.7 zeigt die Ausgabedaten eines fiktiven Geräts, welche die Sensordaten als eigenes Objekt innerhalb eines Attributs zur Verfügung stellt. Erweitert man die Pfad-Interpretation jedoch um Zugriffsmöglichkeiten für Kindobjekte und Arrays und eine rekursive Verarbeitung, so sind auch beliebig-tiefe Hierarchien verwendbar. Mittels der in Abbildung 5.8 aufgelisteten Optionen ergibt sich somit ein Pfad von „sensors[0].temperature“ um in dem komplexeren Beispiel an den Temperaturwert zu gelangen.

abc	**Attribut**
.	**Kind-Objekt**
[n]	**Array-Element**

Abbildung 5.8.: Pfad-Operationen

Für die Wertadressierung von XML-formatierten Daten kann *XPath* verwendet werden. XPath ist eine Abfragesprache, welche es erlaubt, Teile eines XML-Dokuments zu adressieren. Hierfür bietet Python bereits verschiedene Bibliotheken an, wodurch diese Funktionalität leicht integriert werden kann.

5.4.2. Wertegenerierung

Nachdem die Daten nun von den Input-Data-Sources empfangen und von den verwendeten Data-Parser-Strategies extrahiert wurden, können diese nun weiterverwendet werden. Hierzu hat jeder Virtual Value eine zugewiesene Aggregator-Strategy. Diese Interaktion ist in Abbildung 5.9 dargestellt.

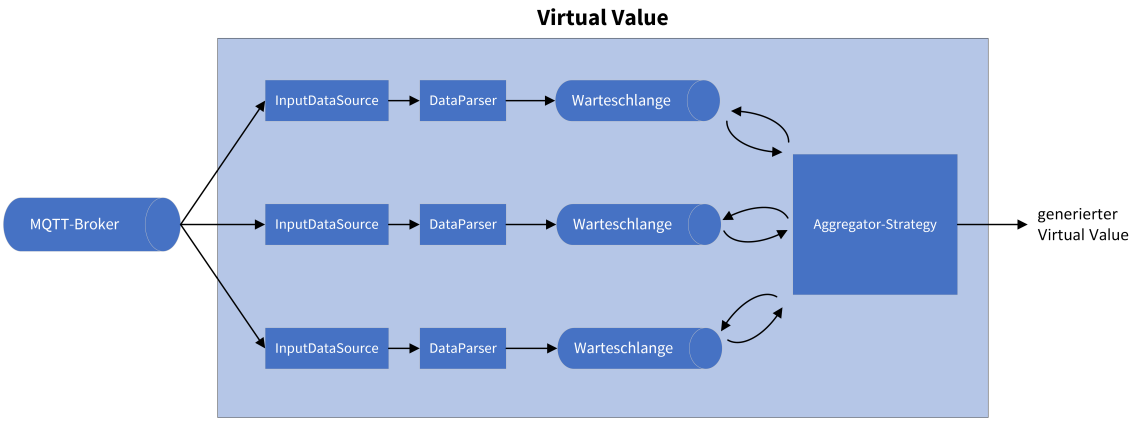

Abbildung 5.9.: Wertegenerierung

Diese Aggregator-Strategy entscheidet, wie aus potenziell mehreren eingehenden Werten ein einzelner synthetischer Ausgabewert errechnet werden soll. Ein Beispiel für eine solche Aggregator-Strategy-Implementation ist die Basic-Float-Average-Aggregator-Strategy, welche aus einer Menge an eingehenden Fließkommazahl-Werten einen Durchschnittswert errechnet. Diese könnte bspw. in Kombination mit Temperatursensoren verwendet werden, um einen durchschnittlichen Temperaturwert für einen Raum oder ein Gebäude als Virtual Value zur Verfügung zu stellen.

Aggregator-Strategy-Instanzen können hierbei „stateful“ implementiert werden, also zwischen Kalkulationen Werte behalten, um so ihren eigenen Verlauf zu kennen. Dies ist bspw. notwendig, um einen Moving-Average-Wert zu berechnen. Die Moving-Average-Aggregator-Strategy speichert die letzten n Werte (n ist hierbei konfigurierbar für eine anpassbare Präzision) um den Moving-Average-Wert zu errechnen.

Technisch gesehen besteht jede Aggregator-Strategy aus (mindestens) drei Methoden, wie im UML-Diagramm in Abbildung 5.10 ersichtlich. Diese sind „prepare“, „aggregate“ und

„synthesize_value“.

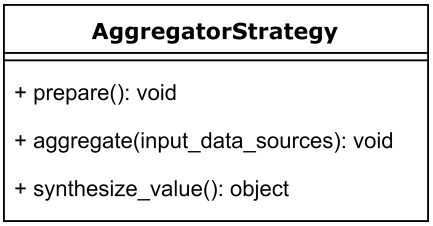

Abbildung 5.10.: UML-Diagramm der abstrakten Aggregator-Strategy

prepare ist die Initialisierungsmethode der Strategy-Implementation. Sie erlaubt die Vorbereitung aller für die spätere Kalkulation benötigten Instanzvariablen.

Die *aggregate*-Methode wird stets ausgeführt, wenn die Input-Data-Sources des Virtual-Values, zu dem die Aggregator-Strategy-Instanz gehört, neue Daten empfangen haben. Diese Daten werden üblicherweise verarbeitet und für die nächste anstehende Kalkulation zwischengespeichert.

synthesize_value wird aufgerufen, sobald ein neuer Wert kalkuliert werden soll. Nach erfolgter Kalkulation wird der neue synthetische Wert zur Weiterverarbeitung zurückgegeben.

5.4.3. Gruppierung

Virtual Values sind selten alleine stehend. Häufig werden mehrere hiervon als eine zusammengehörende Nachricht versendet. *Virtual Value Groups* sind Container, um diese Virtual Values entsprechend zu gruppieren und einige der nachfolgend erklärten Aspekte für alle gruppierten Virtual Values zusammengefasst zu definieren.

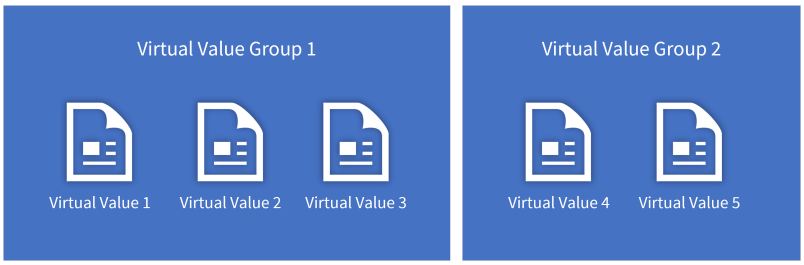

Abbildung 5.11.: Gruppierung mittels Virtual Value Groups

5.4.4. Generierung

Da Virtual Value Groups mitsamt ihren Virtual Values zusammen verschickt werden, sind es die Virtual Value Groups, welche den Anstoß zur Erstellung der Virtual Values geben. Da der Auslöser hierfür auf unterschiedliche Art und Weise stattfinden kann, wird auch hier das Strategy-Pattern angewendet, um dem späteren Benutzer eine höhere Flexibilität zu bieten.

Für die somit resultierende *Generation-Strategy* bietet der Prototyp bereits drei Implementationen:

Die *Timed-Generation-Strategy* kann verwendet werden, falls eine Virtual Value Group zyklisch, also z.B. alle fünf Sekunden, veröffentlicht werden soll. Python bringt hierfür leider keine adäquate Möglichkeit mit, sondern erlaubt nur die Unterbrechung der Ausführung eines Threads für eine definierte Dauer. Dies ist jedoch nicht äquivalent mit der Anforderung, eine Aktion alle n Sekunden durchzuführen, da hierbei die Ausführungszeit zwischen den Unterbrechungen nicht beachtet wird, weswegen unweigerlich eine Abweichung entstehen wird, wie in Abbildung 5.12 ersichtlich. Für diese Strategy-Implementation muss die wirkliche Unterbrechungszeit also bei jedem Durchlauf neu kalkuliert werden, damit diese Abweichung kompensiert werden kann.

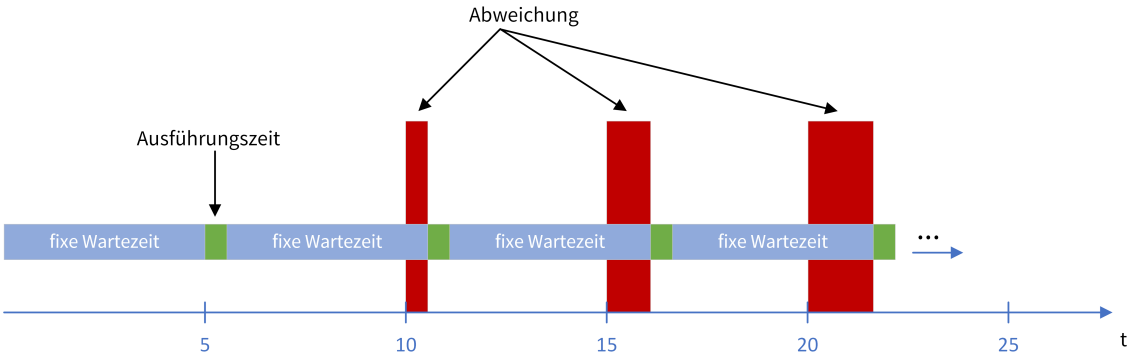

Abbildung 5.12.: Beispiel einer Wartezeitabweichung zwischen Ausführungen

Die *On-Data-Received-Generation-Strategy* hat im Vergleich zur vorigen Strategy-Implementation keinen festen Rhythmus, sondern horcht auf eingehende Daten der Input Data Sources und stößt beim Empfang solcher Daten den Generierungsprozess an. Dies kann sinnvoll sein,

wenn die verwendeten Geräte nur sehr unregelmäßig oder selten neue Daten senden und daher unnötige Generierungen vermieden werden sollen. Ein weiterer sinnvoller Anwendungsfall wäre es, wenn die Wartezeit bis zur nächsten Generierung möglichst kurz sein soll. Eine Gefahr besteht hier jedoch bei einer hohen Sendefrequenz und/oder einem hohen Rechenaufwand für Kalkulationen. Des Weiteren kann hierbei leicht ein Wasserfall-Effekt entstehen, wenn Virtual Values miteinander verkettet sind. Abbildung 5.13 veranschaulicht dies: Wird für jeden abgebildeten Virtual Value diese Generator-Strategy verwendet, so führt jede empfangene Nachricht der Sensoren zu einer Vielzahl an potenziell aufwendigen Kalkulationen

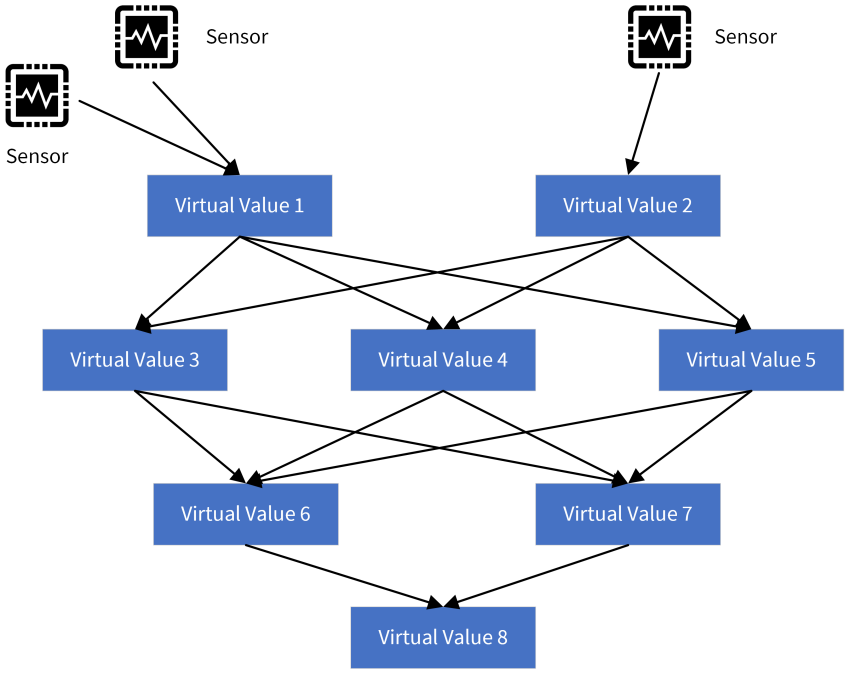

Abbildung 5.13.: Wasserfall-Effekt verketteter Virtual Values

Dies kann erwünschtes Verhalten sein, kann jedoch bei geringer Vorsicht auch leicht zu Leistungsproblemen führen.

Die *Continuous-Generation-Strategy* ist im Vergleich zu den anderen Strategien simpel: Nach jeder erfolgten Generierung wird sofort die nächste Generierung angestoßen.

5.4.5. Packaging

Bevor die virtuellen Werte auf dem spezifizierten Ausgabe-MQTT-Broker publiziert werden können, müssen diese zunächst in ein entsprechendes Format gebracht werden. MQTT schreibt das Format hierbei nicht vor, sondern agiert auf einem simplen Byte-Array. Das favorisierte Format kann jedoch von Anwendungsfall zu Anwendungsfall unterschiedlich sein, daher ist es auch hier sinnvoll eine Erweiterbarkeit mittels des Strategy-Patterns zu ermöglichen. Die Aufgabenstellung an eine solche Packager-Strategy ist demnach die Umwandlung der Daten der Virtual Values in ein Byte-Array.

Der Prototyp beinhaltet bereits zwei solcher Strategien, nämlich die String-Replace-Packager-Strategy sowie die Json-Array-Packager-Strategy.

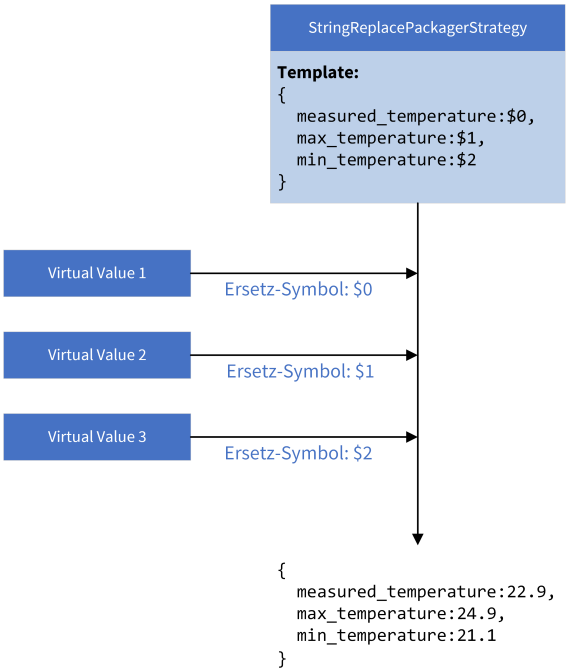

Abbildung 5.14.: Packaging-Beispiel

Die *String-Replace-Packager-Strategy* erfordert eine Zeichenkette als Vorlage einer Nachricht mitsamt austauschbaren Platzhaltern. Diese Platzhalter werden bei jedem Erstellen einer Ausgabe-Nachricht durch die aktuellsten Inhalte der jeweiligen Virtual Values ersetzt. Hierdurch ist es sehr einfach möglich ein sowohl leichtes als auch komplexes Format auszugeben, solange dies als Zeichenkette dargestellt werden kann. Abbildung 5.14 zeigt beispielhaft wie drei Virtual Values in ein JSON-Template integriert werden.

Die *Json-Array-Packager-Strategy* geht hierbei simpler vor und serialisiert alle Virtual Values eines Virtual Devices in ein eindimensionales Array und wandelt dies in eine JSON-Zeichenkette um.

5.4.6. Veröffentlichung

Ist ein Generierungsprozess abgeschlossen und wurde die auszugebende Nachricht von der verwendeten Packager-Strategy gebündelt, so obliegt es dem Data-Publisher diese in das jeweilige MQTT-Topic über eine spezifizierte Broker-Connection zu publizieren. Hierbei wird bei der Initialisierung der Virtual Value Group das Quality-of-Service-Level (kurz: QoS-Level) gesetzt, welches bei der Veröffentlichung genutzt werden soll.

Insgesamt ergibt dies eine Pipeline aus mehreren Komponenten, vom Datenempfang bis zur Datenveröffentlichung. Diese Pipeline ist zum einfacheren Verständnis schematisch in Abbildung 5.15 dargestellt.

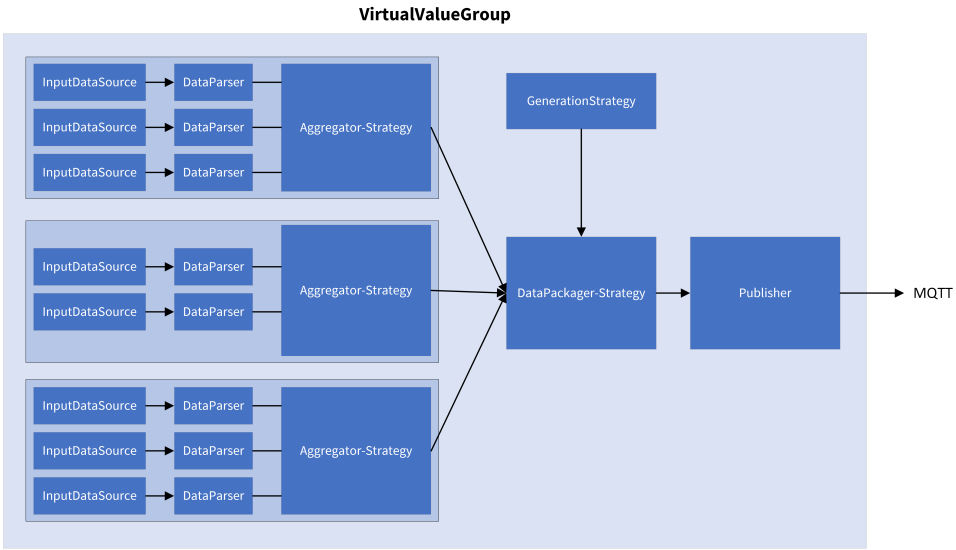

Abbildung 5.15.: Generierungsprozess einer Virtual Value Group

5.5. Virtual Function

Virtual Functions sind das Gegenstück zu den Virtual Values. Statt der Abstrahierung von Sensoren geht es bei Virtual Functions um die Abstrahierung von Aktoren, also Geräten, welche auf eingehende Kommandos horchen und nach Empfang solcher Befehle entsprechende Aktionen ausführen. Die Schnittstelle, welche diese Aktoren anbieten, sind dement-

sprechend aufrufbare Funktionen.

Virtual Functions sollen diese Funktionen abstrahieren und erweiterte Funktionalität hierfür anbieten. Wegen der durch MQTT-gegebenen nachrichtenbasierten Kommunikation werden Virtual Functions genauso wie die physischen Funktionen durch den Empfang einer Nachricht auf einem spezifischen Topic ausgelöst. Diese Nachricht wird Auslösernachricht genannt.

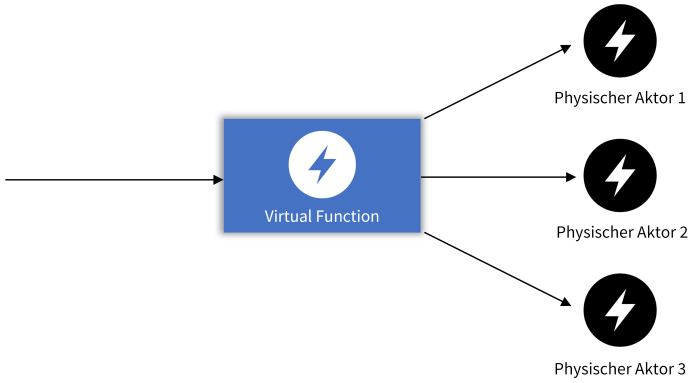

Abbildung 5.16.: Schema einer Virtual Function

5.5.1. Aufrufziele

Jede Virtual Function besitzt eine oder mehrere Aufrufziele (hier: Output-Targets). Diese definieren, an welche Ziele eine Nachricht beim Aufruf der Virtual Function gesendet werden soll. Üblicherweise dürfte jeweils ein Aufrufziel pro physischem Aktor bzw. dessen erreichbare Funktion erstellt werden.

5.5.2. Aufrufgenerierung

Um aus der empfangenen Auslöser-Nachricht eine Nachricht für ein Aufrufziel zu generieren gibt es die *Output-Generator-Strategy*. Abbildung 5.17 zeigt den Prozess, welcher nach dem Empfang einer solchen Auslösernachricht angestoßen wird.

Der Prototyp besitzt bereits vier *Output-Generator-Strategy*-Implementationen. Diese werden nachfolgend vorgestellt:

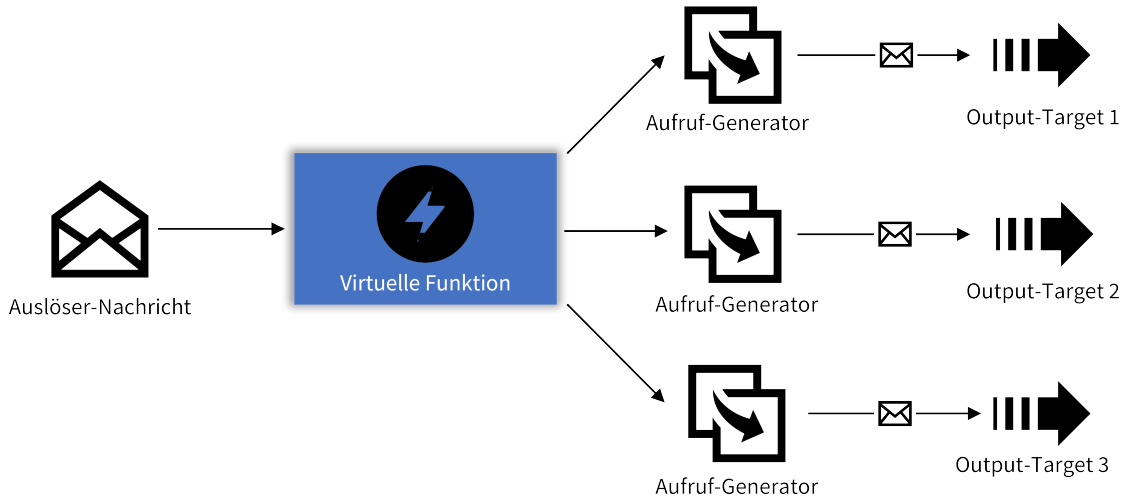

Abbildung 5.17.: Ablauf der Aufrufgenerierung mit drei Aufrufzielen

Kopie

Die *Copy-Output-Generator-Strategy* ist die einfachste Implementation. Sie behält den Inhalt der Auslöser-Nachricht 1:1 bei, sodass eine Kopie an das Aufrufziel verschickt wird.

Zeichenketten-Austausch

Die *String-Replace-Output-Generator-Strategy* wird konfiguriert mit Zeichenketten-Tupeln, womit in der Auslöser-Nachricht Zeichenketten ersetzt werden und die daraus resultierende Zeichenkette an das Aufrufziel verschickt wird.

Reguläre Ausdrücke

Die *Regex-Replace-Output-Generator-Strategy* nutzt einen definierten regulären Ausdruck, um den Inhalt der Auslöser-Nachricht zu modifizieren. Diese Strategie ist sinnvoll, wenn das Austauschen von statischen Zeichenketten nicht möglich oder ausreichend ist. Ein Beispiel für solch eine Verwendung ist in Abbildung 5.18 dargestellt.

JSON-Eigenschaft hinzufügen

Die *Add-Json-Property-Output-Generator-Strategy* fügt einer bereits JSON-formatierten Auslösernachricht eine weitere JSON-Eigenschaft hinzu.

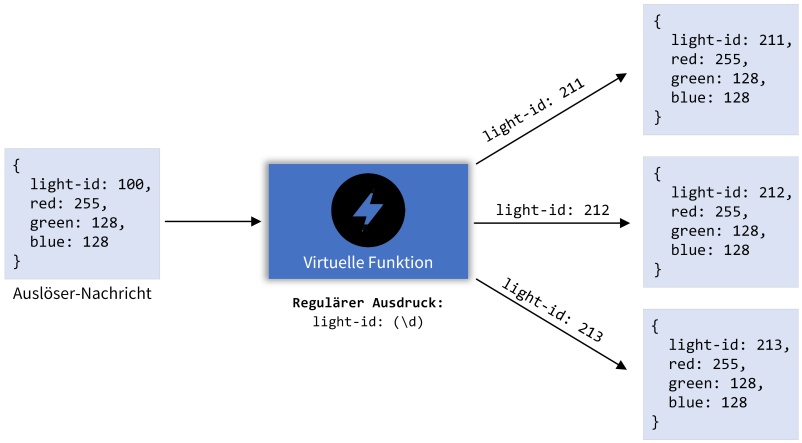

Abbildung 5.18.: Aufrufgenerierung via regulärer Ausdrücke

5.5.3. Asynchrone Verarbeitung

Die Verarbeitung der Auslösernachricht, die Aufrufgenerierung sowie das Versenden aller Nachrichten kann je nach Komplexität der Daten und Strategy-Wahl durchaus eine nicht vernachlässigbare Dauer in Anspruch nehmen. Da die erste Verarbeitung aller empfangenen Nachrichten pro Broker-Connection über einen einzelnen Thread geschieht, ist es wichtig für die Gesamtperformance, dass pro empfangener Nachricht nur minimale Arbeitslast vonnöten ist. Daher soll hier auf eine asynchrone Verarbeitung gesetzt werden.

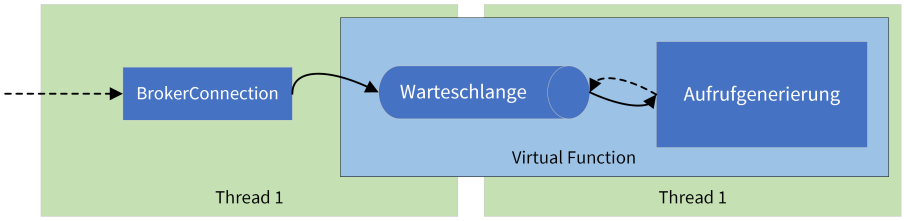

Abbildung 5.19.: Ablauf der asynchronen Verarbeitung

Wie in Abbildung 5.19 dargestellt, wird hierzu jede von einer Broker-Connection empfangene Nachricht, welche für eine Virtual Function relevant ist, in einer Warteschlange gespeichert. Der Thread zur Aufrufgenerierung der Virtual Function kann diese Nachrichten nun asynchron auslesen und entsprechend verarbeiten.

5.6. Virtual Device Host

Die Anwendung wird als Virtual Device Host bezeichnet, da deren Hauptfunktionalität die Organisation und zur-Verfügung-stellung der Virtual Devices ist. Sie soll als Python-Anwendung implementiert werden, da - wie bereits im Abschnitt *Analyse* angedeutet - dies als Vorgabe an das Projekt gestellt wurde. Aus derselben Vorgabe erschließt sich auch bereits der Raspberry Pi als verwendete Plattform, auf dem die Anwendung betrieben werden soll. Python ist die favorisierte Entwicklungsumgebung auf dem Raspberry Pi, daher ist diese Kombination sehr sinnvoll als zu verwendende Plattform.

5.6.1. Konfiguration

Eine Anforderung für die Konfiguration, welche im Rahmen der Analyse festgelegt wurde, ist die Möglichkeit, mehrere erstelle Konfigurationen besitzen und beim Anwendungsstart auswählen zu können.

Die Konfigurierbarkeit selbst könnte hierbei auf unterschiedliche Art und Weise konzipiert werden. Da der ausgewählte Lösungsansatz des Kernproblems das Strategy-Pattern ist, welches ein sehr Code-lastiger Ansatz ist, liegt es jedoch nahe auch die Konfiguration mittels Sourcecode erfolgen zu lassen. Hierzu gibt es eine Configuration-Strategy, welche sechs Schnittstellen zur Konfiguration anbietet.

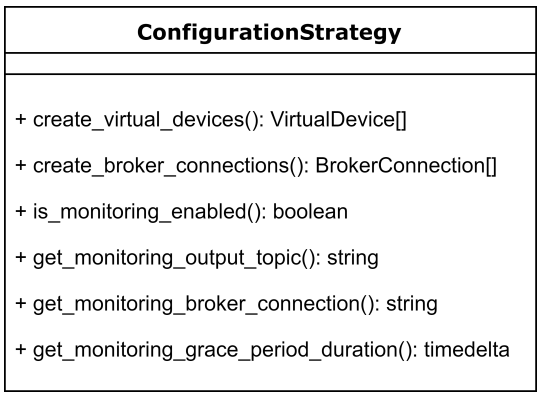

Abbildung 5.20.: UML-Diagramm der Konfigurationsschnittstelle

Diese sechs Schnittstellen, welche im UML-Diagramm in Abbildung 5.20 abgebildet sind, erlauben es, alle für die Anwendung notwendigen Konfigurationswerte zu definieren.

Name	Funktion
create_virtual_devices	Erstellt und konfiguriert die Virtual Device-Instanzen, inkl. Virtual Values und Virtual Functions
create_broker_connections	Erstellt und konfiguriert die Broker-Connection-Instanzen
is_monitoring_enabled	Definiert, ob Monitoring-Daten generiert und ausgegeben werden sollen
get_monitoring_output_topic	Definiert, in welchem Topic die Monitoring-Daten ausgegeben werden sollen
get_monitoring_broker_connection	Definiert, auf welcher Broker-Connection die Monitoring-Daten ausgegeben werden sollen
get_monitoring_grace_period_duration	Definiert, wie lange der Toleranzzeitraum sein soll

Tabelle 5.1.: Konfigurations-Schnittstellen

Da der Virtual Device Host als Konsolenanwendung implementiert wurde, kann die zu verwendende Konfiguration somit als Kommandozeilenparameter ausgewählt werden. Eine *Configuration-Factory* kann nun basierend auf der gewünschten Configuration-Strategy-Implementation diese instanziieren und der restlichen Anwendung zur Verfügung stellen.

Ein alternativer Ansatz wäre die Implementierung einer Möglichkeit zur automatischen Konfiguration gewesen, z.B. basierend auf Ontologien. Solch ein Vorgang hätte jedoch den Rahmen dieser Arbeit überschritten, wird aber im Kapitel *Ausblick* skizziert.

5.7. Monitoring

Im Rahmen der durchgeführten Analyse wurde das Thema der Ausfalltransparenz als sinnvolle und notwendige Funktionalität definiert. Eine Ausfalltransparenz kann durch ein Monitoring-System ermöglicht werden. Die primäre Funktion solch eines Monitoring-Systems ist es, darauf horchende Geräte darüber zu benachrichtigen, wenn Virtual Devices „beeinträchtigt“ sind, also die Konsistenz, Genauigkeit oder sogar Erreichbarkeit der Daten des Virtual Devices nicht mehr sichergestellt werden kann. Abbildung 5.21 zeigt einen solchen Fall, bei dem ein Sensor ausgefallen ist.

5.7.1. Veröffentlichung

Da bisher die gesamte Kommunikation via MQTT erfolgt, ist es sinnvoll, dies auch für die Monitoring-Funktionalität beizubehalten. Da eine out-of-band-Kommunikation, also eine Kommunikation über einen zusätzlichen Verbindungsweg, für Separierungszwecke sinnvoll

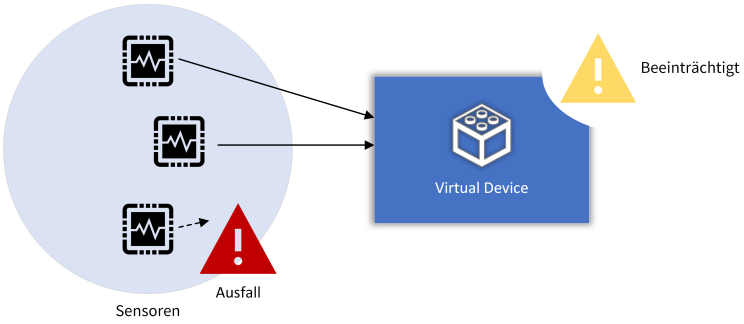

Abbildung 5.21.: Beispiel eines beeinträchtigten Virtual Device

sein kann, soll es jedoch möglich sein, diese über eine eigene Broker-Connection erfolgen zu lassen. Eine out-of-band-Kommunikation ist bspw. vonnöten, falls die Leistung oder Funktion der genutzten Geräte durch das Hinzufügen von weiterer (potenziell inkompatiblen) Kommunikation auf bestehenden Verbindungswegen gestört werden könnten.

Eine Alternative wäre die Integration in die vorhandene Kommunikation. Hierbei könnten die von den Virtual Value Groups ausgehenden Datenpakete den Status des jeweiligen Virtual Devices beinhalten. Da durch das Hinzufügen dieser Werte jedoch leicht Kompatibilitätsprobleme auftreten könnten, wurde stattdessen auf die out-of-band-basierte Kommunikation gesetzt.

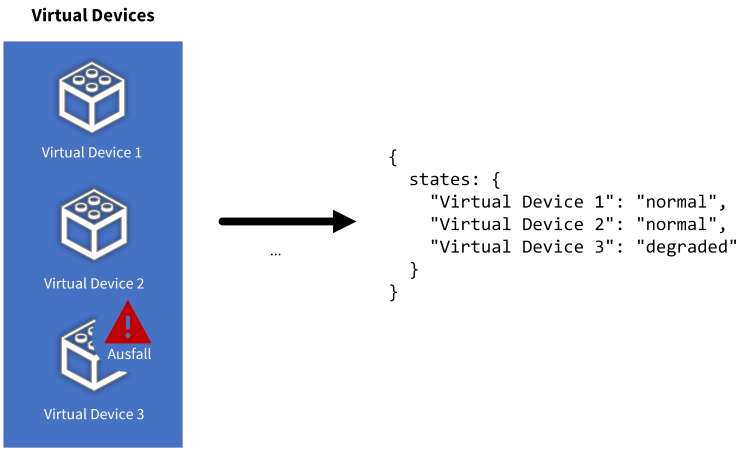

Abbildung 5.22.: Generierung des Monitoring-Datenpakets

Das generierte Monitoring-Datenpaket soll als JSON-formatierte Nachricht verschickt werden. JSON wurde hierfür ausgewählt, da es auf jeder Plattform weiterverwendet werden kann und als menschenlesbares Format auch ohne die Verwendung von Hilfsmitteln für

Diagnostikzwecke gelesen und interpretiert werden kann. Abbildung 5.22 zeigt ein Beispiel für ein generiertes Monitoring-Datenpaket für drei Virtual Devices, wovon eines beeinträchtigt (hier: degraded) ist.

5.7.2. Kalkulation

Es gibt unterschiedliche Varianten, wie entschieden werden kann, ob ein Virtual Device beeinträchtigt ist.

Konventionsbasierte Variante

Eine Lösung wäre es, vorauszusetzen, dass alle Input Data Sources mindestens einmal im Generierungszeitfenster der Virtual Value Group neue Informationen senden. Senden bspw. alle verwendeten Sensoren minütlich neue Daten und soll alle fünf Minuten ein Virtual Value generiert werden, so würde das betreffende Virtual Device als beeinträchtigt gelten, wenn mindestens einer der Sensoren in den fünf Minuten keine neuen Daten gesendet hat.

Policy-basierte Variante

Ein alternativer Ansatz wäre die Implementierung von einem Virtual Device zuweisbaren *Policies*, welche definieren, welche Bedingungen ein Virtual Device erfüllen muss, um als normal funktionierend zu gelten. Dieser Ansatz ist einer Regelmaschine sehr ähnlich.

Die im vorigen Lösungsansatz genannte Konvention könnte hierbei ein Beispiel für eine solche Policy sein. Ein weiteres Beispiel wäre eine Policy, welche die eingehenden oder ausgehenden Daten eines Virtual Device anhand von definierten Wertebereichen validiert, um z.B. einen durchschnittlichen Temperaturwert von -40 °C für einen normalen Büroraum als fehlerhaft zu erkennen.

Variantenauswahl

Da die Monitoring-Funktionalität kein Kernaspekt des Prototyps ist, wurde hier auf den Konventionsbasierten Ansatz gesetzt, da dieser in vielen Fällen bereits ausreicht und simpler zu implementieren ist.

5.7.3. Toleranzzeitraum

Nach einem Start hat die Anwendung noch keine Sensordaten empfangen können. Da diese Daten jedoch maßgeblich für die Kalkulation des Beeinträchtigt-Status verantwortlich sind, könnte hier fälschlicherweise ein Virtual Device als beeinträchtigt markiert werden, obwohl dies nicht der Fall ist, sondern die Daten einfach nur noch nicht empfangen wurden. Daher soll es einen Toleranzzeitraum geben, welcher nach dem Start der Anwendung abgewartet wird, ehe ein Virtual Device als beeinträchtigt markiert werden kann, um so die Zeit bis zum erfolgreichen Empfangen aller benötigten Daten zu überbrücken. Dieser Zeitraum soll im Rahmen der Konfiguration seitens des Benutzers definierbar sein, da dieser sehr von den verwendeten Geräten abhängt.

5.8. Kaskadierung

In komplexeren Anwendungsszenarien kann es dazu kommen, dass mehrere Virtual Device Hosts erforderlich werden, welche in Kombination miteinander, also hintereinander geschaltet, betrieben werden. Laut der Definition von [3] wird solch eine „Hintereinander-Schaltung“ auch als *Kaskadierung* bezeichnet.

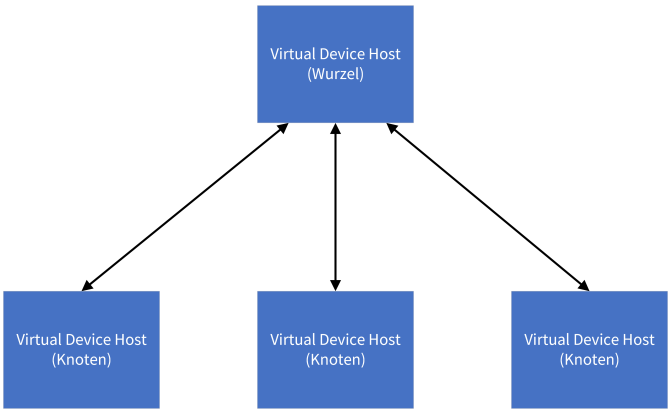

Abbildung 5.23.: Beispielschema eines kaskadierten Aufbaus

Ein Anwendungsszenario, bei dem eine solche Kaskadierung von Vorteil sein kann, ist die Ausstattung von mehreren Gebäuden. Da die Virtual Device Hosts möglichst latenzfrei an die Geräte angebunden werden sollen, kann es somit sinnvoll sein, dass jedes Gebäude mit einem dedizierten Virtual Device Host ausgestattet wird. Für die Kaskadierung bedarf es nun jedoch eines weiteren Virtual Device Host, welcher von allen anderen zugänglich sein

muss. Inwiefern hierfür ein oder mehrere MQTT-Broker verwendet werden und wo diese positioniert werden sollen ist hierbei offen und ist abhängig von den jeweiligen Leistungsanforderungen.

Durch diesen Aufbau, der in Abbildung 5.23 dargestellt ist, können granulare Virtual Devices mit geringer Latenz innerhalb der Knoten-artigen Virtual Device Hosts erstellt werden sowie Gebäude-übergreifende Virtual Devices im Wurzel-artigen Virtual Device Host. Prinzipiell lässt sich solch ein hierarchischer Aufbau beliebig um weitere Ebenen erweitern. Da die Kommunikation vollständig auf MQTT basiert und jeder Virtual Device Host vollkommen unabhängig agiert, gibt es hierbei theoretisch keine Limitierungen. Darüber hinaus muss solch ein Aufbau nicht zwingend unidirektional sein, sondern kann alternativ auch bidirektional aufgebaut sein.

5.8.1. Beispiel

Ein Beispiel für die Nutzung eines solchen kaskadierten Aufbaus ist in Abbildung 5.24 dargestellt. In drei Gebäuden gibt es jeweils eine Vielzahl an verbauten Sensoren sowie jeweils einen lokalen Virtual Device Host und MQTT-Broker. Zusätzlich gibt es ein viertes Gebäude – ebenfalls ausgestattet mit einem Virtual Device Host – welcher auf die erzeugten Daten der drei anderen Virtual Device Hosts der anderen Gebäude zugreift und diese in eigenen Virtual Devices weiter verwendet und somit anders aufbereitet darstellt.

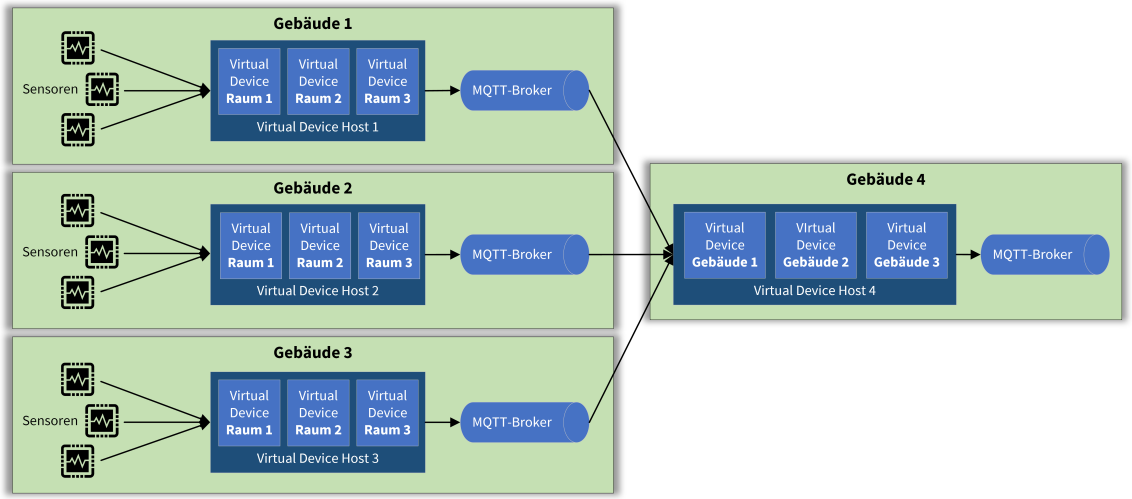

Abbildung 5.24.: Kaskadierung

Statt einem vierten Gebäude könnte hier auch die Cloud als Standort für den Wurzel-artigen Virtual Device Host genutzt werden. Hierbei könnten die sich in der Edge und zu den Sensoren lokal befindlichen Virtual Device Hosts die Sensordaten aufbereiten, filtern und zusammenfassen und so kompakter über den im Vergleich begrenzteren Verbindungsweg zu dem Virtual Device Host in der Cloud zur weiteren Verarbeitung übermitteln. Dies würde dann exakt dem typischen Anwendungsfall einer Edge-Lösung entsprechen.

6. Implementierung

Im Zuge dieses Kapitels sollen einige ausgewählte Themen, welche sich während der Implementierung ergaben, näher betrachtet werden. Dies schließt Herausforderungen ein, für die zunächst Lösungen gefunden werden mussten.

6.1. Verwendeter MQTT-Broker

Die implementierte Software setzt einen eingerichteten und funktionierenden MQTT-Broker voraus. Als MQTT-Broker wurde hierfür *mosquitto* von der Eclipse Foundation verwendet. Bei der auf dem Raspberry Pi verwendeten Betriebssystem-Distribution (Raspbian Stretch) konnte dieser mittels eines simplen apt-get Befehls über den lokalen Package Manager installiert werden und besaß bereits eine ausreichende Standardkonfiguration. Da jedoch keine erweiterten Funktionen von MQTT verwendet bzw. vorausgesetzt werden, ist davon auszugehen, dass die Anwendung mit jedem gebräuchlichen MQTT-Broker kompatibel ist. Dies ist beispielsweise wichtig, wenn erhöhte Performance-Anforderungen einen MQTT-Broker mit Cluster-Funktionalität für Scale-out-Szenarien erfordern, wie z.B. HiveMQ.

6.2. Integration als Hintergrunddienst

Im Rahmen einer produktiven und dauerhaften Nutzung ist es erforderlich, dass die Anwendung dauerhaft im Hintergrund laufen kann. Als primäre Plattform wurde der Raspberry Pi gewählt, wodurch eine Lösung für eine Linux-Umgebung gesucht wird.

Zwei beliebte Möglichkeiten, um dies einzurichten, sind einerseits die Einrichtung eines Cronjobs als auch die Definition als Dienst.

Ein *Cronjob* bezeichnet eine zeitgesteuerte Ausführung eines Befehls (z.B. zum Starten

einer Anwendung) in einer Linux-Umgebung, wie sie auf einem Raspberry Pi zu erwarten ist. Dies wird primär angewendet um zyklische Ausführungen zu organisieren, bspw. stündlich oder viertelstündlich einen spezifizierten Befehl auszuführen, kann jedoch auch angewendet werden um bei einem bestimmten Ereignis (bspw. dem Bootvorgang) eine Anwendung zu starten.

Alternativ kann hierfür ein *Daemon* implementiert werden. Vereinfacht gesagt sind Daemons in der Unix-Welt Hintergrundanwendungen, welche Dienste zur Verfügung stellen. Ein Daemon benötigt hierfür ein „init script“, welches u.a. definiert, wie die Anwendung gestartet oder gestoppt werden kann. Des Weiteren gibt dieses Skript an, zu welchem Zeitpunkt in dem Startprozess der Linux-Umgebung der Dienst gestartet werden soll. Im Falle des entwickelten Prototyps sollte der Dienst bspw. erst starten sobald alle Abhängigkeiten verfügbar sind, also sobald das Netzwerk initialisiert und der MQTT-Broker gestartet wurde.

Von den zwei Möglichkeiten ist die Zweite die sinnvollere, da diese die konventionellere Lösung für eine (pro Bootvorgang) einmalig zu startende und dauerhaft laufende Anwendung darstellt und darüber hinaus sicherstellt, dass alle Abhängigkeiten zum Startzeitpunkt verfügbar sind.

Im Abschnitt Analyse wurde Logging bereits als notwendige Funktionalität definiert. Da die normale Konsole bei einem Hintergrunddienst entfällt, bietet es sich an, Datei-basiertes Logging durchzuführen, um einen Einblick in die Programmausführung haben zu können. Für die Logging-Funktionalität wurde die in Python integrierte Logging-Funktionalität verwendet, welche bereits Features wie z.B. unterschiedliche Logging-Level mitbringt.

6.3. JSON-Format

Während der Implementierung zeigte sich, dass Python Probleme beim Parsen von JSON hat, wenn sich die jeweilige Zeichenkette nicht strikt an den JSON-Standard hält. Der Standard schreibt vor, dass jedwede Zeichenkette umschlossen von Anführungszeichen sein muss. Dies ist eine Regel, welche zumindest bei den Keys (von den Key/Value Tupeln von JSON) nicht immer eingehalten wird. Des Weiteren werden hierfür teilweise halbe

Anführungszeichen verwendet (also »‘« statt »“«), welche ebenfalls nicht dem JSON-Standard entsprechen. [22, S. 7-8]

Da es jedoch unrealistisch wäre, dies von jedwedem zu verwendenden Gerät zu verlangen, musste eine Lösung gefunden werden ohne die Geräte zu verändern. Dies konnte mittels eines regulären Ausdrucks implementiert werden, welcher den eingehenden String syntaktisch korrigiert ohne dessen Semantik zu ändern.

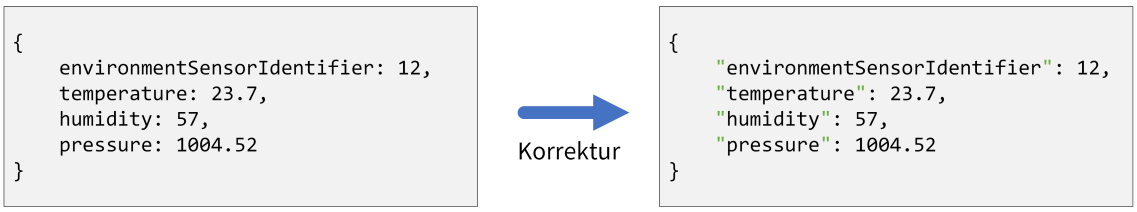

Abbildung 6.1.: JSON-Korrektur

6.4. Code-Styles

Da die angestrebte Lösung auch als Plattform für später darauf aufbauende Lösungen bzw. Erweiterungen gelten soll, ist es essenziell, dass der Sourcecode auch von anderen Personen gut lesbar ist und eine Einarbeitung schnell erfolgen kann. Basierend auf den Hinweisen der bekannten Werke Code Complete ([12]) und Clean Code ([11]) wurde während der Entwicklung insbesondere auf die Punkte Formatierung, Design Patterns und Konventionen geachtet. Wie dies erfolgt ist, wird nun in den folgenden Abschnitten näher erläutert.

6.4.1. Formatierung

Eine einheitliche Formatierung ist hilfreich, um Lesern des Sourcecodes durch eine vertraute Schreibart ein schnelles Verständnis zu erleichtern. Die Syntax von Python schreibt hier bereits sehr viele Regeln vor, wie der Sourcecode formatiert werden soll, wodurch eine einheitliche Formatierung sehr leicht umzusetzen war. Darüber hinaus kann somit auch sichergestellt werden, dass der spätere Benutzer bzw. Sourcecode-Leser exakt diese Formatierungsart erwartet.

6.4.2. Design Patterns

Im Rahmen der Entwicklung des Prototyps wurden drei Design Patterns angewendet:

Das Strategy-Pattern wurde bereits im Abschnitt *Lösungsansätze* genannt und erklärt. Es wurde an verschiedenen Stellen in der Anwendung für die Erhöhung von Anpassbarkeit und Erweiterbarkeit verwendet.

Für die Implementation der Konfigurierbarkeit wurde das Factory-Pattern angewendet, um die zu verwendende Konfiguration basierend auf einem Benutzerwunsch zu laden und diesen Vorgang von der restlichen Anwendung zu abstrahieren.

Für die lose Kopplung mehrerer Klassen wurde das Dependency Injection bzw. Inversion of Control Design Pattern angewendet. Hierbei wird die feste Kopplung gelockert, indem Abhängigkeiten der nutzenden Klasse bei der Initialisierung übergeben („injiziert“) werden und dies somit austauschbar bleibt.

6.4.3. Konventionen

Für eine leichtere Verständlichkeit des Codes wurden zwei elementare Konventionen verwendet: Jeder erweiterbare Aspekt wurde strikt nach der primären Funktion des Aspekts benannt: Eine Strategie für das Parsen von Daten wird somit als Parsing-Strategy bezeichnet. Jede hiervon abgeleitete Implementation trägt zusätzlich diesen Aspekt als Suffix in der Bezeichnung, bspw. Json-Parsing-Strategy. Dies kann zu längeren Titeln führen (bspw. Moving-Average-Aggregator-Strategy), jedoch ist hieraus direkt ersichtlich, dass es sich um eine Aggregator-Strategy handelt und hierbei der *Moving Average* errechnet wird. Eine Übersicht über die verschiedenen Strategy-Implementationen des entwickelten Prototyps sind in Abbildung 6.2 dargestellt.

Des Weiteren hat jede Strategie-Art ein spezifisches Verzeichnis (jeweils passend zu der Strategie benannt, bspw. „Aggregator-Strategies“), in der sich alle entsprechenden Strategien befinden sollen.

ParsingStrategies

Json**ParsingStrategy**
Raw**ParsingStrategy**
Xml**ParsingStrategy**

AggregatorStrategies

BasicFloatAverage**AggregatorStrategy**
MovingAverage**AggregatorStrategy**
MostCurrentSnapshot**AggregatorStrategy**
FloatMaximum**AggregatorStrategy**
FloatMinimum**AggregatorStrategy**
AllBooleanValuesEqualValue**AggregatorStrategy**

OutputGeneratorStrategies

Copy**OutputGeneratorStrategy**
StringReplace**OutputGeneratorStrategy**
RegexReplace**OutputGeneratorStrategy**
AddJsonProperty**OutputGeneratorStrategy**

GenerationStrategies

Timed**GenerationStrategy**
Continuous**GenerationStrategy**
OnDataReceived**GenerationStrategy**

PackagerStrategies

StringReplace**PackagerStrategy**
JsonArray**PackagerStrategy**

Abbildung 6.2.: Strategie-Konventionen

6.5. Temperatursensor

Die Temperatursensoren, welche als zusätzliche Sensor-Komponenten für den Prototypen entworfen und umgesetzt wurden, reichten aus, um bei der Nutzung stets genügend unterschiedliche Werte zu haben um bspw. Aggregationen durchführen zu können. Für einen produktiven Betrieb ergab sich jedoch das Problem, dass das Gehäuse zu dicht war, wodurch sich Wärme im Inneren des Gehäuses staute und die Temperaturwerte des Sensors kontinuierlich steigerte. Dieses Problem wäre mit einer Überarbeitung des Gehäuses lösbar, jedoch hat dies den Testablauf nicht beeinflusst, weswegen dies ignoriert werden konnte. Der fertige Temperatursensor mit abgenommener Abdeckungsplatte kann in Abbildung 6.3 betrachtet werden.

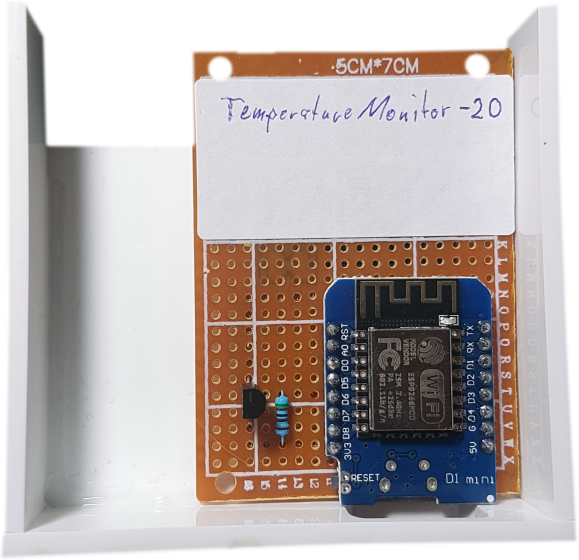

Abbildung 6.3.: Temperatursensor im geöffneten Gehäuse

7. Test

Im Rahmen dieses Abschnitts soll erklärt werden, wie die in Abschnitt 3 definierten funktionalen und nicht-funktionalen Anforderungen auf deren korrekte und ausreichende Umsetzung getestet wurden.

7.1. Testaufbau

Für den Test der Funktionsweise standen die folgenden Komponenten zur Verfügung:

- Raspberry Pi 3
- 3 Temperatursensoren
- 1 Umgebungssensor
- 1 Fenstersensor
- 1 Türsensor
- 3 LED-Strip-Controller
- Macbook Air (als Ausgabegerät)
- WLAN-Basisstation (für Netzwerk-Infrastruktur)

7.2. Funktionale Testszenarien

Der Prototyp soll für eine Vielzahl an unterschiedlichen Szenarien angewendet werden können, weswegen er möglichst flexibel konzipiert wurde. Dementsprechend sollen acht verschiedene Testszenarien durchgeführt werden, welche diese Flexibilität zeigen sollen.

Um den Anhang dieses Dokuments nicht unnötig mit mehreren Code-Dateien zu erschweren wurde darauf verzichtet die Konfigurationen anzuhängen. Diese sind jedoch im entsprechenden Konfigurationsverzeichnis des gebündelten Sourcecodes verfügbar.

7.2.1. Szenario 1: Float-Aggregation

Das erste Testszenario besteht aus drei Temperatursensoren sowie einem Umgebungssensor. Beide Sensoren veröffentlichen alle fünf Sekunden ein Datenpaket mit den jeweils ermittelten Sensorwerten eines Raums.

Hierfür soll ein Virtual Device erstellt werden, welches jeweils Virtual Values für den Durchschnitts-, Minimal-, Maximal- sowie Moving-Average-Wert führen soll. Diese Werte sollen alle 30 Sekunden als eine JSON-basierte Nachricht veröffentlicht werden.

7.2.2. Szenario 2: Boolean-Aggregation

Ein Gebäude (bzw. eine Wohnung) hat vier Fenster und zwei Türen, jeweils mit verbauten JarvisSuite-basierten Fenster- und Türsensoren. Diese Sensoren übermitteln den jeweils aktuellen „Geöffnet“-Status, sobald dieser sich ändert. Basierend auf diesen Werten soll alle fünf Minuten eine Nachricht mit virtuellen Werten veröffentlicht werden, welche angeben, ob alle Türen sowie alle Fenster geschlossen sind.

Hierfür ist eine Erweiterung des Sourcecodes vonnöten, da die vorhandenen Aggregator-Strategy-Implementationen mit den „Geöffnet“-Daten, welche den Wert „opened“ oder „closed“ annehmen können, nichts anfangen können.

7.2.3. Szenario 3: Boolean-Aggregation (On-Demand)

Das dritte Testszenario ähnelt dem zweiten Szenario, jedoch sollen nun diese Werte zu dem Zeitpunkt veröffentlicht werden, an dem eine potenzielle Änderung möglich sein könnte, also beim Empfang einer Statusnachricht einer der abstrahierten Sensoren.

Hierzu wird statt der *Timed-Generation-Strategy* stattdessen eine *On-Data-Received-Generation-Strategy* verwendet und der Test erneut durchgeführt.

7.2.4. Szenario 4: Verschachteltes JSON bzw. Hierarchie

Für das vierte Testszenario wurde einer der konstruierten Temperatursensoren mit einer alternativen Firmware versehen, welche den gemessenen Temperaturwert in einem komplexeren hierarchischen JSON-Dokument zur Verfügung stellt. Dies soll eine komplexe multifunktionale Sensorkomponente emulieren, deren Information nicht mittels trivialem Parsen weiterverwendet werden können. Das hierfür verwendete Format wurde bereits in Abschnitt 5.4.1 in Abbildung 5.7 vorgestellt.

7.2.5. Szenario 5: Kommando-Weiterleitung

Das fünfte Testszenario soll die Funktionalität von den Virtual Functions testen. Hierbei sollen drei LED-Strip-Controller als ein Virtual Device abstrahiert und mittels zweier Skripte an- und ausgeschaltet werden können. Die Skripte fungieren hierbei stellvertretend für einen Lichtschalter.

7.2.6. Szenario 6: Kommando-Weiterleitung mit Ersetzung

Im sechsten Testszenario soll das Szenario 5 erweitert werden. Statt Kopien der Auslösernachricht an verschiedene Topics zu versenden, soll diese Nachricht zunächst mittels regulären Ausdrücken verändert werden. Die durchgeführte Veränderung soll hierbei die Angabe der darzustellenden Farbe betreffen.

7.2.7. Szenario 7: Kaskadierung

Um die Kaskadierung zu demonstrieren, bedarf es mindestens zweier MQTT-Broker. Um dies zu ermöglichen, wurde eine Virtuelle Maschine in Windows Azure, Microsofts Cloud-Umgebung, erstellt und konfiguriert um einen nach außen erreichbaren MQTT-Endpunkt zur Verfügung zu stellen. Als Ausgangssituation werden erneut die Vorgaben von Szenario 1 verwendet, bis auf dass die Ausgabe auf dem zweiten Broker (und somit zu einem entfernten Endpunkt) stattfindet.

Dieser Aufbau ist in Abbildung 7.1 gezeigt. Die vier Sensoren senden ihre Daten zu einem lokalen MQTT-Broker. Ein lokaler Virtual Device Host nutzt diese Daten und veröffentlicht dessen virtuelle Werte auf einen entfernten MQTT-Broker, auf dem wiederum ein ebenfalls entfernter Virtual Device Host zugreift und virtuelle Werte kalkuliert.

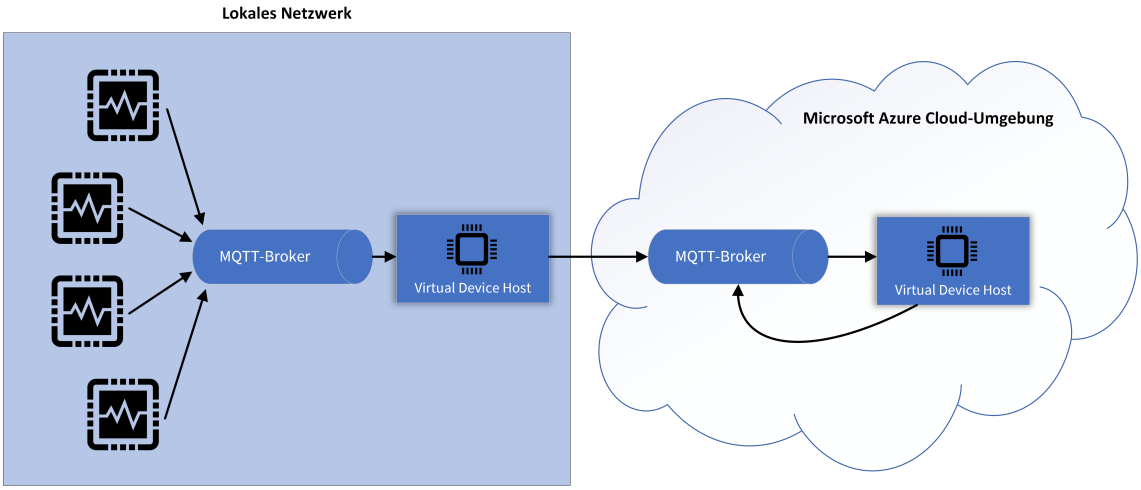

Abbildung 7.1.: Aufbau von Testszenario 7

7.2.8. Szenario 8: Monitoring

Das achte und letzte Testszenario soll das Monitoring-System testen. Hierfür wird der Aufbau des ersten Testszenarios verwendet und im laufenden Betrieb einer der Temperatursensoren vom Strom getrennt, um einen Ausfall zu simulieren. Ändert sich daraufhin der Zustand des Virtual Devices auf „Beeinträchtigt“, so ist der Test erfolgreich.

7.3. Nicht-funktionale Testszenarien

Im Rahmen der nicht-funktionalen Testszenarien wurden einige der in Analyse aufgestellten nicht-funktionalen Anforderungen auf deren ausreichende Umsetzung geprüft.

7.3.1. Erweiterbarkeit

Die Erweiterbarkeit einer Software kann am ehesten mit dem Versuch, solch eine Erweiterung durchzuführen, gezeigt werden. Daher wurde das dritte Testszenario so gewählt, dass eine Erweiterung hierfür vonnöten ist.

7.3.2. Verfügbarkeit

Die Verfügbarkeit kann getestet werden, indem ein „Anwendungscrash“ simuliert und anschließend überprüft wird, ob die Anwendung nach einem Neustart weiterhin ordnungsgemäß funktioniert.

Der Anwendungs-Prozess kann in einer Linux-Umgebung mittels des kill-Befehls terminiert werden. Die Service-Umgebung sollte den „Crash“ bemerken und automatisch die Anwendung erneut starten. Die Dauer, bis die Anwendung neugestartet wurde, sollte sich hierbei unter 30 Sekunden befinden.

8. Ergebnisbewertung

In diesem Abschnitt wird nun anhand der Ergebnisse des vorigen Abschnitts ermittelt, ob die entwickelte Lösung die in Analyse ausgestellten Anforderungen erfüllen kann.

8.1. Funktionale Ergebnisbewertung

Alle acht definierten Testszenarien konnten erfolgreich durchgeführt werden. Diese Testszenarien wurden gewählt, um einen großen Funktionsbereich der prototypisch entwickelten Anwendung abzudecken. Da alle Testszenarien das gewünschte Ergebnis erbrachten, kann die Anwendung als funktionell getestet und korrekt funktionierend angesehen werden.

8.2. Nichtfunktionale Ergebnisbewertung

Während der Entwicklung sowie während der Tests konnten auch die nichtfunktionalen Anforderungen auf deren adäquate und ausreichende Umsetzung geprüft werden.

8.2.1. Konfigurierbarkeit

Der Aspekt der Konfigurierbarkeit wurde bereits getestet, da im Rahmen der funktionalen Testszenarien acht verschiedene Konfigurationen erstellt wurden, zwischen denen beim Testen gewechselt wurde. Dementsprechend ist eine Konfigurierbarkeit gegeben und kann als getestet angesehen werden.

8.2.2. Erweiterbarkeit

Die entwickelte Lösung kann in den folgenden Aspekten erweitert werden:

- Aggregator-Strategy
- Generation-Strategy

- Parsing-Strategy
- Packager-Strategy
- Output-Generator-Strategy
- Configuration-Strategy

Ein Testszenario, dessen spezifische Anforderungen ursprünglich nicht vorgesehen wurden, konnte erfolgreich mittels einer einzelnen hinzugefügten Strategy-Klasse erweitert werden. Hierzu musste eine Python-Datei mit der jeweiligen enthaltenen Klasse in das Verzeichnis der entsprechenden Strategy-Art kopiert und in einer Konfigurationsdatei genutzt werden.

8.2.3. Geräteunterstützung

Im Rahmen der Testszenarien wurden sechs verschiedene Geräte mit dem Prototyp verwendet. Durch die Erweiterbarkeit ist theoretisch jedes Gerät anbindbar, solange die Kommunikation via MQTT erfolgen kann.

8.2.4. Skalierbarkeit

Durch eine Anwendungsarchitektur, welche auf die Verwendung von Threads für die parallele Verarbeitung der Virtual Devices setzt, kann die Anwendung problemlos vertikal skaliert werden. Die Threads verteilen sich hierbei auf die zur Verfügung stehenden CPU-Kerne.

Die horizontale Skalierbarkeit musste nicht getestet werden, da die Anwendung keine Abhängigkeiten oder infrastrukturelle Ansprüche hat, welche hier im Wege stehen könnten.

8.2.5. Verfügbarkeit

Nach einem absichtlichen sofortigen Beendens der Anwendung konnte die Anwendung ohne ersichtliche Probleme neugestartet und weiterverwendet werden. Hier zeigt es sich als praktisch, dass die Anwendung keinen persistenten Datenspeicher hat, der korrupt werden könnte.

8.3. Auswertung

Sowohl die Tests für die funktionalen Anforderungen als auch für die nichtfunktionalen Anforderungen konnten mit positivem Ergebnis abgeschlossen werden. Es wurde somit verifiziert, dass der entwickelte Prototyp die in der Analyse gefundenen Anforderungen hinreichend erfüllen kann.

9. Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung einer Edge-Lösung zur Virtualisierung von MQTT-basierten IoT-Sensoren. Diese Lösung sollte in der Lage sein, miteinander kommunizierende Geräte voneinander zu entkoppeln und so eine Anpassung der ausgetauschten Daten zu ermöglichen. Das Ergebnis der Arbeit sollte eine Plattform für darauf aufbauende Projekte ergeben, weswegen sich die Flexibilität, Erweiterbarkeit und Konfigurierbarkeit der zu entwickelnden Lösung als primäre Anforderungen ergaben.

Nach einer Analyse der Anforderungen und der Untersuchung mehrerer Lösungsansätze wurde daraufhin eine Software entworfen, welche physische Geräte (wie z.B. Sensoren und Aktoren) virtualisieren konnte. Sensordaten können hierbei aggregiert und projiziert werden sowie Kommandos – bei Bedarf modifiziert – weitergeleitet werden. Hierbei sollten sowohl kleine Szenarien, bestehend aus einzelnen Sensoren und Aktoren, als auch komplexere Szenarien, bestehend aus Dutzenden oder Hunderten an Geräten inklusive einer Kaskadierung der Recheneinheiten unterstützt werden.

Darüber hinaus wurde eine Monitoring-Funktion konzipiert und implementiert, welche es ermöglichte, Probleme mit den verwendeten physischen Geräten erkennbar zu machen. Dies war elementar für einen produktiven Einsatz, da durch die Verwendung von Virtualisierung eine Abstrahierung der physischen Geräte erreicht wird, wodurch ansonsten Fehler oder Ausfälle dieser Geräte nicht von Datenkonsumenten erkannt werden könnten.

Basierend auf durchgeführten Tests konnte festgestellt werden, dass der entwickelte Prototyp mehrere verschiedene Testszenarien, welche unterschiedliche potenzielle Anwendungsfälle abdeckten, erfüllen konnte. Insgesamt wurden alle Anforderungen an das Projekt hinreichend erfüllt, sodass es als Erfolg und zur produktiven Anwendung als tauglich angesehen werden kann.

10. Ausblick

Während der Entwurfs- und Entwicklungsphase ergaben sich mehrere Themengebiete und Ideen, welche der Funktionalität und Erweiterbarkeit der entwickelten Lösung zuträglich gewesen wären, jedoch nicht im Rahmen dieser Arbeit genauer betrachtet werden konnten. Innerhalb dieses Kapitels werden nun ausgewählte Thematiken angeschnitten.

10.1. Auto-Konfiguration durch Integration von Ontologien

Die in dieser Arbeit vorgestellte Lösung erfordert eine einmalige manuelle Konfiguration inklusive der expliziten Nennung aller verwendeten MQTT Topics. Dies ermöglicht eine hohe Flexibilität, bedeutet jedoch einen nicht-unerheblichen Aufwand bei einer größeren Gerätelandschaft und wird bei Veränderungen schnell unflexibel, da jedwede Veränderungen der Geräte und deren Einstellungen ebenfalls manuell in die Konfiguration eingetragen werden müssen.

Eine Maßnahme dem zu begegnen wäre die Integration einer Möglichkeit für eine automatische Konfiguration. Dies kann mittels veröffentlichter Metadaten der zu nutzenden Geräte erfolgen. Diese Metadaten können basierend auf Ontologien aufgebaut sein und somit die Funktionsweise und Menge der Schnittstellen eines Geräts semantisch interpretierbar darstellen. Somit könnte bspw. beschrieben werden, dass ein Gerät ein Sensor ist und einen Temperaturwert zur Verfügung stellt.

Auf dieser semantischen Wissensbasis aufbauend könnten nun Regeln definiert werden. Eine solche Regel könnte bspw. lauten, dass alle Sensorwerte von Sensoren, welche semantisch ein Temperatursensor sind und dieselbe Raumbezeichnung haben, aggregiert werden sollen.

Dies erlaubt es, dass sich das System basierend auf diesen Regeldefinitionen und der

vorhandenen Ontologie selbst konfiguriert und potenziell sogar auf Änderungen in der Gerätelandschaft (bspw. durch hinzukommende oder entfernte/ausgefallene Geräte) reagiert und neu konfiguriert.

Als erster Schritt wäre es hierfür notwendig, einen Algorithmus zu entwickeln, welcher diese Ontologie empfangen und interpretieren sowie diese Daten in das für den Prototyp entworfene Konfigurationsformat überführen kann. Perspektivisch wäre es auch möglich, das Konfigurationsformat als Zwischenschritt zu entfernen und direkt die Ontologieinformationen einzulesen, jedoch würde diese Änderung die Möglichkeit der manuellen Konfiguration unmöglich machen und fortan eine Ontologie zur Nutzung zwingend erforderlich machen.

10.2. Live-Konfiguration

Basierend auf dem aktuellen Stand des Prototyps bei Beendigung dieser Arbeit ist eine Änderung der Konfiguration der laufenden Virtual Devices stets mit einem Neustart der Anwendung verbunden. Dies hat somit natürlich einen negativen Einfluss auf die Verfügbarkeit.

Eine Möglichkeit, dies zu optimieren, wäre, den Status und die Menge der laufenden Virtual Devices zur Laufzeit anpassen zu können. Fünf Operationen wären hierfür sinnvoll:

- Virtual Device stoppen
- Virtual Device starten
- Virtual Device hinzufügen
- Virtual Device entfernen
- Virtual Device Konfiguration ändern

Hierzu wäre eine Schnittstelle notwendig, über die diese Operationen angestoßen werden können. Da die gesamte Kommunikation der Anwendung bisher auf MQTT basiert, bietet es sich an dies auch hierfür weiterzuführen. Alternativ wäre eine Kommunikation via HTTP möglich. Dies wäre im Rahmen von REST-Befehlen, einer Weboberfläche oder beidem implementierbar, je nachdem, ob der Benutzer direkt hierauf zugreifen können soll

oder eher indirekt (bspw. durch Skripte).

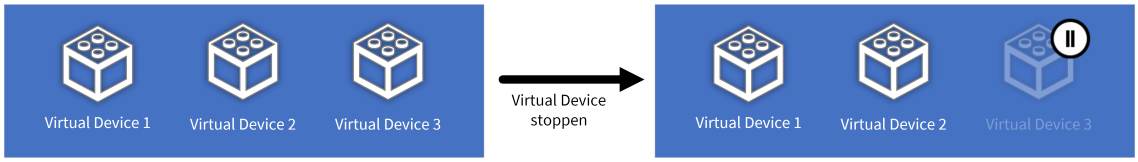

Abbildung 10.1.: Stop-Vorgang eines Virtual Devices

10.3. Persistenz

Derzeit besitzt der Prototyp keine Notwendigkeit, während der Laufzeit aufkommende (transiente) Daten über die Laufzeit hinaus zu persistieren. Dies ist akzeptabel, da es wahrscheinlich ist, dass die meisten Aggregator-Strategy-Implementationen innerhalb kurzer Zeit ausreichend Daten sammeln um wieder verlässliche Daten ausgeben zu können. Die Dauer dieser „Aufwärmphase“ ist abhängig von der verwendeten Hardware, Strategie und Konfiguration, aber dürfte in den meisten Fällen zwischen „sofort“ und „wenigen Minuten“ liegen. Senden die Sensoren, auf denen die Daten agieren, nur sehr selten Daten, so verlängert sich der Zeitraum entsprechend.

Dasselbe Problem ergibt sich bei sehr langlebigen Strategien. Soll eine Strategie bspw. einen täglichen Mittelwert generieren, so würde bei einem Neustart der Anwendung wahrscheinlich der Wert für den aktuellen Tag nicht generiert werden können, da alle bereits gesammelten Daten des jeweiligen Tages verworfen werden würden.

10.3.1. Ansatz

Alle Klassen des Prototyps sind prinzipiell statuslos konzipiert worden, ausgenommen der verschiedenen Strategy-Implementationen sowie der Input-Data-Source-Instanzen.

Als Teil der Persistierung sollte hierbei jedoch ein eindeutiger Identifikator der verwendeten Konfiguration gespeichert werden, um zu vermeiden, dass eine andere Konfiguration mit den (sicherlich inkompatiblen) gespeicherten Daten einer anderen Konfiguration versucht wird zu rekonstruieren. Das kann darüber hinaus noch erweitert werden, dass ein Hash der Konfiguration gespeichert wird, um sogar Änderungen erkennen zu können, welche

das erfolgreiche Laden verhindern könnten.

Persistierung von Strategy-Implementationen

Um den Zustand der Aggregator-Strategies beim Programmstart wiederherstellen zu können, dürfte es genügen, die Instanzen bei Programmbeendigung mittels Serialisierung zu persistieren und bei Programmstart entsprechend wieder zu deserialisieren.

Potenzieller Nachteil hierbei könnte sein, dass nicht alle Instanzvariablen wiederhergestellt werden sollen, sondern den beim Konstruktoraufruf initialisierten Wert behalten sollen. Dies lässt sich jedoch mittels einer neuen Konvention lösen, welche besagt, dass Instanzvariablen einer Aggregator-Strategy bei Bedarf in dessen prepare-Methode initialisiert werden sollen. Da diese chronologisch nach einer potenziellen Wiederherstellung aufgerufen werden würde, hat der jeweilige Entwickler der Strategy-Implementation die Freiheit zu entscheiden, ob der wiederhergestellte Wert behalten oder überschrieben werden soll.

Persistierung von Input-Data-Sources

Input-Data-Sources haben hauptsächlich ihre innere Warteschlange, welche persistiert werden könnte. Zum Zeitpunkt der Persistierung, also bei Programmbeendigung, würden sich in dieser Warteschlange ausschließlich Daten befinden, welche noch nicht von den Aggregator-Strategies verarbeitet werden konnten. Da jedoch nach erfolgter Programmbeendigung ohnehin jedwede neu aufkommenden Daten verworfen (bzw. garnicht erst empfangen) werden, ist der Verlust dieser Warteschlangen-Daten verkraftbar und würde nicht den Zusatzaufwand rechtfertigen, diese zu persistieren.

Literaturverzeichnis

[1] Alam, M. ; Nielsen, R. H. ; Prasad, N. R.: The evolution of M2M into IoT. In: *2013 First International Black Sea Conference on Communications and Networking (BlackSeaCom)*, 2013, S. 112–115

[2] Calin Dragos: *How to pick the best temperature sensor for your Arduino project.* – Homepage - Zugegriffen am: 02.08.2018 - Verfügbar unter `https://www.intorobotics.com/pick-best-temperature-sensor-arduino-project/`

[3] DATACOM Buchverlag GmbH: *kaskadierung :: cascading :: ITWissen.info.* – Homepage - Zugegriffen am: 30.08.2018 - Verfügbar unter `https://www.itwissen.info/Kaskadierung-cascading.html`

[4] dc-square: *MQTT Essentials Part 6: Quality of Service Levels.* – Homepage - Zugegriffen am: 01.09.2018 - Verfügbar unter `https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels`

[5] Fowler, Martin: *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002. – ISBN 978-0-321-12742-6

[6] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software.* mitp Verlag, 2014. – ISBN 978-3826697005

[7] Hersent, Olivier ; Elloumi, Omar ; Boswarthick, David: *M2M Communications: A Systems Approach.* John Wiley & Sons, 2012. – ISBN 978-1119994756

[8] Hohpe, Gregor ; Woolf, Bobby: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional, 2003. – ISBN 978-0-321-20068-6

[9] Lea, Perry: *Internet of Things for Architects*. Packt Publishing, 2018. – ISBN 978-1788470599

[10] Maged Michael, José E. Moreira, Doron Shiloach, Robert W. Wisniewski: Scale-up x Scale-out: A Case Study using Nutch/Lucene.

[11] Martin, Robert C.: *Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*. mitp, 2009. – ISBN 978-3826655487

[12] McConnell, Steve: *Code Complete - Deutsche Ausgabe der Second Edition*. Microsoft, 2005. – ISBN 978-3860635933

[13] Mehlei, Nicolas: *Entwicklung einer Heimautomationslösung mit Multi-System-Kompatibilität und Tablet-Interface*. 2016. – Verfügbar unter `https://www.mehlei.de/Content/Files/Nicolas-Mehlei-Projektarbeit-JarvisSuite.pdf`

[14] Mehlei, Nicolas: *Entwicklung von vernetzten Benutzungsschnittstellen für die Heimautomation*. 2016. – Verfügbar unter `https://www.mehlei.de/Content/Files/Nicolas-Mehlei-Projektarbeit-Lichtschalter.pdf`

[15] Mehlei, Nicolas: *Benutzerzentrierte Entwicklung eines Systems zur ortsensitiven Steuerung von Elementen eines Smart Homes*. 2018. – Verfügbar unter `https://www.mehlei.de/Content/Files/Nicolas-Mehlei-Forschungsprojekt.pdf`

[16] Merriam-Webster: *Projection — Definition of Projection by Merriam-Webster*. – Homepage - Zugegriffen am: 30.08.2018 - Verfügbar unter `https://www.merriam-webster.com/dictionary/projection`

[17] MQTT: *FAQ - Frequently Asked Questions — MQTT*. – Homepage - Zugegriffen am: 01.09.2018 - Verfügbar unter `http://mqtt.org/faq`

[18] Ravulavaru, Arving: *Enterprise Internet of Things Handbook*. Packt Publishing, 2018. – ISBN 978-1788838399

[19] Robert J. Smith: *Wide range of Hygrometers: DHT22, AM2302, AM2320, AM2321, SHT71, HTU21D, Si7021, BME280*. – Homepage - Zugegriffen am: 02.08.2018 - Verfügbar unter `http://www.kandrsmith.org/RJS/Misc/Hygrometers/calib_many.html`

[20] Starke, Gernot: *Effektive Softwarearchitekturen.* Carl Hanser Verlag GmbH & Co. KG, 2018. – ISBN 978-3446452077

[21] Tanenbaum, Andrew S. ; Steen, Maarten van: *Verteilte Systeme : Prinzipien und Paradigmen.* München u.a : Pearson Studium, 2008. – ISBN 978-38-2737293-2

[22] Tim Bray: *RFC 8259 - The JavaScript Object Notation (JSON) Data Interchange Format.* – Homepage - Zugegriffen am: 02.07.2018 - Verfügbar unter `https://tools.ietf.org/html/rfc8259`

[23] Vossen, Gottfried ; Haselmann, Till ; Hoeren, Thomas: *Cloud-Computing für Unternehmen.* dpunkt.verlag, 2012. – ISBN 978-3-89864-808-0

[24] Waher, Peter: *Mastering Internet of Things.* Packt Publishing, 2018. – ISBN 978-1788397483

Abbildungsverzeichnis

1.1. Klassischer Zugriff 2
1.2. Ansatz für entkoppelten Zugriff 3

2.1. Messaging Pattern: Warteschlangen 6
2.2. Messaging Pattern: Publish-Subscribe 6
2.3. Zustellungsgarantien 7
2.4. Edge-Computing Vergleich 9
2.5. MQTT-Aufbau 10
2.6. Strategy-Pattern 11
2.7. Regelbeispiel 12
2.8. Skalierungsarten 13
2.9. Umgebungssensor 14
2.10. Fenstersensor 15
2.11. LED-Strip-Controller mitsamt angebundener LED-Lichtleiste 15

3.1. Werteaggregation 17
3.2. Sensor-Entkoppelung 18
3.3. Ausfall einer Teilmenge von Sensoren 18
3.4. Ausfall aller Sensoren 19
3.5. Beispiel einer kombinierten Darstellung 20
3.6. Beispiel einer Projektion von Sensoren 21
3.7. Kommando-Verarbeitung 22
3.8. Kommando-Projektion für erhöhte Kompatibilität 23
3.9. Lastverteilung verschiedener physischer Aktoren 23
3.10. Horizontale Skalierung 26

4.1. Skizzierung des Regelmaschinen-Lösungsansatzes 29
4.2. Skizzierung des Strategy-Pattern-Lösungsansatzes 30

5.1. Aufbau 32
5.2. Kommunikationsverlauf von Virtual Devices 34
5.3. Vergleich der Subscribe-Verhaltensansätze 37
5.4. Wertentgegennahme unterschiedlicher Formate 37
5.5. Beispielnachricht von Umgebungssensor 38
5.6. Beispiel einer Wertadressierung 38
5.7. Beispielnachricht von komplexem Umgebungssensor 39
5.8. Pfad-Operationen 39
5.9. Wertegenerierung 40
5.10. UML-Diagramm der abstrakten Aggregator-Strategy 41
5.11. Gruppierung mittels Virtual Value Groups 41
5.12. Beispiel einer Wartezeitabweichung zwischen Ausführungen 42
5.13. Wasserfall-Effekt verketteter Virtual Values 43
5.14. Packaging-Beispiel 44
5.15. Generierungsprozess einer Virtual Value Group 45
5.16. Schema einer Virtual Function 46
5.17. Ablauf der Aufrufgenerierung mit drei Aufrufzielen 47
5.18. Aufrufgenerierung via regulärer Ausdrücke 48
5.19. Ablauf der asynchronen Verarbeitung 48
5.20. UML-Diagramm der Konfigurationsschnittstelle 49
5.21. Beispiel eines beeinträchtigten Virtual Device 51
5.22. Generierung des Monitoring-Datenpakets 51
5.23. Beispielschema eines kaskadierten Aufbaus 53
5.24. Kaskadierung 54

6.1. JSON-Korrektur 58
6.2. Strategie-Konventionen 60
6.3. Temperatursensor im geöffneten Gehäuse 60

7.1. Aufbau von Testszenario 7 64

10.1. Stop-Vorgang eines Virtual Devices 72

Tabellenverzeichnis

5.1. Konfigurations-Schnittstellen 50

Abkürzungen

BC BrokerConnection

GUID Globally Unique Identifier

IDS Input Data Source

IoT Internet of Things

JSON JavaScript Object Notation

M2M Machine-to-Machine

MQTT MQ Telemetry Transport

PIR Passives Infrarot

QoS Quality-of-Service

SPOF Single-Point-of-Failure

VD Virtual Device

VDH Virtual Device Host

VF Virtual Function

VV Virtual Value

VVG Virtual Value Group

Glossar

autonom Miteinander verbundene, *autonome* Systeme oder Komponenten, funktionieren unabhängig voneinander und können auch einzeln betrieben werden.

Inhärenz Eine inhärente Eigenschaft ist eine unveränderliche, essenzielle Eigenschaft.

Single Point of Failure Ein *Single Point of Failure* beschreibt eine Komponente eines Systems, deren Ausfall das gesamte System in einen nicht-betriebsfähigen Zustand bringen würde.

synthetische Daten Synthetische Daten sind Daten, welche keiner direkten Messung entstammen, sondern basierend auf anderen Daten erschaffen wurden.

Technologieagnostisch Etwas *Technologieagnostisches* funktioniert identisch, unabhängig davon welche Technologie verwendet wurde.

transient Transiente Daten sind Daten mit zeitlich begrenzter Lebensdauer.

User Experience Die User Experience beschreibt die Erfahrungen eines Benutzers bei der Interaktion mit einer Software oder einem Dienst.

A. Anhang

A.1. Inhalt der beiliegenden CD

- Verzeichnis **Masterarbeit-PDF**

  Diese Arbeit in digitaler Version als PDF-Dokument.

- Verzeichnis **Quellen**

  Alle Online-Quellen, welche im Literaturverzeichnis aufgeführt sind.

- Verzeichnis **Source-Code**

  Der Source-Code vom entwickelten Prototyp.

- Verzeichnis **Zusätzlicher Test-Code**

  Der Code der verwendeten Test-Geräte.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Ort, Datum	Unterschrift