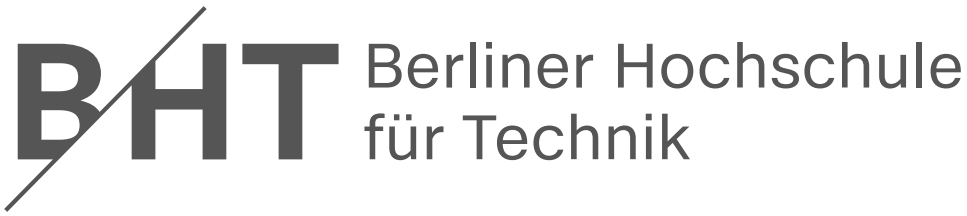

Fachbereich VI - Informatik und Medien
Studiengang Technische Informatik

Entwicklung eines Code Generators für die Variation der Kommunikationstechnologien von IoT Sensoren

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

René Heldmaier (Martikelnummer: 884975)
geboren am 03.07.90 in Tettnang

Betreuer: Prof. Dr. Edzard Höfig
Gutachter: Prof. Dr. Dragan Macos

Abgabetermin: 17.10.22

Vorwort

Die Welt befindet sich aufgrund des technologischen Fortschritts in einem steten Wandel. Von der Erfindung des Buchdrucks, über die Dampfmaschine bis zum Automobil, fand diese Entwicklung meist in Stufen statt, bei denen ein technologischer Durchbruch das zuvor undenkbare möglich machte.
Die Erfindung des Computers und der Aufbau des Internets im 20. Jahrhunderts stellt eine dieser „Stufen“ dar, die unser Leben so stark beeinflusst hat, das die Epoche, in der wir heute Leben, als Informationszeitalter bezeichnet wird. Das Internet of Things (IoT) hat das Potential, die nächste „Stufe“ in der (technologischen) Entwicklung der Menschheit zu werden.

Eidesstattliche Erklärung

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

_____

René Heldmaier

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

ATT	Attribute Protocol
BLE	Bluetooth Low Energy
GAP	Generic Access Profile
GATT	Generic Attribute Profile
IoT	Internet of Things
JSON	JavaScript Object Notation
MKP	Minimales Kommunikatiosprotokoll
OASIS	Organization for the Advancement of Structured Information Standards
RFID	Radio-frequency Identification
UUID	Universally Unique Identifier
Anwender	Softwareentwickler der mit dem Codegenerator eine IoT Anwendung entwickelt
Nutzer	Nutzer der IoT Anwendung

1 Einleitung

In vergangenen Jahrzehnten ließen sich enorme Fortschritte bei der Herstellung von Computern, in Bezug auf ihre Leistung, Kosten und Größe erzielen. So sind internetfähige Computer produzierbar, die so klein sind, wie ein Fingernagel, und deren Kosten im einstelligen Euro-Bereich liegen. Diese Weiterentwicklung ermöglicht es uns, „Dinge“ (engl. Things) in der realen Welt mit Sensoren und Aktoren auszustatten und mit dem Internet zu verbinden.
Etwaige Anwendungsgebiete finden sich in der Industrie, unter anderem im Bereich der Logistik („Smart Transport“) und der Steuerung von Industrieanlagen („Smart Factory“), aber auch im privaten Bereich wieder [7, 2793]. Für ihre Nutzer verspricht die als „Smart Home“ bezeichnete Steuerung von Wohnraum (Heizung, Licht, Lüftung, Einkäufe, etc.) sowohl mehr Komfort als auch die Möglichkeit, Energie zu sparen[23, 1267].

Das Internet of Things (IoT) ist derzeit in einer Wachstumsphase. So wurde prognostiziert, dass sich die Anzahl der IoT-Geräte, im Jahr 2023, im Vergleich zum Jahr 2018, auf 43 Milliarden nahezu verdreifachen wird. Im Jahr 2019 war bereits zu beobachten, wie die Anzahl der Unternehmen, die IoT-Technologien einsetzten von 13% (2013) auf 25% anstieg [15, 2]. Da alle diese Geräte auch Software benötigen, um ihre Aufgaben zu erfüllen, gewinnt das „Internet of Things“ auch in der Softwareentwicklung zunehmend an Bedeutung.

1.1 Problemstellung und Motivation

Um die Vernetzung und den Datenaustausch zwischen diversen IoT-Geräte zu ermöglichen, werden üblicherweise Kommunikationsprotokolle eingesetzt, die speziell für ihren jeweiligen Anwendungsfall optimiert wurden. Hierbei steht ein breites Spektrum an Kommunikationsprotokollen zur Auswahl (z.B. BLE, MQTT, CoAP, LoRa, etc.) die unterschiedliche Vor- und Nachteile aufweisen und sich teilweise miteinander ergänzen, teilweise jedoch auch zueinander in Konkurrenz stehen [1, 2350] [24, 555].
Das „Internet der Dinge“ ist ein breit gefasster Begriff, und die verwendeten Geräte können sehr unterschiedliche technische Eigenschaften besitzen, was sich auch in den zur Verfügung stehenden Kommunkationprotokollen widerspiegelt. So stellt ein Sensor zur Überwachung einer Ölpipeline mit Sattelitenverbinung an ein Kommunikationsprotokoll andere Anforderungenen als eine *Smart Watch*, die Herzfrequenz und Blutdruck des Trägers misst.
Auch in Bezug auf das OSI-Schichtenmodell werden von den Protokollen unterschiedliche Ebe-

nen definiert. Um ein paar Beispiele zu nennen:

1. Bluetooth Low Energy (BLE) beinhaltet eine Beschreibung aller OSI-Schichten (von der Bitübertragung bis zur Anwendungsschicht) [2]
2. MQTT definiert lediglich die Anwendungsschicht und verwendet für alle tieferen Schichten ein anderes Protokoll (üblicherweise TCP/IP) [19, 200]
3. LoRa beschreibt die physikalische Datenübertragung über Funk, und damit die unteren OSI-Schichten der Bitübertragung und Sicherung [11]

Bei der Entwicklung von Softwareanwendungen im IoT-Bereich kann diese Vielfalt eine erhebliche Komplexität verursachen, die nicht (direkt) im Zusammenhang mit der eigentlichen Anwendung steht. Dies wird auch als *accidential complexity* bezeichnet. Da die Komplexität nicht inhärent mit dem eigentlichen Problem zusammenhängt, wurde diese von Ingenieuren selbst geschaffen und kann auch von Ingenieuren beseitigt werden [9].

1.2 Ziel der Arbeit

Es soll ein Ansatz gezeigt werden, mit dem sich die, durch eine Vielzahl von Kommunikationsprotokollen entstandene, Komplexität bei der Softwareentwicklung von IoT-Anwendungen, reduzieren lässt. Hierbei kommt die *modellgetriebene Softwareentwicklung* zur Anwendung: Bei dieser wird zuerst ein Modell erstellt, dessen Abstraktion es ermöglichen soll, eine darunterliegende Implementierung zu verstecken. Aus diesem Modell soll, mit Hilfe eines Codegenerators, automatisch Quellcode erzeugt werden [20, 2.0].

Im hier vorgestellten Ansatz beschreibt das Modell die miteinander vernetzten IoT-Geräte und deren Eigenschaften. Der Codegenerator verwendet das Modell, um für jedes beschriebene IoT-Gerät ein Programmgrundgerüst zu erzeugen. Das Programmgrundgerüst ist bereits selbstständig lauffähig und enthält Code, der eine Kommunikation mit anderen Geräten ermöglicht. Ein Programmierer soll anschließend (an einer definierten Stelle) seinen eigenen Quellcode in den Programmgrundgerüsten einfügen. Der hier vorgestellte Codegenerator lässt sich daher auch als *Software Framework*[1] bezeichnen [20, 1.5].
Das Modell stellt demnach eine Abstraktionsebene zwischen dem Anwendungsentwickler und den verwendeten Kommunikationsprotokollen dar. Dies ermöglicht es, die darunter liegende Implementierung mit unterschiedlichen Kommunikationsprotokollen zu realisieren.

Eine „Variation“ der Kommunikationsprotokolle kann auf zwei unterschiedliche Arten erreicht

[1] Ein Software Framework bezeichnet eine wiederverwendbare, unvollständige Programmstruktur mit flexibler Anwendungsmöglichkeit, welche durch vom Anwender geschriebenen Code ergänzt werden muss, um sich in einem konkreten Programm einbauen zu lassen.

werden:

- Geräte, welche unterschiedliche Kommunikationsprotokolle verwenden, lassen sich miteinander vernetzen. Für den Anwender ist dies transparent, da er lediglich das *Modell* verwendet.
- Durch erneutes Ausführen des Codegenerators kann ein neues Programmgrundgerüst erzeugt werden, das mit einem anderen Kommunikationsprotokoll arbeitet als zuvor. Der bereits bestehende, vom Anwendungsentwickler manuell geschriebene Code, bleibt mit dem neuen Programmgrundgerüst kompatibel. Dies ermöglicht es das von einem Gerät verwendete Kommunikationsprotokoll zu ändern, ohne den vom Anwender geschriebenen Code zu modifizieren.

Im Zuge der vorliegenden Arbeit wird ein Codegenerator implementiert, der es erlaubt, den gewählten Ansatz anhand von Beispielprojekten auf realer Hardware zu untersuchen. Der Codegenerator soll (auf beide oben genannte Arten) eine Variation zwischen den Kommunikationsprotokollen MQTT und Bluetooth Low Energy (BLE) ermöglichen. Für den Testaufbau werden ESP32 Mikrocontroller verwendet, die eine Kommunikation über beide Protokolle, ohne zusätzliche Hardware, erlauben. Die erzeugten Programmgrundgerüste werden in der Programmiersprache C++ generiert und sind mit der „Arduino Platform“ kompatibel.

1.3 Aufbau der Arbeit

Der Kern der Arbeit lässt sich in die Kapitel: Konzept, Implementierung und Anwendung unterteilen, aus deren gesammelten Erkenntnissen die Ergebnisse hervorgehen.

Konzept

Im Konzept (Kapitel 3) wird zu Beginn das gewählte Modell beschrieben. Hierzu wird das *Datenmodell* in Abschnitt 3.1.1 definiert, welches festlegt, welche Eigenschaften die IoT-Geräte besitzen, und wie diese miteinander interagieren können. Die Beschreibung des Modells erfolgt über eine JSON-Datei, welche *Konfigurationsdatei* genannt wird. Der Aufbau der Konfigurationsdatei steht in Abschnitt 3.1.2. In Abschnitt 3.1.3 (Programmgrundgerüst) wird festgelegt, wie ein generiertes Programmgrundgerüst vom Anwendungsentwickler verwendet wird. Es wird geklärt, an welchen Stellen der manuell geschriebene Code eingefügt wird, und wie der Zugriff auf andere IoT-Geräte erfolgen kann.

Weitere daran anknüpfende Abschnitte des Konzepts befassen sich mit der Umsetzung des gewählten Modells, mit Hilfe der Protokolle MQTT und BLE.

Implementierung

Das Kapitel der Implementierung beschreibt die Arbeitsweise des Codegenerators, sowie Aufbau und Funktionsweise eines, vom Codegenerator erzeugten, Programmgrundgerüsts.
An ausgewählten Beispielen wird gezeigt, wie die Funktionsweise des Programmgrundgerüsts implementiert wurde. Ziel war es, zu zeigen, wie die im Datenmodell definierten *Attribute* und *Methoden* realisiert wurden. Außerdem wird erläutert, wie durch den Einsatz von C++-Template-Programmierung erreicht wird, die Generierung von C++ Quellcode (durch den Codegenerator) auf ein Minimum zu reduzieren.

Es wurden zwei Programme implementiert, die unabhängig vom Codegenerator arbeiten. Die „BLE-Bridge“ ermöglicht eine Verbindung zwischen MQTT und BLE Geräten, sowie eine Verbindung zwischen BLE-Geräten, die räumlich weit voneinander entfernt sind (außerhalb der Funkreichweite).
Eine als „Webinterface“ bezeichnete Webapp erlaubt es, über einen Browser auf die IoT-Geräte zuzugreifen und sie durch Funktionsaufrufe zu steuern. Von den Geräten übermittelte Daten werden in einer Datenbank abgelegt und visualisiert.

Anwendung

Anhand von drei Beispielprojekten wird der Codegenerator mit realer Hardware getestet, um die Funktionsfähigkeit des Konzeptes zu zeigen. Anhand der Beispiele werden auch Vor- und Nachteile des vorgestellten Ansatzes untersucht.
Das erste Testprojekt befasst sich mit der temperaturabhängigen Steuerung eines Fensters. Es werden zwei Versionen vorgestellt: In der ersten Version arbeiten alle Geräte mit MQTT. In der zweiten Version verwenden zwei Geräte BLE, wodurch ein gemischtes Netzwerk entsteht.
Auf diese Weise soll einerseits eine Variation der Kommunikationsprotokolle, bezogen auf die Kommunikation zwischen Geräten, gezeigt werden; Andererseits, dass es (ohne manuell Programmierung) möglich ist, das von einem Gerät verwendete Kommunikationprotokoll zu ändern.

Die beiden anderen Testprojekte (Steaming und Fragmentierung) befassen sich damit, wie mit dem vorgestellten Konzept größere Datenmengen übertragen werden können.

2 Grundlagen

2.1 Internet of Things

Der Begriff IoT wurde erstmals im Jahr 1999 von Kevin Ashton verwendet. Dieser bezeichnete damit Computer, die - ohne menschliche Eingabe - Informationen über ihre Umwelt sammeln. Das biete die Möglichkeit, Informationen über „alles worüber es etwas zu wissen gibt“ zusammenzutragen. „Dies könne die Welt ebenso stark verändern wie das Internet selbst.“ [6]

Heute findet der Begriff auch allgemeinere Verwendung. D. Guisto bezeichnet das Internet der Dinge als „neuartiges Paradigma“, bei dem „eine Vielzahl von 'Dingen' oder 'Gegenständen', wie Radio-frequency Identification (RFID), Sensoren, Aktoren, Mobiltelefone, welche, durch einzigartige Adressierungsschemata, dazu in der Lage sind miteinander zu interagieren und mit ihren benachbarten 'intelligenten' Komponenten gemeinsame Ziele zu erreichen“. [18]

Die Architektur des Internet of Things kann in fünf Schichten eingeteilt werden, die in Abbildung 2.1 dargestellt sind: [23, 1266]

- **Perception Layer:** Physikalische Geräte z.B. Sensor/Aktor
- **Network Layer:** Zuständig für die sichere Übertragung der Daten
- **Middleware Layer:** Verarbeitet Daten z.B. Speicherung in einer Datenbank
- **Application Layer:** Verwaltet die Anwendung abhängig von den Daten in der *Middleware*
- **Business Layer:** Verwaltet das komplette IoT-System. Erstellt Analysen basierend auf Informationen der vorherigen Schichten und sagt zukünftige Aktionen voraus

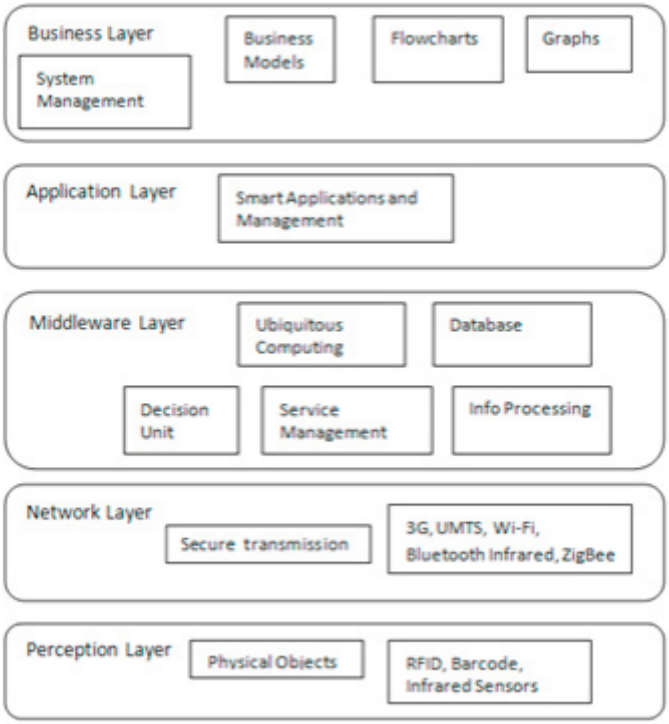

Abbildung 2.1: IoT Architektur [23, 1266]

2.2 Bluetooth Low Energy

Bluetooth Low Energy (BLE) wurde im Jahr 2010 von der „Bluetooth SIG“ als Teil der Bluetooth 4.0 Spezifikation veröffentlicht. Ziel war es, einen Funkstandard mit dem niedrigst möglichen Energieverbrauch zu entwickeln, der speziell für geringe Kosten, geringe Bandbreite, geringe Leistung und geringe Komplexität optimiert ist [34, 1].
Die Bluetooth Spezifikation behandelt dabei sowohl *Bluetooth* (der bekannte und seit vielen Jahren eingesetzte Standard) als auch *Bluetooth Low Energy* (der neue Standard). Die beiden Standards sind nicht direkt kompatibel, und Bluetooth Geräte, die auf einem Standard vor Bluetooth 4.0 basieren, können nicht mit BLE Geräten kommunizieren [34, 3].
Das BLE Protokoll umfasst dabei alle Schichten des OSI-Referenzmodelles, von der Bitübertragungs- bis zur Anwendungsschicht[2]. Abbildung 2.2 stellt den „BLE protocol stack“ dar.

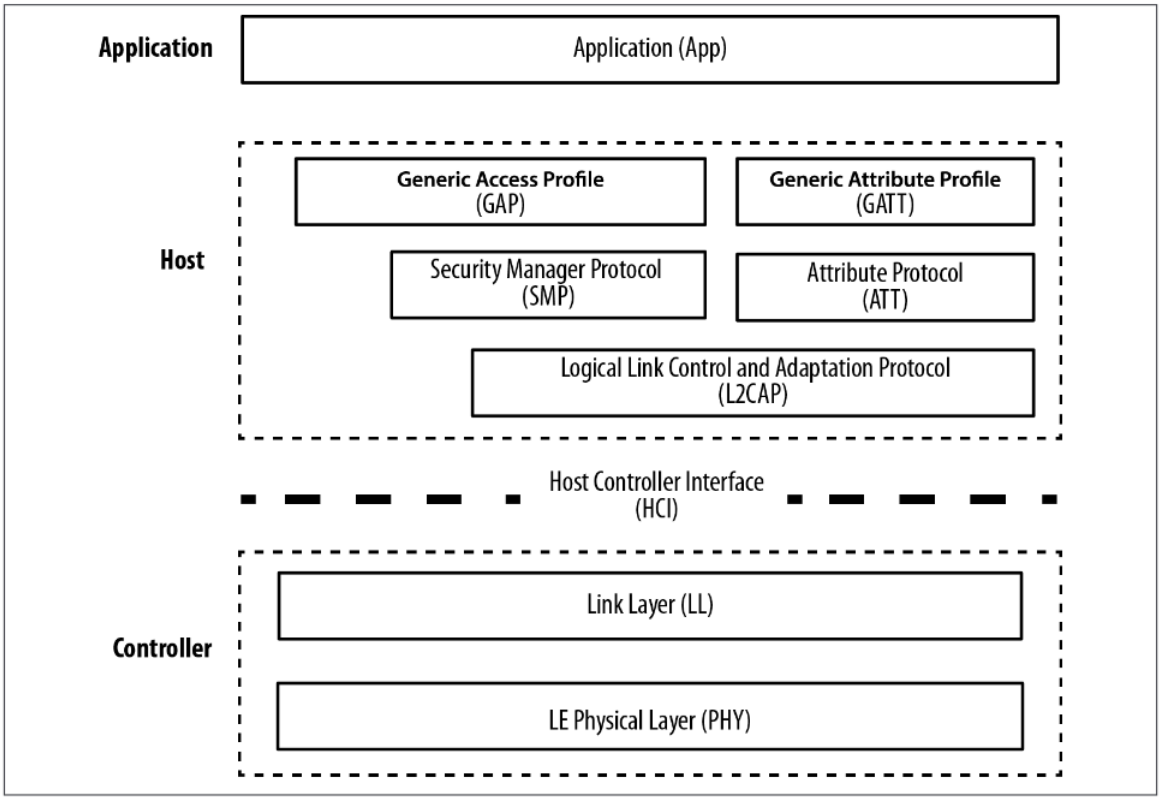

Abbildung 2.2: ble protocol stack [34, 16]

Da der BLE-Standard sehr umfangreich ist, sollen sich die Erläuterungen in diesem Abschnitt mit den Aspekten beschäftigen, die für das Verständnis dieser Arbeit relevant sind. Als wohl wichtigster Teil der BLE Spezifikation wäre hierbei das Generic Attribute Profile (GATT) zu nennen, welches auf dem Attribute Protocol (ATT) basiert. Einen weiteren wichtigen Aspekt bildet die Thematik, wie sich die (drahtlosen) Geräte finden können, um ein Verbindung aufzubauen, was zum Generic Access Profile (GAP) genannten Teil, des BLE-Protokolls, gehört.

2.2.1 Attribute Protocol

Das *Attribute Protocol* (ATT) ist ein zustandsloses Client/Server Protokoll, bei dem der Server „Attribute“ zur Verfügung stellt, auf die der Client lesend/schreibend zugreifen kann. Jedes Attribut besteht aus [34, 26] [34, 55]:

- **16-bit attribute handle:** Stellt eine Referenz auf das Attribut dar. Nachdem der Client die Referenz vom Server erhalten hat, kann sie zum Zugriff auf das Attribut verwendet werden
- **Universally Unique Identifier (UUID):** Eine einzigartige Zahl (engl. unique), die der Identifikation des Attributes dient

- **Permissions:** Eine Reihe von Berechtigungen, wie z.B. ob lesend oder schreibend zugegriffen werden darf
- **Value:** Für den Wert des Attributes sind maximal 512 Bytes an Daten vorgesehen, die beliebig interpretiert werden können.

Ein BLE Gerät kann Server, Client oder beides zugleich sein. Die Tabelle in Abbildung 2.3 stellt beispielhaft einen Server mit einer Reihe von Attributen dar.

Handle	Type	Permissions	Value	Value length
0x0201	$UUID_1$ (16-bit)	Read only, no security	`0x180A`	2
0x0202	$UUID_2$ (16-bit)	Read only, no security	`0x2A29`	2
0x0215	$UUID_3$ (16-bit)	Read/write, authorization required	"a readable UTF-8 string"	23
0x030C	$UUID_4$ (128-bit)	Write only, no security	`{0xFF, 0xFF, 0x00, 0x00}`	4
0x030D	$UUID_5$ (128-bit)	Read/write, authenticated encryption required	`36.43`	8
0x031A	$UUID_1$ (16-bit)	Read only, no security	`0x1801`	2

Abbildung 2.3: ble protocol stack [34, 56]

2.2.2 Generic Attribute Profile

Das *Generic Attribute Profile* (GATT) baut auf dem ATT auf und erweitert es um eine Hierarchie und ein Modell zur Abstraktion von Daten [34, 32]. Eine hierarchische Strukturierung der Attribute wird dadurch erreicht, dass sie sich in *Services*, *Characteristics* und *Descriptors* unterteilen lässt. Ein GATT Server enthält dabei *Services*, ein *Service* enthält *Characteristics* und eine *Characteristic* enthält *Descriptors*. Jedes dieser Elemente verfügt über die (unter ATT beschriebenen) Eigenschaften eines Attributes.
Abbildung 2.4 veranschaulicht die Strukturierung auf einem GATT Server.

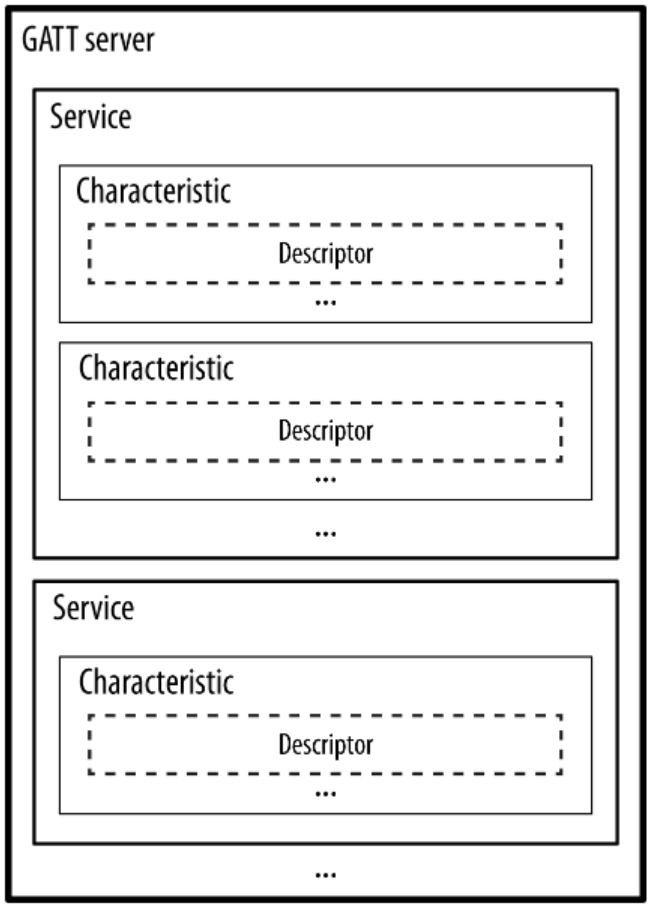

Abbildung 2.4: GATT Server [34, 57]

Anhand der Definition von „use-case-specific GATT Profiles“ wird für bestimmte Anwendungsfälle festgelegt, wie diese mit GATT realisiert werden sollen. So definiert z.B. das „Cycling Speed and Cadence Profile“ wie ein Sensor, der beim Fahrradfahren die Geschwindigkeit und die Trittfrequenz misst, diese über GATT anderen BLE Geräten zur Verfügung stellen kann. Dadurch kann der Sensor mit anderen Geräten oder Apps herstellerunabhängig zusammenarbeiten. [34, 13-14] Er muss lediglich den im „Cycling Speed and Cadence Profile“ definierten Service bereitstellen.

2.2.3 Generic Acces Profile

Das *Generic Acces Profile* (GAP) bestimmt Abläufe zur Steuerung der Interaktion zwischen Geräten. Dazu gehört z.B. das Veröffentlichen von Daten (engl. „broadcast“), das Auffinden von Geräten (engl. discovery) und der Verbindungsaufbau zwischen zwei Geräten. Zu diesem Zweck definiert das GAP unter anderem **Roles** (deutsch Rollen) und **Procedures** (deutsch Abläufe) [34, 33-38].

Das BLE Protokol besitzt vier Rollen, die ein Gerät im Netzwerk übernehmen kann. Dabei

kann ein Gerät auch mehrere Rollen gleichzeitig übernehmen. Die vier Rollen sind [34, 36-37]:

- **Broadcaster:** Sendet in regelmäßigen Abständen Datenpackete, die als „advertising packets“ (deutsch Ankündigungs-Pakete) bezeichnet werden
- **Observer:** Wartet auf den Empfang von „advertising packets“
- **Central:** Ein Gerät, das mehrere Verbindungen zu anderen Geräten aufrecht erhalten kann. Initiiert den Verbindungsaufbau. Entspricht im *Link Layer* der *Master* Rolle
- **Peripheral:** Entspricht der *Slave* Rolle des *Link Layer*

Ein häufig verwendeter Ablauf zum Verbindungsaufbau (der auch in dieser Arbeit Anwendung fand) wird als *General connection establishment procedure* bezeichnet[34, 42]. Hierbei agiert zunächst ein Gerät als *Broadcaster* und eines als *Observer*. Der *Observer* empfängt ein *advertising packet* und entscheidet, dass er sich mit dem *Broadcaster* verbinden möchte.
Der *Observer* initiert einen Verbindungsaufbau mit dem *Broadcaster*. Dadurch nimmt das Gerät, welches zuvor *Observer* war, die Rolle *Central* ein und das Andere die Rolle *Peripheral*.
Nachdem die Verbindung aufgebaut wurde, können beide Geräte über die im GATT definierten *Client/Server* Rollen Daten austauschen. Die *Central/Peripheral* Rollen sind dabei völlig unabhängig von den (im GATT definierten) Rollen *Client/Server*. Das heißt, ein Gerät kann GATT *Server/Client* sein, unabhängig davon ob es die Rolle *Peripheral* oder *Central* übernimmt. [34, 37]

2.3 MQTT

Die erste Version von MQTT wurde 1999 von Andy Standford-Clark und Alan Nipper entwickelt, um Ölpipelines, mit über Satteliten verbundenen Sensoren, zu überwachen [25, 96]. Ziel bei der Entwicklung des Protokolles war es, möglichst wenig Bandbreite zu verwenden und die Batterien der Sensoren zu schonen [10, 627]. Anschließend übernahm IBM die Entwicklung des Protokolls und veröffentlichte die Versionen 3.1 und 3.1.1 .
Seit 2013 wird MQTT von der Organization for the Advancement of Structured Information Standards (OASIS) weiterentwickelt und standardisiert. Diese veröffentlichte 2019 die derzeit aktuelle Version MQTT 5.0 [33].

Im weiteren Verlauf werden die - für diese Arbeit - wichtigsten Aspekte von MQTT beschrieben. Da MQTT ein umfangreiches Protokoll ist, sollen erneut nur die Teile erläutert werden, die für das Verständnis des vorgestellten Konzepts und dessen Implementierung relevant sind. Da in der Implementierung eine Software-Bibliothek verwendet wird, ist es zudem nicht notwendig, den genauen Aufbau von Message-Header und Kontrollnachrichten zu kennen.

2.3.1 Netzwerkarchitektur

Das MQTT Protokoll definiert zwei Arten von Teilnehmern im Netzwerk. Einen „message broker“ (deutsch „Nachrichten Vermittler“) und einen „client“ (deutsch „Klient“). Es wird das „publish/subscribe“ Patter verwendet, um Nachrichten zwischen den Clients zu übertragen.
Für die Datenübertragung wird üblicherweise das TCP/IP Protokoll verwendet. Es kann jedoch auch jedes andere Protokoll zum Einsatz kommen, das eine geordnete, verlustlose und bidirektionale Kommunikation ermöglicht [30, 1234-1235].

Broker

Der MQTT-Broker ist ein Server mit dem alle MQTT-Clients verbunden sind. Der Broker empfängt Nachrichten, die von den Clients veröffentlicht (engl. published) werden. Diese werden gefiltert und an andere Clients weitergeleitet, die sich für den Empfang der Nachrichten beim Broker abonniert (engl. subscribed) haben.[30, 1234-1235]

Client

Ein MQTT-Client ist ein Gerät, welches mit dem MQTT-Broker verbunden ist. Der Client kann z.B. ein IoT-Gerät, eine Webanwendung oder ein Mobilgerät sein. Zwei Rollen können hierbei übernommen werden: Der Client kann Nachrichten veröffentlichen (publish) oder sich für deren Empfang registrieren (subscribe). Auch das Übernehmen von beiden Rollen ist gleichzeitig möglich [30, 1234-1235].

Das publish/subscribe Pattern

Im Gegensatz zur klassischen Client-Server Architektur, bei der die Clients direkt mit anderen Clients kommunizieren, sind beim publish/subscribe Pattern Sender und Empfänger einer Nachricht voneinander entkoppelt. Die Kommunikation findet, wie in Abbildung 2.5 dargestellt, über den MQTT-Broker statt [30, 1234-1235].

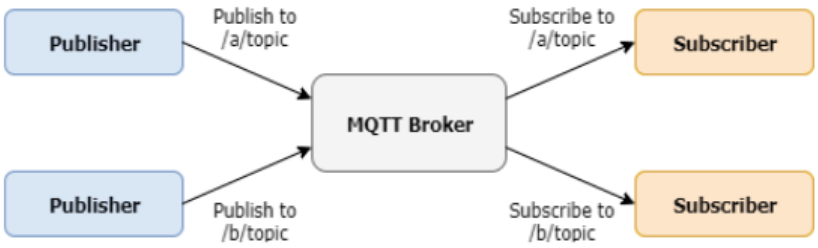

Abbildung 2.5: Publish/Subscribe Architektur Besipiel [30, 1234]

Topics

Alle MQTT-Nachrichten verfügen über ein *Topic* (deutsch Thema), welches eine Zeichenkette im UTF-8 Format ist [33, 55-56]. Dies ermöglicht eine Filterung der Nachrichten. Ein Client kann sich beim Broker für den Empfang von Nachrichten, mit einem bestimmten Topic, registrieren (*subscribe)*. Sobald ein anderer Client eine Nachricht zu diesem Thema veröffentlicht (*publish*), sendet der Broker die Nachricht zum ersten Client [30, 1234-1235].

Ein MQTT-Topic wird durch eine Zeichenkette (engl. string) repräsentiert, und kann - ähnlich dem Dateisystem von Computern - eine hierarchische Struktur besitzen. Dabei ist das Topic in „Level“ (deutsch Ebenen) unterteilt, die durch einen Slash („/“) getrennt sind [33, 98-100]. Ein Topic mit drei Leveln kann z.B. folgendermaßen aussehen: „*Sensoren/Wohnzimmer/Temperatur*“.

Durch die Verwendung von Leveln ist es möglich, mehrere Themen gleichzeitig zu abbonieren. So kann z.B. eine „multi level wildcard“ (repräsentiert durch das Raute Symbol „#“) verwendet werden, um alle Themen zu abbonieren, die mit einem bestimmten Topic beginnen [33, 98-100]. Im soeben genannten Beispiel können die Nachrichten von allen Sensoren mit dem Topic-String „*Sensoren/#*“ abboniert werden und die von allen Sensoren im Wohnzimmer mit „*Sensoren/Wohnzimmer/#*“.

2.4 Arduino

Arduino ist ein Unternehmen, welches sowohl elektronische Geräte als auch Software herstellt und designt [12]. Bekannt ist Arduino für die gleichnamige Arduino Platform, welche sowohl Hardware (Arduino-Boards) als auch Software (Arduino-IDE) Komponenten umfasst.
Bei einem Arduino-Board handelt es sich um einen Mikrocontroller, welcher, zusammen mit weiteren elektrischen Komponenten, auf einem Entwicklungsboard montiert ist. Die Schaltung auf dem Arduino-Board verfügt bereits über wichtige elektrische Kompnenten (wie Überspannungsschutz oder Spannungswandler). Über Steckverbinder (oder vorgeshene Lötösen) können die I/Os des Microcontrollers mit anderen Geräten verbunden werden.

Die Arduino-IDE ist eine Entwicklungsumgebung, die für das Programmieren der Arduino-Boards verwendet wird. Diese beinhaltet sowohl Cross-Compiler als auch eine Software, um das compilierte Programm auf den Microcontroller zu laden (engl. flashen). Über die Arduino-IDE können auch eine Vielzahl von Software-Bibliotheken installiert werden.

Ein besonderes Merkmal der Arduino-Platform besteht darin, dass die Hardware und Software möglichst „einfach zu verwenden“ ist [12]. Sowohl Hardware-Designs der Arduino-Boards als auch die Quellcodes der Arduino-Software werden unter einer Open-Source Lizenz veröffentlich [13]. Dies ermöglicht, dass andere Unternehmen mit der Arduino-Plattform kompatible Microcontroller-Boards anbieten. So wird z.B. in dieser Arbeit ein ESP32-Board des Herstellers

„esperrif“ verwendet.

2.5 ESP32

Beim ESP32 handelt es sich um eine Serie von 32-Bit Microcontrollern, die in unterschiedlichen Ausführungen (z.B. Single oder Dual Core) angeboten werden. Ein herausragendes Merkmal (und der Grund warum der ESP32 in dieser Arbeit verwendet wird) besteht darin, dass er über einen hybriden Wifi- und Bluetooth-Chip verfügt. Dadurch kann der ESP32 (ohne externe Geräte) sowohl über BLE als auch MQTT kommunizieren [16].

2.6 Raspberry Pi 3

Beim Raspberry Pi 3 handelt es sich um einen Einplatinencomputer. Die Platine besitzt ca. die Größe einer Handfläche und beinhaltet alle für einen Computer wesentlichen Komponenten. Zu diesen gehören unter anderem CPU, GPU (integriert in CPU), Arbeitsspeicher, Flash-Speicher (als eingesteckte SD-Karte), Wifi-Empfänger, Bluetooth-Empfänger, sowie Anschlussmöglichkeiten für externe Geräte (USB, HDMI, Ethernet, GPIO-Pins) [17].

Der hier verwendete „Raspberry Pi Model 3 B“ verfügt über folgende Spezifikationen:

- Prozessor: Quad Core 1.4GHz Broadcom BCM2837B0 64bit CPU
- Arbeitsspeicher: 1GB LPDDR2 SDRAM
- Betriebssystem: Raspbian GNU/Linux 11 (bullseye)

2.7 Jinja2

Jinja 2 ist eine *Template Engine*, die in der Programmiersprache Python geschrieben ist. Ein Template bezeichnet hier eine Textdatei (z.B. html) die, durch bestimmte Blöcke abgegrenzt, Variablen und Ausrücke enthält. Bei einem als *Rendering* bezeichneten Prozess wertet die *Template Engine* diese aus und ersetzt die Blöcke durch die, bei der Auswertung erhaltenen, Zeichenketten [28].

Die von Jinja2 verwendeten Blöcke innerhalb der Textdatei werden durch folgende Zeichenketten gekennzeichnet:

- {% ... %} - für Ausdrücke (engl. *Statements*)
- {{ ... }} - für Anweisungen (engl. *Expressions*)
- {# ... #} - für Kommentare

Ein Ausdruck ist üblicherweise eine Variable, die als Wert eine Zeichenkette enthält. Zu den möglichen Anweisungen gehören unter anderem Kontrollstrukturen wie *for loop* und *if/else*.

Abbildung 2.6 zeigt beispielhaft eine Jinja2 Template Datei. Die HTML-Datei enthält Jinja2 Template Blöcke mit denen der Titel der Website, sowie Links in der *Navigation Bar* dynamisch eingefügt werden. Dazu wird die Variable „title“ in Zeile 4 ersetzt und der *for-Loop* in Zeile 8-10 ausgewertet.
Das Rendern des Templates wird durch einen Aufruf der Jinja2 Python Bibliothek ausgeführt (dargestellt in Abbildung 2.7). Dazu wird als Parameter ein Python *Dictionary* übergeben, welches die vom Template verwendeten Variablen enthält (hier unter „title“ eine Zeichenkette und unter „navigation“ eine Liste aus *Dictionaries*).
Nach dem Ausführen der Template Engine enthält die Variable „renderedTemplate“ eine Zeichenkette, die das ausgefüllte Template enthält. Abbildung 2.8 zeigt, wie diese im hier vorgestellten Beispiel aussehen würde.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <title>{{ title }}</title>
5  </head>
6  <body>
7      <ul id="navigation">
8      {% for item in navigation %}
9          <li><a href="{{␣item.href␣}}">{{ item.caption }}</a></li>
10     {% endfor %}
11     </ul>
12
13     {# a comment #}
14 </body>
15 </html>
```

Abbildung 2.6: Jinja2 Template Beispiel

```
templateEnv = jinja2.Environment(loader=jinja2.FileSystemLoader("
    exampleDirectoryPath"))

templateParam = {
    "title" : "MyExampleWebpage",
    "navigation" : [
        {
            "href" : "www.google.com",
            "caption" : "Search␣with␣Google"
        },
        {
            "href" : "www.bing.com",
            "caption" : "Search␣with␣Bing"
        },
    ]
}

renderedTemplate = templateEnv.get_template("exampleTemplateFile.html").render
    (templateParam)
```

Abbildung 2.7: Jinja2 Python Beispiel - Rendering

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>MyExampleWebpage</title>
</head>
<body>
    <ul id="navigation">
        <li><a href="www.google.com">Search with Google</a></li>
        <li><a href="www.bing.com">Search with Bing</a></li>
    </ul>
</body>
</html>
```

Abbildung 2.8: Jinja2 Beispiel - Rendering Ergebnis

2.8 JSON

JavaScript Object Notation (JSON) ist ein Dateiformat, welches zum Datenaustausch zwischen Anwendungen und Programmiersprachen verwendet wird. Das JSON Dateiformat verwendet Textdateien, die sowohl von Menschen als auch Maschinen leicht gelesen und geschrieben weden können [22]. Der Standard des Dateiformats wird in den Dokumenten *RFC 8259* [8] und *ECM-404* [21] beschrieben. In JSON gibt es folgende Elemente [22]:

- **Objekt:** Ungeordnete Menge von *Name/Wert Paaren*. Wird durch geschweifte Klammern gekennzeichnet: { ... }

- **Array:** Geordnete Liste von Werten. Wird durch eckige Klammern gekennzeichnet: [...]
- **Wert:** Kann ein Objekt, Array, Zeichenkette, Zahl, „true“, „false“ oder „null“ sein.
- **Zeichenkette:** Eine Zeichenkette (*string*) aus Unicode Zeichen. Wird von doppelten Anführungszeichen umschlossen.
- **Zahl:** Kann Integer oder Fließkommazahl sein

3 Konzept

In den Abschnitten *Datenmodell*, *Konfigurationsdatei* und *Programmgrundgerüst* wird beschrieben, wie der Codegenerator aus Sicht des Anwenders (hier Softwareentwickler der eine IoT Anwendung erstellt) funktioniert und verwendet werden kann. Daran anknüpfend wird diskutiert, wie die Geräte-Kommunikation mit den gewählten Protokollen BLE und MQTT umgesetzt werden soll.

3.1 Codegenerator aus Sicht des Anwenders

Im *Datenmodell* sollen die Eigenschaften der IoT-Geräte und die möglichen Interaktionen zwischen den IoT-Geräten abgebildet werden, die sich vom Anwender gestalten lassen. Dies stellt eine Abstraktionsebene zwischen dem Anwender und den verwendeten Kommunikationsprotokollen dar. Zur Beschreibung des Modells erstellt der Anwender eine Konfigurationsdatei. In dieser werden alle im Netz vorhandenen Geräte, die nachfolgend fortlaufend als *Knoten* bezeichnet werden, und deren Schnittstellen aufgelistet.
Angelehnt an das, aus der objektorientierten Programmierung bekannte Modell von Klassen und Objekten, kann die Schnittstelle eines *Knotens* über *Attribute* und *Methoden* verfügen (die genauen Eigenschaften werden im Abschnitt 3.1.1 „Datenmodell“ beschrieben).

Durch Ausführen des Codegenerators erzeugt dieser aus der Konfigurationsdatei ein Programmgrundgerüst für jeden einzelnen *Knoten*. Bei dem Programmgrundgerüst handelt es sich um ein Arduino Projekt, in der Programmiersprache C++, für den ESP32 Mikrocontroller, welches bereits ohne Modifikation lauffähig ist.
Das Programmgrundgerüst verbindet sich selbstständig mit dem Netzwerk und stellt dem Anwender Code zur Verfügung mit dem - über die definierten Schnittstellen - auf andere Knoten zugegriffen werden kann.
Ähnlich wie bei der *objektorientierten Programmierung* wird nun eine Implementierung der, im Modell deklarierten, Schnittstellen benötigt. Dazu verfügt jedes Programmgrundgerüst über eine Datei mit dem Namen „CustomCode.cpp“, die leere Funktionsrümpfe enthält. Es ist vorgesehen, dass der Anwender hier seinen eigenen C++ Code einfügt. Abbildung 3.1 stellt dies graphisch dar.

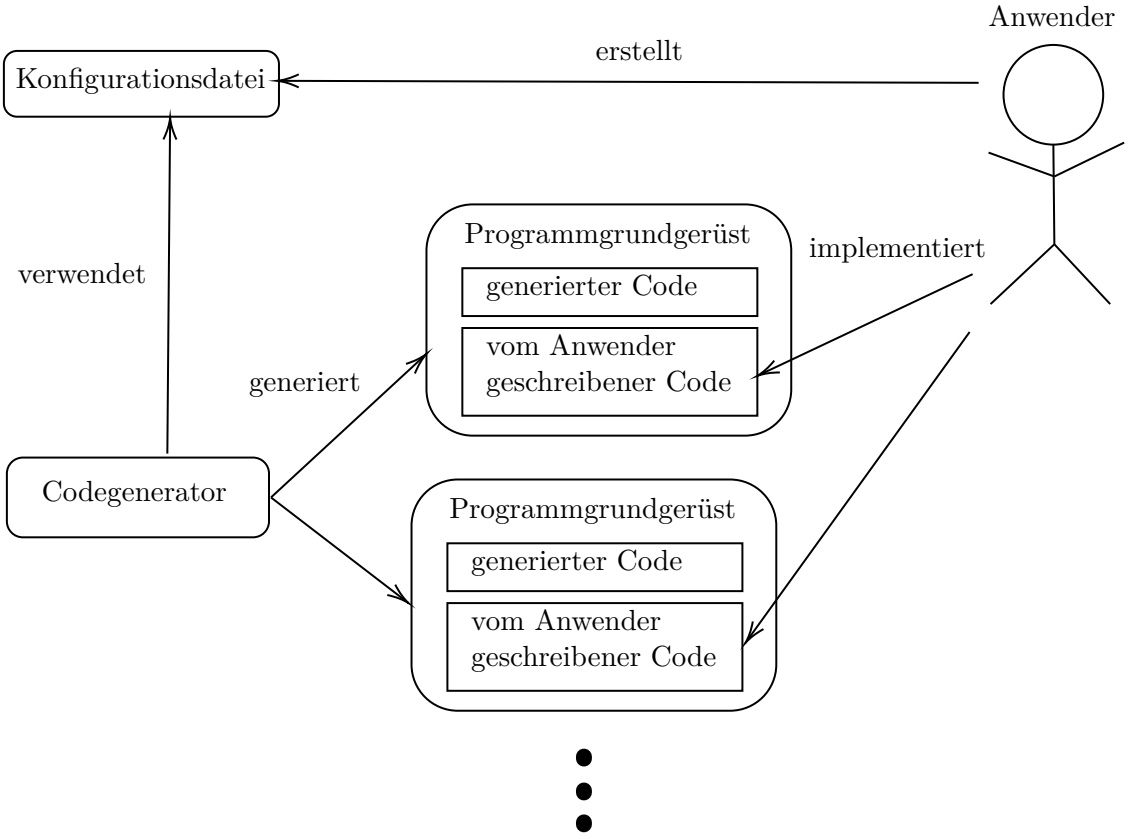

Abbildung 3.1: Verwendung des Codegenerators

Der Ablauf beim Erstellen einer IoT Anwendung - mit dem hier vorgestellten Ansatz - wird in Abbildung 3.2 dargestellt. Zuerst erstellt der Anwender eine Beschreibung des Modells in Form der Konfigurationsdatei. Aus dieser generiert der Codegenerator Programmgrundgerüste. Der Anwender vervollständigt die Programmgrundgerüste, indem er manuell C++ Code hinzufügt. Dadurch wird aus einem Programmgrundgerüst ein Programm. Die so erstellten Programme können mit Hilfe der Arduino IDE kompiliert und auf die Microcontroller geladen werden.

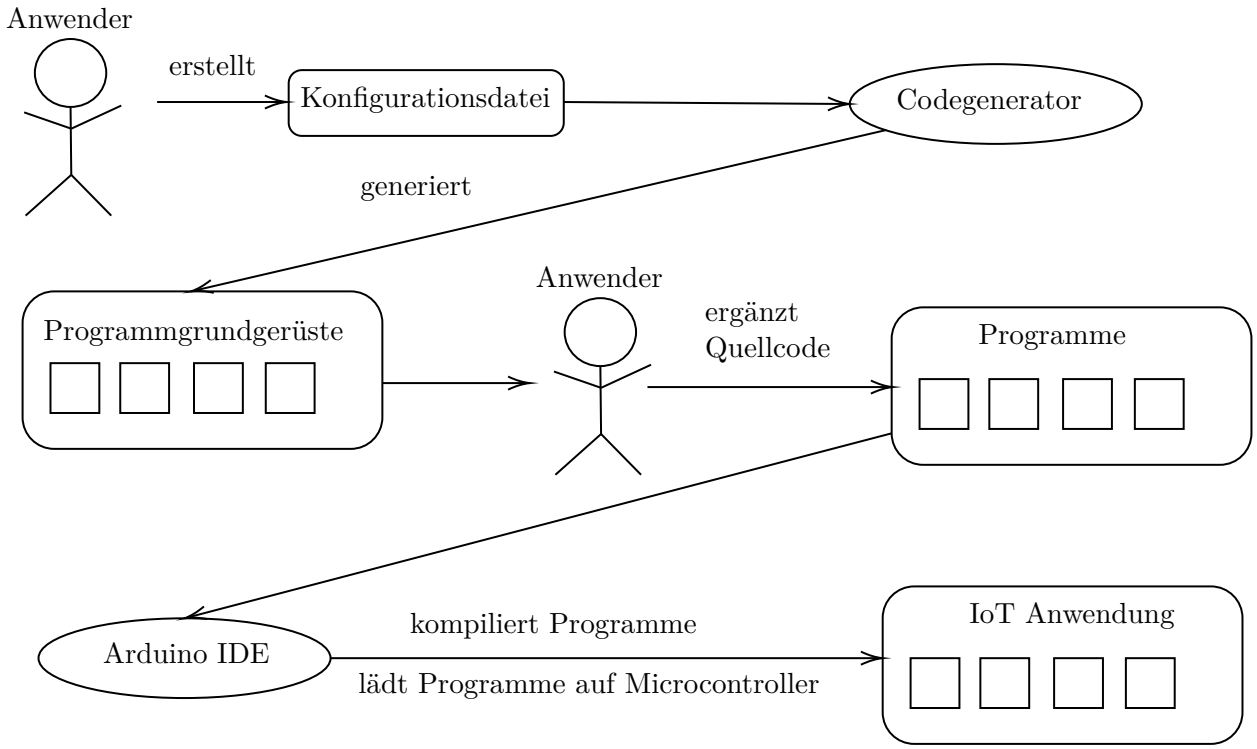

Abbildung 3.2: Erstellen einer IoT Anwendung

Da die Abschnitte *Datenmodell*, *Konfigurationsdatei* und *Programmgrundgerüst* abstrakt beschrieben sind, kann es für das Verständnis des Lesers hilfreich sein, nach jedem Abschnitt, das dazugehörige Beispiel im Kapitel 5.1 zu lesen. Die Beispiele gehen ausführlich auf diese drei wesentlichen Elemente ein und zeigen das Gelesene in praktischer Umsetzung.

3.1.1 Das Datenmodell

Das *Datenmodell* beschreibt das Modell der vernetzen IoT-Geräte, welches der Nutzer des Codegenerators beim Erstellen einer IoT-Anwendung verwendet. Das *Datenmodell* wurde so gewählt, dass es gewisse Ähnlichkeiten mit dem, in der Softwareentwicklung sehr verbreiteten, Konzept der *objektorientierten Programmierung* aufweist.

Im Gegensatz zur *objektorientierten Programmierung* werden hier jedoch nicht Objekte, im Sinne von Instanzen einer Klasse innerhalb eines Programmes beschrieben. Stattdessen beschreibt es Geräte, die über eine Netzwerkverbindung miteinander kommunizieren. Ein mit dem Netzwerk verbundenes Gerät wird auch als *Knoten* bezeichnet. Ähnlich wie bei der *objektorientierten Programmierung* verfügt jeder Knoten über eine Schnittstelle, die definiert, wie mit ihm interagiert werden kann.

Die Schnittstelle eines *Knotens* setzt sich aus *Attributen* und *Methoden* zusammen.

- Ein **Attribut** bezeichnet einen Wert, welcher von einem Knoten bereitgestellt wird. Andere Knoten können ausschließlich lesend auf das Attribut zugreifen. Es ist außerdem möglich, eventgesteuert auf die Änderung eines Attributes zu reagieren. Dazu kann ein Knoten das Attribut beobachten. Wenn ein neuer Wert zur Verfügung steht, wird dieser dem Beobachter automatisch übermittelt.
- Eine **Methode** bezeichnet eine blockierende Funktion. Diese wird von einem Knoten zur Verfügung gestellt und darf von anderen Knoten aufgerufen werden. Die Funktion kann über mehrere Parameter und einen Rückgabe-Wert verfügen. Wenn die Funktion von einem anderen Knoten aufgerufen wird, wartet dieser, bis die Funktion mit einem Rückgabewert geantwortet hat.

Für die Typisierung von Attributen und Methoden-Parametern können folgende C++ Datentypen verwendet werden:

- bool
- float, double
- alle ganzzahligen Datentypen, z.B. int, short, uint8_t ...
- Arrays mit std::vector<T>
- Strings mit std::string

Die Knoten bilden, abstrakt gesehen, ein Netzwerk, dessen Topologie vom Modell des Anwenders abhängt[2]. Damit ein Knoten auf Attribute und Methoden eines anderen Knoten zugreifen kann, muss dies in der Beschreibung angegeben werden. Abbildung 3.3 zeigt ein Modell mit drei Knoten, die jeweils ein Attribut und eine Methode besitzen. Folgende Zugriffe sind nach dem Modell möglich:

1. KnotenA darf auf Attribute und Methoden von KnotenC zugreifen
2. KnotenC hat keine Zugrifssrechte für KnotenA
3. Zwischen KnotenB und KnotenC ist ein Zugriff in beide Richtungen möglich
4. Es besteht keine Interaktionsmöglichkeit zwischen KnotenA und KnotenB

[2]Wie in Abschnitt 3.3 noch erläutert wird, findet in der Umsetzung jederzeit eine sternförmige Netzwerk Topologie Anwendung. Dies ist jedoch für den Anwender nicht sichtbar.

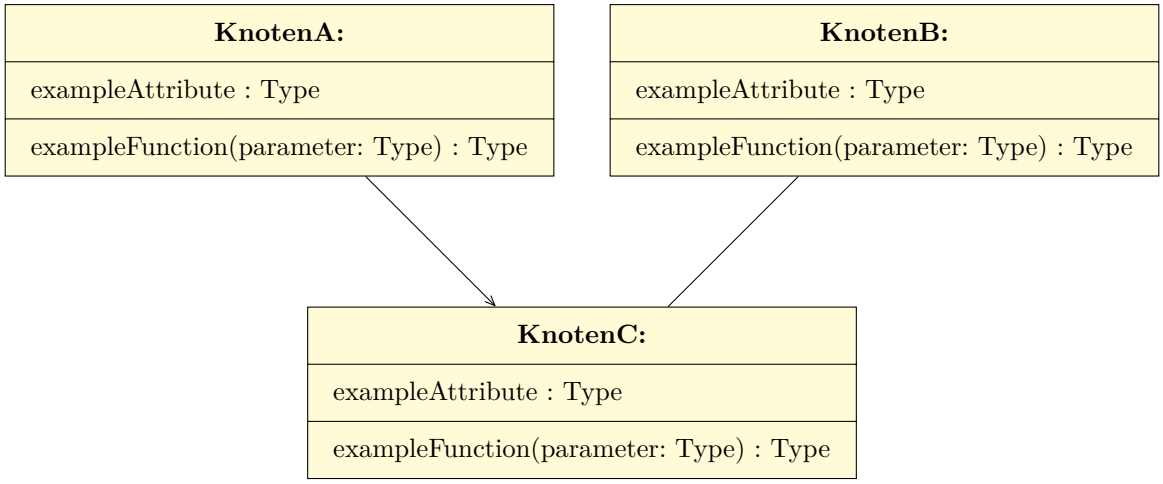

Abbildung 3.3: Modell mit drei IoT-Geräten

Da die Möglichkeit besteht, dass ein Gerät aufgrund eines Verbindungsfehlers nicht erreichbar ist, muss beim Lesen von Attributen oder dem Aufrufen von Methoden stets eine Fehlerbehandlung stattfinden.

3.1.2 Die Konfigurationsdatei

In der Konfigurationsdatei wird aufgelistet, welche *Knoten* es im Netzwerk gibt, über welche *Attribute* und *Methoden* sie verfügen, und wie sie sich mit dem Netzwerk verbinden (Kommunikationsprotokoll). Das Erstellen der Konfigurationsdatei kann als Teil der Programmierarbeit betrachtet werden, bei der das „Modell“ (Model Driven Development) der vernetzen Geräte beschrieben wird.

Für die Konfigurationsdatei wird das JSON-Dateiformat verwendet, da dieses für Mensch und Maschine einfach zu schreiben und zu lesen ist [22]. Der Aufbau einer Konfigurationsdatei sieht dabei wie folgt aus: Unter „nodes“ wird ein Array von Objekten abgelegt. Jedes Objekt enthält die Beschreibung eines Knotens mit folgender Strukturierung:

- „name“ enthält den Namen des Knotens. Dieser dient der eindeutigen Identifizierung und wird später u.a. als C++ Namespace im generierten Code verwendet.
- Unter „communication_protocol“ wird das vom *Knoten* zu verwendende Kommunikationsprotokoll, sowie für die Verbindung notwendigen Informationen angegeben. Bei MQTT sind dies beispielsweise das SSID und Passwort des Wlan Netzwerkes, bei BLE der Name des „BLE-Servers“.
- „variables“ beinhaltet alle zum Objekt gehörenden *Attribute*. Name und Typ des *Attributes* werden mit „name“ und „type“ festgelegt
- Unter „functions“ werden alle *Methoden* aufgelistet, welche der *Knoten* bereitstellt. Die Be-

schreibung einer *Methode* enthält den Namen („name“), sowie Funktionsparameter („params“), und den Typ des Rückgabewertes („returnType“).

- In dem Array „using“ werden alle anderen *Knoten* aufgelistet, auf deren *Attribute* und *Methoden* zugegriffen werden soll.
- Das Array „observe“ beschreibt alle *Attribute*, die beobachtet werden. Ein zu beobachtendes *Attribut* wird (wie später im C++ Code) mit der Schreibweise <Knotenname>::<Attributname> angegeben.

Außerdem enthält die Konfigurationsdatei unter „mqtt_broker“ und „ble_servers“ Informationen über den verwendeten MQTT-Broker, sowie die benötigten BLE-Server. Abbildung 3.4 zeigt den grundsätzlichen Aufbau der Konfigurationsdatei.

```
{
    "nodes" :
    [
        {
            "name" : "_",
            "communication_protocol" :
            {
                "name" : "BLE",
                "server" : "Server"
            },
            "variables" :
            [
                {
                    "name" : "_",
                    "type" : "_"
                },
                ...
            ],
            "functions" :
            [
                {
                    "name" : "setTolerance",
                    "params" :
                    [
                        {
                            "name" : "_",
                            "type" : "_"
                        }
                    ]
                    "returnType" : "_"
                },
                ...
            ],
            "using" : [
                "_",
                ...
            ],
            "observe" : [
                "<nodeName>::<variableName>"
                ...,
            ]
        },
        ...
    ],
    "mqtt_broker" :
    {
        "broker_address" : "_",
        "broker_port" : "_"
    },
    "ble_servers" :
    [
        {
            "name": "_"
        },
        ...
    ]
}
```


Abbildung 3.4: Aufbau der Konfigurationsdatei

3.1.3 Das Programmgrundgerüst

Wie bereits in Abschnitt 3.1 erläutert wurde, erzeugt der Codegenerator unter anderem die Datei CustomCode.cpp, die leere Funktionsrümpfe enthält. Im Gegensatz zu einer Software-Bibliothek, die Funktionen bereitstellt, welche vom Nutzer aufgerufen werden, sollen an dieser Stelle vom Nutzer Funktionen bereitgestellt werden, welche vom generierten Code aufgerufen werden. Dieser Ansatz ist in der Informatik auch unter dem Namen „Inversion of Control Prinziple“ bzw. „Hollywood Pattern“ („Don’t call us, we call you.“) bekannt [31].

Analog zu den bei Arduino-Projekten üblichen Funktionen „setup()“ und „loop()“, werden die Funktionen „void Setup()“ und „void Loop()“ erzeugt. „void Setup()“ wird dabei einmal beim Starten des Microcontrollers ausgeführt. Danach wird „void Loop()“ in einer Endlosschleife aufgerufen.
Außerdem werden die Funktionen „onConnect()“ sowie „onDisconnect()“ erzeugt, die bei Verbindung bzw. Verbindungsabbruch aufgerufen werden. Über die Funktion „isConnected()“ kann überprüft werden, ob eine Verbindung besteht oder nicht.
Über die „log“-Funktion übermittelt der generierte Code Log-Nachrichten. Diese werden durch das enum „Loglevel“ in „status“, „debug“ und „error“ unterteilt. Dadurch z.B. eine Fehlererkennung stattfinden, wenn eine Verbindung zwischen Gerät und Netzwerk nicht möglich ist.

Auf die Änderung eines *Attributes* kann eventbasiert reagiert werden. Um dies zu erreichen, kann in der Konfigurationsdatei beschrieben werden, dass ein *Attribut* eines anderen *Knotens* „beobachtet“ werden soll. Der Codegenerator erzeugt dann eine leere Funktion, die als Parameter den neuen Wert erhält.

```
1  // wird beim Start des Geraetes ausgefuehrt
2  void Setup()
3  {
4  }
5
6  // wird in einer Endlosschleife auf dem Geraet ausgefuehrt
7  void Loop()
8  {
9  }
10
11 // Geraet hat sich mit dem Server verbunden
12 void onConnect()
13 {
14 }
15
16 // Geraet hat Verbindung zum Server verloren
17 void onDisconnect()
18 {
19 }
20
21 // Log Nachrichten vom generierten Code
22 void log(const char* message, Loglevel::Loglevel loglevel)
23 {
24 }
25
26 // wird fuer jedes beobachtete Attribut erzeugt
27 void KnotenName::OnChange_AttributName(T newValue)
28 {
29 }
```

Abbildung 3.5: Programmgrundgerüst CustomCode.cpp

Im C++ Code erhält jeder *Knoten* einen „namespace“. Auf *Attribute* und *Methoden* kann folglich über <Knotenname>::<Attributname> zugeriffen werden. Für den Zugriff auf knoteneigene Attribute ist die Angabe des Namespace jedoch nicht erforderlich[3].

Attribute verfügen über eine „set“ und „get“ Methode, wobei *set* nur dem *Knoten* zur Verfügung steht, zu dem das *Attribut* gehört. Da das Lesen eines *Attributes* scheitern kann, wird für den Rückgabewert von *get* der Typ „std::optional<T>“ verwendet. Hier kann über „bool has_value()“ abgeprüft werden, ob der Wert erfolgreich gelesen werden konnte. Wenn „has_value“ *true* zurück gibt, kann über „T value()“ der eigentliche Wert erhalten werden.
Bei Methodenaufrufen muss immer ein „Timeout“ verwendet werden, da ansonsten der Fehlerfall zum endlosen Blockieren des Codes führen würde. Wird kein Timeout angegeben, wird ein Wert von 3 Sekunden verwendet. Für den Rückgabewert wird, wie auch bei den *Attributen*, der Datentyp „std::optional<T>“ verwendet. Wenn nach Ablauf des Timeouts keine Antwort

[3]Dadurch ist es möglich, dass Knoten mit der gleichen Funktionalität (wie z.B. Temperatursensoren) die exakt gleiche CustomCode.cpp Datei verwenden können.

erhalten wurde, gibt „std::optional<t>::has_value()“ *false* zurück.

Es wurde diese Art der Fehlerbehandlung gewählt, da Exceptions einen recht hohen Overhead erzeugen, und bei der Programmierung von Microcontrollern üblicherweise nicht verwendet werden. Im Vergleich zur Verwendung eines boolschen Flags, ermöglicht die Verwendung des „optional“ Datentypes eine leserlichere Gestaltung des Quellcodes.

```
1
2  void Loop()
3  {
4      if (isConnected())
5      {
6          // Lesen eines Attributes
7          std::optional<T> readAttrib = KnotenName::AttributName.get();
8          if (readAttrib.has_value())
9          {
10             // lesen des Attributes war erfolgreich
11             T value = readAttrib.value();
12         }
13
14         // Schreiben eines Attributes.
15         // Da es zu diesem Knoten gehoert muss Namespace nicht
               ↪ angegeben werden
16         AttributName.set(...);
17
18         // Methodenaufruf
19         std::optional<T> result = KnotenName::MethodenName(...,
               ↪ timeout);
20         if (result.has_value())
21         {
22             // Aufruf war erfolgreich
23             T returnValue = result.value();
24         }
25     }
26     else
27     {
28         // Fehlerbehandlung. Geraet hat keine Verbindung zu Server
29     }
30 }
```

Abbildung 3.6: Programmgrundgerüst CustomCode.cpp

3.2 Realisierung mit BLE und MQTT

Alle nun folgenden Abschnitte des Konzepts befassen sich mit der Realisierung des *Datenmodells* (3.1.1) mit den Kommunikationsprotokollen BLE und MQTT. Zuerst wird das *minimale Kommunikationsprotokoll* (MKP) 3.3 definiert. Dieses stellt die Minimalanforderung an ein Kommunikationsprotokoll dar, die erfüllt werden müssen, damit das *Datenmodell* realisiert werden

kann. Es wird gezeigt, wie sich das *minimale Kommunikationsprotokoll* mit MQTT und BLE umsetzen lässt. Anschließend wird dargelegt, wie die, im *Datenmodell* definierten, *Attribute* und *Methoden* mit dem *minimalen Kommunikationsprotokoll* realisiert werden können.

3.3 Das minimale Kommunikationsprotokoll

Um das soeben beschriebene Datenmodell mit einem IoT-Kommunikationsprotokoll umzusetzen, muss das Protokoll gewisse Minimalanforderungen erfüllen. Wie bei der Suche nach dem kleinsten gemeinsamen Teiler zweier Zahlen, werden Eigenschaften gesucht, die möglichst vielen IoT-Kommunikationtsprotokollen gemein sind, und die gleichzeitig die Realisierung des beschriebenen *Datenmodells* ermöglichen. Außerdem sollten möglichst die Vorteile der Kommunikationsprotokolle erhalten bleiben. Diese wurden (unter anderem) dazu entwickelt, ein Vernetzen von Geräten mit beschränkten Resourcen (z.B. Microcontroller) effizient zu ermöglichen.

Es wird im Folgenden ein Kommunikationsprotokoll definiert, welches als Minimales Kommunikatiosprotokoll (MKP) bezeichnet wird:

- Alle Knoten sind mit einem zentralen Gerät, dem *MKP-Server*, verbunden
- Der *MKP-Server* stellt *MKP-Datenfelder* bereit, auf welche die Knoten mit Hilfe eines eindeutigen Bezeichners (engl. Identifier oder ID) lesend und schreibend zugreifen können.
- Ein *MKP-Datenfeld* besteht aus einem eindutigen Bezeichner (*MKP-Datenfeld-ID*) und den dazugehörigen *Daten*.
- Die maximale Größe eines *MKP-Datenfeldes* ist begrenzt. D.h. es steht eine begrenzte Kapazität (in Bytes) für die *Daten* zur Verfügung
- Ein Knoten kann sich beim Server registrieren, um ein *MKP-Datenfeld* zu beobachten. Wenn ein anderer Knoten schreibend auf das Feld zugreift, wird dies dem Beobachter mitgeteilt. Dies entspricht dem *Publish/Subscribe-Pattern*, welches in 2.3.1 kurz dargestellt wurde.

Daraus resultiert eine sternförmige Netzwerk-Topologie, die in Abbildung 3.7 dargestellt ist. Die sternförmige Netzwerk-Topologie besitzt jedoch den Nachteil, dass beim Ausfall des Servers das gesamte Netz zusammenbricht. Einer seiner Vorteile besteht darin, dass die Anforderungen an die Clients sehr gering sind. Diese können sich (wenn gewünscht) nur sporadisch mit dem Netz verbinden, um Werte zu lesen oder zu schreiben (z.B. bei batteriebetriebenen Geräte). Da alle Werte über den *MKP-Server* ausgetauscht und auf diesem gespeichert werden, können Geräte miteinander interagieren, die nicht zum gleichen Zeitpunkt mit dem Netzwerk verbunden sind. Die sternförmige Netzwerktopologie ist auch bei IoT-Kommunikationsprotokollen sehr verbreitet. So wird sie unter anderem von BLE und MQTT verwendet (wobei bei BLE auch vermaschte Netze möglich sind).

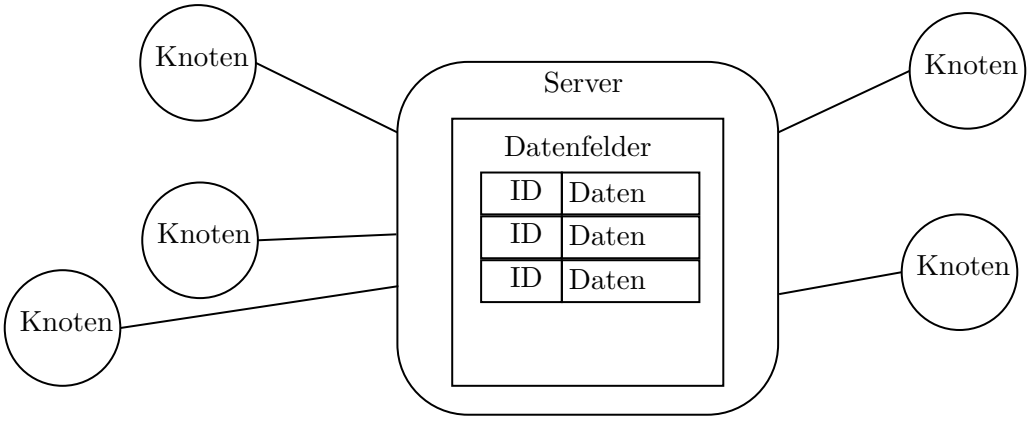

Abbildung 3.7: Netzwerk Topologie

Im Folgenden wird gezeigt, wie sich mit dem gewählten minimalen Kommunikationsprotokoll die Eigenschaften des beschriebenen *Datenmodells* umsetzen lassen, und dass die Protokolle BLE und MQTT mit dem MKP kompatibel sind.

3.4 Umsetzung mit MQTT

Der „MQTT-Broker“ verfügt über alle Eigenschaften, die notwendig sind, um den *MKP-Server* abzubilden. Für die Umsetzung mit MQTT kann hierbei ein beliebiger MQTT-Broker verwendet werden. Ein *MKP-Datenfeld* entspricht einer MQTT-Nachricht, deren „Topic“ mit der *MKP-Datenfeld-ID* gleichzusetzen ist. Das Beobachten eines Topic ist das bei MQTT vorherschende Idiom. Alle Clients die sich als Beobachter für ein Topic registriert haben (engl. subscribe) bekommen vom Broker Nachrichten zu dem Topic zugestellt.
Das Lesen eines *MKP-Datenfeldes* (ohne es zu beobachten) kann bei MQTT mit Hilfe des Retain-Flags umgesetzt werden. Dadurch speichert der Broker die letzte Nachricht eines Topics. Wenn sich ein Client beim Broker für ein Topic „subsricbed“, wird ihm die gespeicherte Nachricht vom Broker zugestellt.

3.5 Umsetzung mit BLE

Bei BLE entspricht der Server einem Gerät (im weiteren BLE-Server genannt), welches für jedes *MKP-Datenfeld* über eine BLE-Characteristic verfügt. Die *MKP-Datenfeld-ID* wird mit der UUID der Characteristic umgesetzt. Alle Charakteristiken sind einem BLE-Service zugeordnet, dessen UUID vom „BLE-Server“ durch Advertising publiziert wird. Dadurch können die Knoten den BLE-Server finden und sich mit ihm verbinden. Das Beobachten eines *MKP-Datenfeldes* wird mit Hilfe von BLE-Notifications realisiert.
Da für die gewünschte Funktionalität keine bereits vorgefertigte Software bzw. kein fertiges

Gerät zur Verfügung steht, muss der Codegenerator, zusätzlich zu den Arduino Projekten für die Knoten, auch ein Arduino Projekt für den BLE-Server erstellen. Für den Betrieb des BLE-Servers wird ein zusätzliches Gerät (hier ESP32) verwendet.

3.6 Gemischtes Netzwerk

Mit dem vorgestellten Konzept ist es möglich, dass *Knoten*, die mit unterschiedlichen Kommunikationsprotokollen arbeiten, miteinander kommunizieren. Hierzu wird (ein im weiteren *Brücke*) genanntes Gerät benötigt, das mit MQTT und BLE kommunizieren kann. Es werden zwei (oder mehr) *MKP-Server* verwendet, die über eine Brücke verbunden werden. Die Brücke beobachtet die *MKP-Datenfelder* auf einem Server und schreibt im Falle einer Änderung den Wert auf den anderen Server (und umgekehrt).

Durch die in der Konfigurationsdatei enthaltenen Informationen, weiß die Brücke auf welchem Server Datenfelder geschrieben werden können und welche Werte auf den anderen Server geschrieben werden müssen. So können z.B. *Attribute* nur auf dem Server geschrieben werden mit dem der zugehörige Knoten verbunden ist. Das „Call-Feld“ eines Methodenaufrufes kann allerdings auf jedem Server beschrieben werden.

3.6.1 Vernetzung von BLE-Servern

BLE arbeitet mit Funk und hat üblicherweiße eine Reichweite von weniger als 100 Metern. Damit lässt sich zwar ein Netzwerk aus Geräten aufbauen, es kann jedoch nicht als „Internet of Things“ bezeichnet werden.
Bei MQTT hingegen arbeitet der Broker mit TCP/IP und kann im Prinzip unbegrenzt viele Geräte (abhängig von der verwendeten Hardware) auf der ganzen Welt miteinander verbinden.

Es ist also sinnvoll, die Geräte von unterschiedlichen BLE-Servern miteinander zu vernetzen. Hier soll das im vorherigen Abschnitt vorgestellte Konzept der Brücke angewandt werden. Jeder BLE-Server wird über eine Brücke mit einem zentralen MQTT-Broker verbunden.
Das Verwenden von MQTT für die Verbindung der BLE-Server erleichtert die Implementierung, da die Brücke für ein gemischtes Netzwerk (aus BLE und MQTT) ohnehin benötigt wird. Da mit dem MQTT-Broker auch MQTT fähige Knoten verbunden werden können, ist somit eine Mischung mit MQTT fähigen Knoten noch immer problemlos möglich.

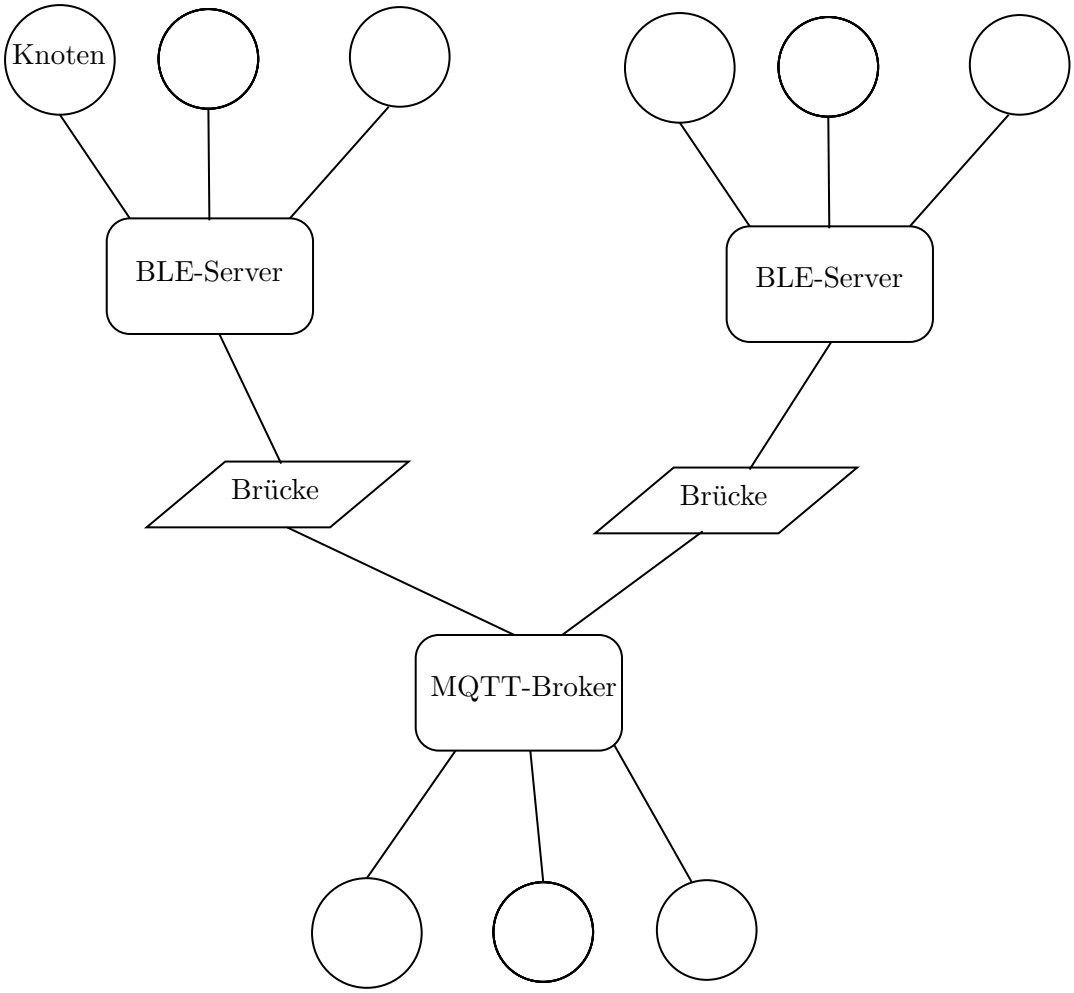

Abbildung 3.8: gemischtes Netzwerk mit mehreren BLE-Servern

3.7 Attribute

Attribute können nur von ihrem eigenen Knoten aus beschrieben werden. Es ist problemlos möglich eine „Setter-Methode“ zu verwenden, wenn der Wert eines Attributes von einem anderen Knoten geändert werden soll.

Die Umsetzung mit dem minimalen Kommunikationsprotokoll erfolgt über ein zum *Attribut* gehörendes *MKP-Datenfeld* auf dem *MKP-Server*. Nur der Besitzer des Attributes schreibt die Daten auf den *MKP-Server*, die anderen Knoten können den Wert lesen.

Wenn ein *Knoten* eventgesteuert auf die Änderung eines *Attributes* reagieren soll, registriert sich der Knoten beim *MKP-Server* als Beobachter. Bei Änderung des Wertes wird anschließend eine Funktion ausgeführt, die den neuen Wert als Parameter erhält.

3.8 Methoden

Beim Aufruf einer *Methode* blockiert der aufrufende *Knoten* (Caller) so lange bis der aufgerufene *Knoten* (Callee) antwortet. Dafür werden auf dem *MKP-Server* zwei *MKP-Datenfelder* benötigt: Eines durch das der Funktionsaufruf gestartet und Parameterwerte übermittelt werden (im Folgenden Parameterfeld genannt), und ein Anderes für den Rückgabewert (Returnfeld).

Der Callee beobachtet stets das Parameterfeld. Wenn die Funktion aufgerufen werden soll, registriert sich der Caller für das Beobachten des Returnfeldes und schreibt die Parameter ins Parameterfeld. Dies löst den Funktionsaufruf beim Callee aus, welcher den Rückgabewert der Funktion in das Returnfeld schreibt. Der Caller wartet bis er über die Änderung des Returnfeldes benachrichtigt wird, womit der Methoden-Aufruf abgeschlossen ist.

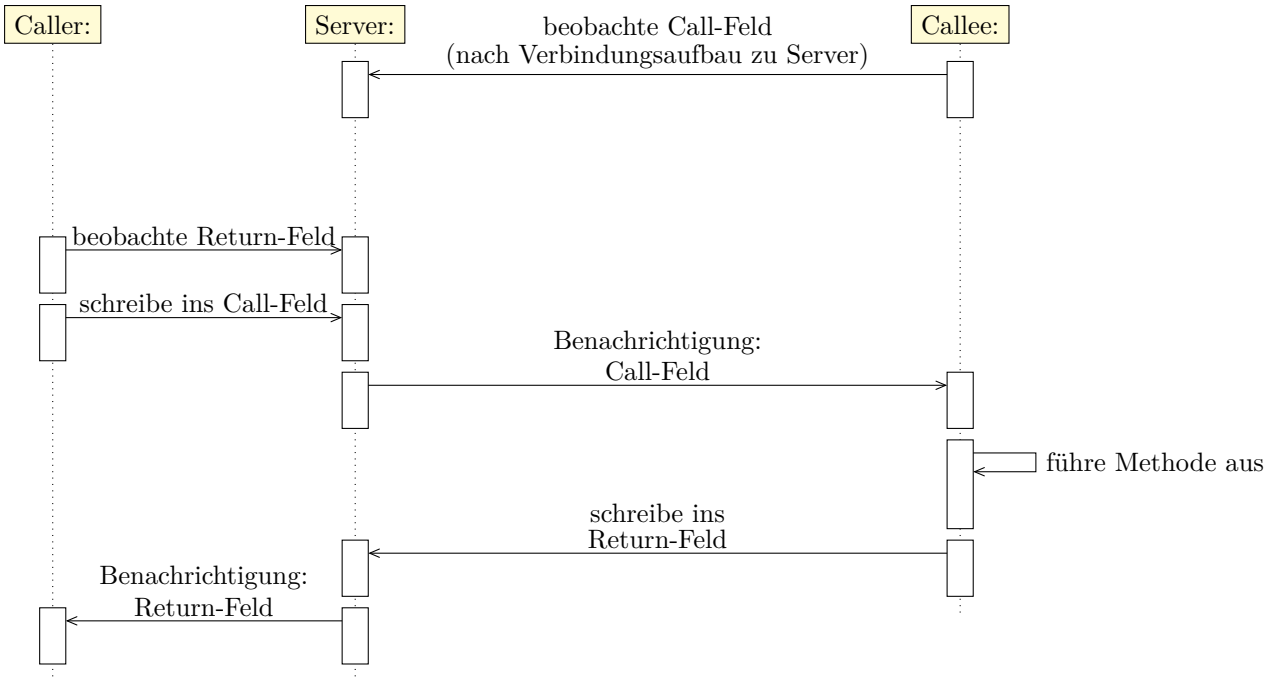

Abbildung 3.9: Methodenaufruf mit *Datenfeldern*

Dieser naive Ansatz der Implementierung von *Methoden* funktioniert jedoch nicht, wenn mehrere *Knoten* eine Funktion aufrufen können. Durch das gleichzeitige Schreiben in das Parameterfeld kann es zu Race Conditions kommen, und ein *Knoten* erhält als Rückgabewert möglicherweise den Wert des Funktionsaufrufes eines dritten Knotens.

Um dies zu verhindern, ließe sich für jeden potentiellen Caller ein separates Parameter- und Returnfeld verwenden. Das hätte jedoch zum Nachteil, dass die Menge der *MKP-Datenfelder* auf dem Server linear von der Menge der Verbindungen abhängt. Auch das nachträgliche Hinzufügen von Knoten wäre schwieriger, da nun der Callee eine Abhängigkeit zum Caller hat (er muss die speziellen Datenfelder kennen).

Um die gravierenden Nachteile dieses Ansatzes zu vermeiden, wurde ein anderer Ansatz gewählt, der nur mit einem Parameter- und Returnfeld auskommt:

Zum Eliminieren möglicher Race Conditions wird den zum Funktionsaufruf gehörenden Daten eine einzigartige Zahl (genannt Tag) hinzugefügt. Mit Hilfe des Tags kann der aufrufende Knoten überprüfen, ob eine Antwort im Returnfeld, zu dem von ihm gesendeten Funktionsaufruf gehört. Der Tag besteht dabei aus einer für jeden Knoten einzigartigen Zahl (Knoten-ID), sowie einer Zahl die vom Knoten inkrementiert wird (rolling Number). Dadurch wird sichergestellt, dass ein Knoten nicht fälschlicherweise die Daten eines anderen oder eines vorherigen Funktionsaufrufes erhält.
Insofern beim gleichzeitigen Schreiben von mehreren Knoten in ein *MKP-Datenfeld* eine Benachrichtigung (an Beobachter) für jeden Schreibvorgang gesendet wird, können bei Methodenaufrufen mit diesem Mechanismus keine Race Conditions vorkommen.

3.9 Fragmentierung

Im *minimalen Kommunikationsprotokoll* ist die Speicherkapazität eines *MKP-Datenfeldes* begrenzt. Wird dieses beispielsweise mit BLE realisiert, kann ein *MKP-Datenfeld* maximal 512 Bytes groß sein. Laut Datenmodell können Attribute und Funktionsparameter jedoch auch Werte von nicht-primitiven Datentypen wie „Array“ oder „String“ sein. Die Größenbeschränkung eines Arrays auf maximal 512 Bytes ist jedoch nicht sehr praktikabel.
Zur Speicherung größerer Datenmengen soll es ermöglicht werden, mehrere *MKP-Datenfelder* für einen Wert zu verwenden. Diese Aufteilung von Daten wird im Weiteren als *Fragmentierung* bezeichnet.

3.9.1 Fragmentierung von Attributen

Beim naiven Ansatz, die Daten eines Attributes in x Byte große Fragmente aufzuteilen, und in n viele *MKP-Datenfelder* auf dem *MKP-Server* zu schreiben, besteht die Gefahr, von Race Conditions. Wenn ein *Knoten* in dem Augenblick die Datenfelder eines zu ihm gehörenden *Attributes* beschreibt, während ein anderer *Knoten* die Daten liest, werden inkorrekte Daten empfangen (ein Teil der alten und neuen Daten).

Da beim vorgestellten Datenmodell die Daten eines *Attributes* nur von einem einzigen *Knoten* geschrieben werden können, kann das Eliminieren von Race Conditions stark vereinfacht werden. Es ist ausreichend, wenn der lesende *Knoten* erkennen kann, ob die Daten korrupt sind. Dies wird durch hinzufügen einer Prüfsumme bzw. eines Hashwerts erreicht. Nach dem Lesen der *MKP-Datenfelder*, überprüft der lesende *Knoten*, ob die errechnete Prüfsumme mit der Prüfsumme der gelesenen Daten übereinstimmt. Wenn dies nicht der Fall ist, sind die Daten korrupt. Das Lesen der Daten kann in diesem Fall kurze Zeit später wiederholt werden.

Wenn die *MKP-Datenfelder* in einer definierten Reihenfolge beschrieben werden, und der lesende Client das letzte Feld beobachtet, kann das Eintreten des Fehlerfalls weiter minimiert werden.

3.9.2 Fragmentierung von Methoden

Bei Methodenaufrufen gestaltet sich die Fragmentierung komplizierter. Da die *Methode* eines *Knotens A* zeitgleich von mehreren anderen *Knoten (B, C, ...)* aufgerufen werden kann, können beim Schreiben in das Parameter-Feld Race Conditions auftreten. Mit einer Prüfsumme könnte zwar, wie schon bei den *Attributen*, überprüft werden, ob die Daten valide sind. Sollten jedoch zwei Knoten durch gleichzeitiges Schreiben die Daten korrumpieren, wären die Daten von beiden Knoten verloren.
Um den für die *Attribute* verwendeten Ansatz auch für Methodenaufrufe zu verwenden, muss zuerst sichergestellt werden, dass zu jedem Zeitpunkt nur ein einziger *Knoten* die Berechtigung hat, in das Parameter-Feld zu schreiben. Dazu muss der Caller vor dem Schreiben in das *Parameter-Feld* die Berechtigung dazu beim Callee einholen. Diese Berechtigung wird nur an einen einziger Caller vergeben, die Anderen müssen warten.
Beim Return-Feld kann die Fragmentierung auf die gleiche Weise wie bei den Attributen erfolgen, da der einzige *Knoten* mit Schreibberechtigung der Callee ist.

3.9.3 Übertragung von Daten beliebiger Größe

Bei der *Fragmentierung* wird die maximale Größe eines Wertes statisch festgelegt. Der Ansatz ermöglicht es nicht, zur Laufzeit beliebig große Datenmengen zu übertragen. Um auch zur Laufzeit beliebig große Datenmengen auszutauschen, werden im Folgenden zwei Möglichkeiten erörtert.

Eine Möglichkeit zur Laufzeit beliebig große *Attribute* zu ermöglichen besteht darin, nur ein einziges *MKP-Datenfeld* für die Daten zu verwenden. Die fragmentierten Daten werden nacheinander in das gleiche Feld geschrieben. Alle Knoten die das Feld beobachten, können die Daten nach dem Empfang selbst abspeichern und zusammensetzten.
Durch einen Tag (z.B. das erste Byte in jedem *MKP-Datenfeld*) kann erkannt werden, ob es sich um das erste Paket, ein weiterführendes, oder das letzte Paket handelt.
Dies hat jedoch den Nachteil, dass ein Knoten nur die Werte von Attributen sehen kann, die geschrieben wurden, als er selbst verbunden war. Da es auch IoT-Geräte gibt, die sich (z.B. zur Energieeffizienz) nur gelegentlich mit dem Netzwerk verbinden, ist dieser Ansatz für den hier vorgesehenen Anwendungszweck nicht geeignet.

Die zweite Möglichkeit große Datenmengen zwischen zwei Knoten zu übertragen, kann mit den bereits beschriebenen Methodenaufrufen (ohne *Fragmentierung*) realisiert werden. Dazu benötigt ein *Knoten* der Daten empfangen soll eine *Methode*, über deren Parameter ihm der

aufrufende Knoten die Daten sendet. Diese könnte z.B. folgende Signatur haben:

```
1  bool inputStream(std::vector<uint8_t> data, bool isFirstPackage,
       ↪ bool isLastPackage)
```

Abbildung 3.10: inputStream Funktionssignatur

Die Parameter „isFirstPackage“ und „isLastPackage“ signalisieren, ob eine neue Übertragung gestartet werden soll oder die Übertragung abgeschlossen ist.
Über den Rückgabewert wird mitgeteilt, ob weitere Daten empfangen werden können oder die Übertragung aufgrund eines Fehlers abgebrochen werden musste (z.B. interner Speicher voll). Auf diese Weise kann der Nutzer des Codegenerators mit wenig Aufwand das Übertragen von großen Datenmengen bei Bedarf selbst implementieren. In Abschnitt 5.3 wird ein Besipielprojekt vorgestellt, bei dem dieser Ansatz getestet wird.

4 Implementierung

Um das vorgestellte Konzept zu realisieren, wurden mehrere Programme implementiert. Der Codegenerator erzeugt die Programmgrundgerüste für die *Knoten* und, wenn erforderlich, fertige Programme für BLE-Server 3.5.
Die, in Abschnitt 3.6 beschriebene, BLE-Brücke ermöglicht eine Kommunikation zwischen BLE- und MQTT-Geräten und stellt eine eigenständige Software dar.
Beim *Webinterface* (4.3) handelt es sich um eine weitere, vom Codegenerator unabhängige Software, die es möglich macht, über eine Webbrowser auf die IoT-Anwendung zuzugreifen. Dazu speichert das *Webinterface* alle Werte von *Attributen* in einer Datenbank (mit Zeitstempel) und ermöglicht eine Visualisierung der enstandenen Zeitreihen. Durch den Aufruf von *Methoden* kann auch eine manuelle Steuerung über den Browser erfolgen. Die Verwendung des *Webinterface* ist optional.

4.1 Der Codegenerator

Für die Implementierung des Codegenerators wurde die Programmiersprache Python gewählt. Python besitzt eine umfangreiche Standardbibliothek, die unter anderem einen betriebssystemunabhängigen Zugriff auf das Dateisystem ermöglicht. Das für das Lesen der Konfigurationsdatei notwendige Parsen von JSON-Dateien ist ebenfalls in der Standartbiblothek enthalten.
Python besitzt zudem eine umfangreiche Auswahl von Open-Source Biblotheken, welche sehr einfach über „pip“ installiert werden können.
Aus diesem Grund ist Python sehr effizient, was die Zeit betrifft, die zur Programmierung benötigt wird. Für den Anwender bietet Python den Vorteil, dass dieser den Codegenerator auf diversen Betriebssystemen, ohne aufwendige Installation, nutzen kann.
Im Bezug auf die Ausführungsdauer des Codes ist Python im Vergleich zu anderen populären Programmiersprachen wie C++, C# oder Java zwar sehr langsam, da dies jedoch nur die Ausführungszeit der Codegenerierung betrifft, ist das von untergeordneter Bedeutung.

Für die Implementierung des Codegenerators wird die, in Abschnitt 2.7 beschriebene, Template-Engine „Jinja2“ verwendet. Bei dem hier vorgestellten Codegenerator eignet sich die Implementierung mit einer Template-Engine, da der zu erzeugende Quellcode eine sehr begrenzte Komplexität hat. So lassen sich alle benötigten Features durch das Einfügen von Textblöcken, mit den Kontrollstrukturen „for“ und „if“/„else“, realisieren.

Ein anderer Ansatz, bei dem die Programmlogik zuerst in Form einer Baumstruktur (genannt Abstract Syntax Tree) vorliegt, aus welcher der Quellcode erzeugt wird, ist im Vergleich wesentlich mächtiger. Hier kann die Baumstruktur programmatisch modifiziert, und somit beliebig komplexer Quellcode erzeugt werden. Für die Implementierung des (hier vorgestellten) Codegenerators wird dies jedoch nicht benötigt. Da der Ansatz den Nachteil einer wesentlich höheren Komplexität besitzt, wurde die Template-Engine bevorzugt.

Ein weiterer Vorteil bei der Verwendung einer Template-Engine liegt im „Workflow“ der Implementierung.
Es wurde so vorgegangen, dass zuerst ein C++ Arduino Projekt (wie es später erzeugt werden soll) manuell programmiert wurde. Dieses wurde getestet, und diente anschließend als Vorlage zum Erstellen einer C++ Template-Datei. Dazu wurden alle Teile, die später von der Konfiguration abhängen, durch Template-Anweisungen ersetzt.

Damit der Codegenerator leicht zu verstehen und modifizieren ist, wird möglichst wenig Code mit der Template-Engine erzeugt, und stattdessen ein großer Teil des Codes in regulärem C++ implementiert. Zu diesem Zweck wird, unter anderem, „C++ Template Programmierung“ eingesetzt.
So wird beispielsweise das Serialisieren und Deserialisieren von Daten mittels *C++ Template Funktionen* realisiert[4]. Die Funktionen erhalten als C++ Template Parameter eine Liste von Typen, die (beim konkreten Funktionsaufruf) mit Jinja2-Templates eingefügt wird. Wie das Serialisieren von Werten (eines bestimmten Typs) abläuft, kann jedoch trotzdem anhand von regulärem C++ Code nachvollzogen werden.

Der Codegenerator besteht aus der Datei „codegen.py“, regulären C++ Dateien (die bei der Codegenerierung lediglich kopiert werden) und C++ Dateien die Jinja2 Templates enthalten. Die C++ Dateien sind dabei in unterschiedlichen Verzeichnissen untergebracht.

- „TemplateNode“ enthält C++ Dateien, welche sowohl für BLE als auch MQTT Knoten verwendet werden
- „TemplateNodeBLE“ enhält C++ Dateien, welche BLE spezifisch sind
- „TemplateNodeBLE“ enhält C++ Dateien, welche MQTT spezifisch sind
- „TemplateServerBLE“ enhält C++ Dateien, die für das Erzeugen eines BLE-Servers verwendet werden

In den folgenden Abschnitten wird zuerst der Aufbau eines Programmgrundgerüstes beschrieben. Anschließend werden C++ Klassen, die für die Implementierung von Attributen und Methoden verwendet wurden, vorgestellt und MQTT bzw. BLE spezifische Details gezeigt.

[4]Erklärung erfolgt in Abschnitt 4.1.2

4.1.1 Programmgrundgerüst eines Knotens

Wie bereits erwähnt wird als Teil des Programmgrundgerüstes vom Codegenearator die Datei *CustomCode.cpp* erzeugt, die leere Funktionsrümpfe enthält. Außerdem muss Code erzeugt werden, welcher im Hintergrund eine Verbindung mit dem Netzwerk aufbaut und die Funktionen in CustomCode.cpp zum richtigen Zeitpunkt aufruft. Dazu werden die Datei „<Knotenname>.ino“, sowie weitere C++ Dateien (.cpp und .h) im „/src“ Verzeichnis erzeugt.

Eine .ino Datei ist bei jedem Arduino Projekt zwingend erforderlich. Sie enthält die Funktionen „setup()“ und „loop()“, welche (analog zu „main()“ bei C++) dem Einsprungpunkt des Programmes entsprechen.
Jeder weitere vom Codegenerator erzeugte interne Code (der vom Anwender nicht modifiziert werden sollte) wird im Projektverzeichnis, im Ordner „/src“, abgelegt. Der Inhalt dieser Dateien ist folgendermaßen strukturiert:

- „Serialize.h“ enthält alle Funktionen für das Serialisieren/Deserialisieren von Daten
- „RemoteValue.h“ enthält Klassen, mit denen Attribute und Methoden implementiert sind
- „RemoteValues.h/.cpp“ verwendet die in RemoteValue.h definierten Klassen, um für die vom Knoten verwendeten Attribute und Methoden globaler Objekte zu definieren
- „RemoteFunktionAbstract.h“ enthält abstrakte Klasse die den Ablauf eines Methodenaufrufes implementiert
- „ComposedAttributeAbstract.h“ enthält abstrakte Klasse die die Fragmentierung von Attributen implementiert
- „Internal.h“ enthält BLE/MQTT spezifische Helper-Funktionen. So werden z.B. Callbacks für die Verwendung der BLE-Biblothek definiert. Bei einer Verbindung mit dem MQTT-Broker werden die vom Knoten benötigten Topics abooniert (engl. subscribe), etc.

Das generierte Programmgrundgerüst ist bereits selbstständig lauffähig. Abbildung 4.1 zeigt den Kontrollfluss beim Start eines *Knotens*. Zuerst wird intern Code ausgeführt, um das Gerät zu initialisieren (z.B. Anschalten der Wifi Antenne) und danach die vom Anwender bereitgestellte *Setup()* Funktion aufgerufen.
Es wird ein zweiter Thread erzeugt, in dem die vom Anwender bereitgestellte *Loop()* Funktion, in einer Endlosschleife, ausgeführt wird. Parallel dazu wird (im ersten Thread) stets überprüft, ob eine Verbindung zum *Server* (hier BLE-Server oder MQTT-Broker) besteht. Wenn keine Verbindung besteht, wird versucht diese wiederherzustellen.

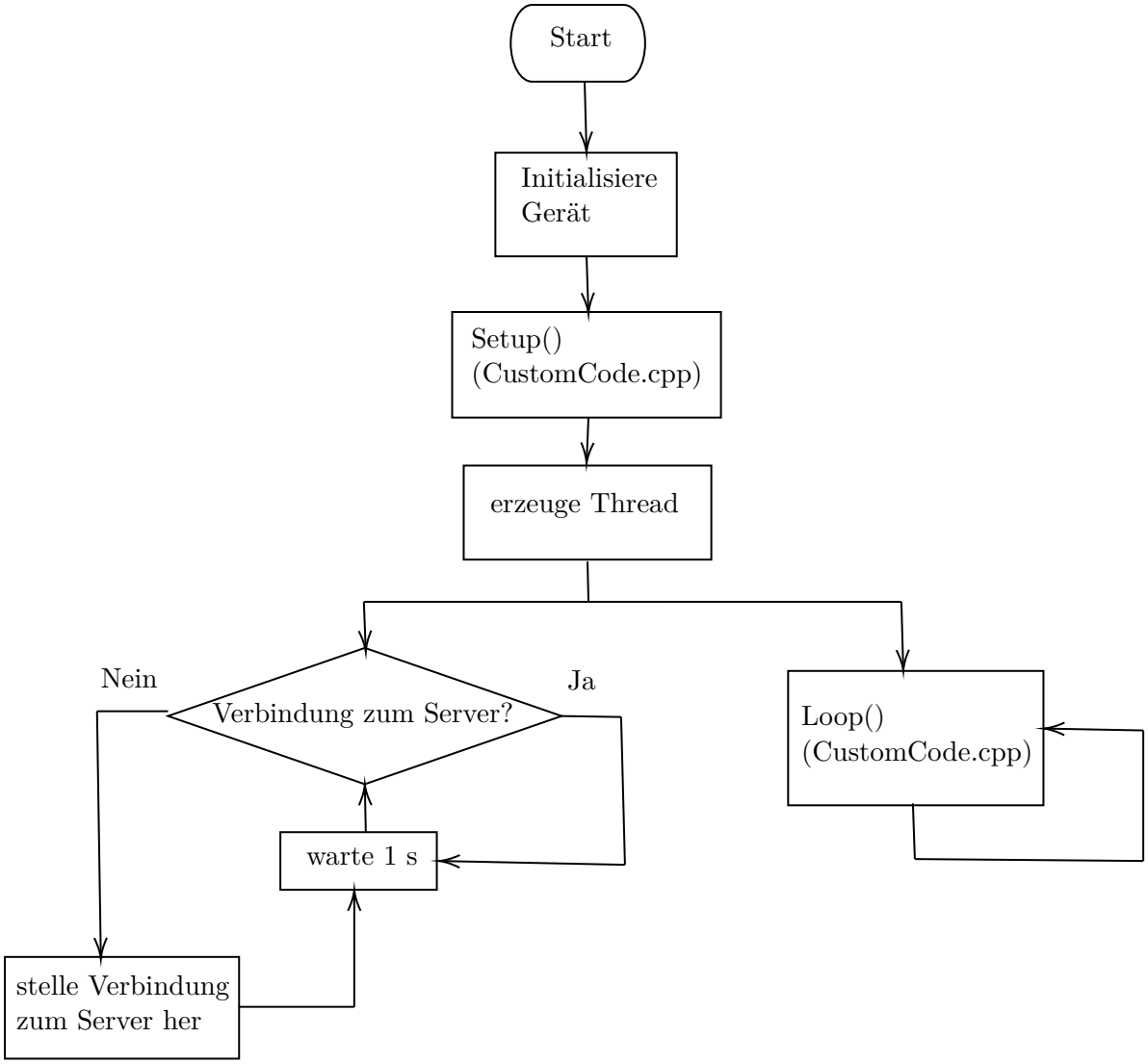

Abbildung 4.1: Kontrollfluss Programmgrundgerüst

4.1.2 Serialisierung und Deserialisierung von Daten

Um die Werte von Attributen und Methoden-Parametern über das Netzwerk übertragen zu können, müssen diese in ein Array aus Bytes (in C++ z.B. std::vector<uint8_t>) umgewandelt werden. Dies wird auch als Serialisierung (bzw. Deserialisierung in die andere Richtung) bezeichnet.

Bei primitiven Datentypen (wie z.B. „int“ oder „float“) entspricht die Bitrepräsentation bereits den serialisierten Daten, welche lediglich anders interpretiert werden müssen. So kann z.B. ein „float“ auch als Array aus 4 Bytes interpretiert werden.
Bei Datentypen die Pointer oder Referenzen enthalten (hier „std::vector“ und „std::string“) ist dies jedoch nicht möglich. Wenn hier lediglich die Bitrepräsentation übertragen wird, enthält diese den Zahlenwert eines Pointers, jedoch nicht die dahinter liegenden Daten.

Um also die Daten, welche zur gespeicherten Adresse gehören, und nicht die Adresse selbst zu serialisieren, wurde ein möglichst einfacher Ansatz gewählt. Zuerst wird, in einem 4 Byte großen Integer, die Anzahl der Elemente abgespeichert. Danach wird auf die Daten hinter der Adresse zugegriffen und sie werden (serialisiert) an das Ergebnis angehängt.

Da alle Kommunikationsteilnehmer den Datentyp eines Attributes (bzw. die Datentypen von Methodenparametern) kennen, ist es bei dieser Anwendung nicht notwendig, weitere Meta-Informationen (wie z.B. den Datentyp eines Wertes) in den serialisierten Daten zu speichern. Alle verwendeten Geräte arbeiten mit der „Little-Endian“ Sortierung. Daher werden die serialisierten Daten auch im „Little-Endian“ Format abgelegt.

Beispiel

Wenn eine *Methode* mit der Signatur „void doSomething(int a, std::vector<short> b, std::string c)“ aufgerufen wird, müssen die Parameter serialisiert und in ein *MKP-Datenfeld* des *MKP-Servers* (Call-Feld) geschrieben werden. Die serialisierten Daten würden dann wie in Abbildung 4.2 aussehen. Da der Empfänger auch die Datentypen der Parameter kennt, liest er zuerst vier Bytes (Größe von int), um den Parameter „a“ zu deserialisieren. Aus den nächsten 4 Bytes wird die Anzahl der Elemente des Vektors ermittelt. Im Beispiel besitzt der Vektor eine Länge von drei Elementen, daher werden drei mal zwei Bytes (Größe von short) an Daten eingelesen, um die Elemente zu deserialisieren. Die nächsten 4 Bytes geben dann die Länge des „std::string“ von Parameter „c“ an, womit auch dieser deserialisiert werden kann.

int a = 42				std::vector<short> = {7, 11, 13}									
				Anzahl der Elemente				Daten					
42	0	0	0	3	0	0	0	7	0	11	0	13	0

...

...

std::string = „hallo“								
Anzahl der Elemente				Daten				
5	0	0	0	h	a	l	l	o

Abbildung 4.2: Beispiel Serialisierung von Daten

Implementierung in C++

In C++ wurde das Serialisieren und Deserialisieren mit Hilfe von Template-Funktionen implementiert. Sie erhalten als Template-Parameter eine Liste von Typen und können mit dieser Information Daten (de)serialisieren.

Es wird ein im C++-11 Standard eingeführtes Feature verwendet, welches als „parameter pack“ bezeichnet wird. Das Feature erlaubt es eine beliebige Anzahl (null oder mehr) Argumente an einen Template-Parameter zu übergeben.

Der Code in Abbildung 4.3 zeigt, wie das Serialisieren mit der „toBytes“ Methode implementiert wurde. Durch Überladung der „toBytes“-Funktion wird abhängig vom Typ des Parameters die korrekte Form der Serialisierung gewählt.
Für alle einfachen Typen deren Bitrepräsentation mit den serialisierten Daten identisch ist, wird die in Zeile 1 gezeigte Funktion verwendet.
Für std::vector<T> und std::string wurden zwei Überladungen der „toBytes“-Funktion implementiert. Die in Zeile 1 gezeigte Funktion kann zwar mit Parametern von beliebigen Typen aufgerufen werden, wenn jedoch eine spezielle Überladung existiert, wird diese stattdessen verwendet. Daher wird beim Aufruf der „toBytes“ Funktion mit einem std::vector (bzw. std::string) die Funktion in Zeile 10 (bzw. Zeile 27) ausgeführt.
Für das Serialisieren von mehreren Werten, kann die Funktion in Zeile 50 verwendet werden. Diese kann eine beliebige Zahl an Parametern erhalten. Für jeden Parameter wird die toBytes() Funktion mit einem Parameter aufgerufen, wodurch, abhängig vom Typ des Parameters, automatisch die korrekte Überladung gewählt wird. Anschließend werden, mit der „flatten()“ Funktion, die Ergebnisse der einzelnen Funktionsaufrufe zu einem std::vector<uint8_t> zusammengefügt.

Der Ansatz wurde gewählt, um die vom Codegenerator verwendeten Jinja2-Templates zu vereinfachen. Der Codegenerator muss lediglich den Quellcode für den Aufruf der „toBytes“ Methode mit den korrekten Parametern generieren. Die komplexe Logik der Serialisierung ist in regulärem C++ implementiert.
Außerdem ist es mit dem vorgestellten Ansatz sehr leicht möglich, weitere Datentypen zu unterstützen. Es muss hierfür lediglich eine weitere Überladung der „toBytes()“ Funktion implementiert werden. Anpassungen am bestehenden Code sind jedoch nicht notwendig.

```cpp
template <typename T>
std::vector<uint8_t> toBytes(T data)
{
    std::vector<uint8_t> result;
    uint8_t* tmp = (uint8_t*)&data;
    result.insert(result.begin(), tmp, tmp + sizeof(T));
    return result;
}

template <typename T>
std::vector<uint8_t> toBytes(std::vector<T> data)
{
    std::vector<uint8_t> result;
    uint32_t len = data.size();
    uint8_t* tmp = (uint8_t*)&len;
    result.insert(result.begin(), tmp, tmp + sizeof(uint32_t));

    tmp = (uint8_t*)data.data();
    for (auto& e: data)
    {
        std::vector<uint8_t> elementData = toBytes(e);
        result.insert(result.end(), elementData.begin(),
            ↪ elementData.end());
    }
    return result;
}

inline std::vector<uint8_t> toBytes(std::string data)
{
    std::vector<uint8_t> str(data.begin(), data.end());
    uint32_t len = str.size();
    uint8_t* tmp = (uint8_t*)&len;

    std::vector<uint8_t> result;
    result.insert(result.begin(), tmp, tmp + sizeof(uint32_t));

    result.insert(result.end(), str.begin(), str.end());
    return result;
}

inline std::vector<uint8_t> flatten(std::vector<std::vector<uint8_t>>
    ↪ data)
{
    std::vector<uint8_t> result;
    for (auto& v: data)
    {
        result.insert(result.end(), v.begin(), v.end());
    }
    return result;
}

template <typename... Args>
std::vector<uint8_t> toBytes(Args... args)
{
    return flatten(std::vector<std::vector<uint8_t>>{
        ↪ toBytes(args)... });
}
```


Abbildung 4.3: Serialisierung in C++

Der Ansatz findet sich auch in der Deserialisierung wieder, in Form der „deserialize()“ Funktion (sowie ihrer Überladungen). Der Funktion werden als Template-Parameter die Typen der Werte übergeben, die in den serialisierten Daten enthalten sind. Bei erfolgreicher Deserialisierung wird ein std::tuple zurückgegeben, das die deserialisierten Werte enthält. Hier muss jedoch zusätzlich eine Fehlerbehandlung stattfinden, da es möglich ist, dass ein fehlerhafter Knoten Daten versendet, die nicht deserialisiert werden können. Um auch hier die Auswahl der korrekten Funktion durch Überladung zu ermöglichen, wird als zusätzlicher Parameter ein Null-Pointer übergeben. Er wird nicht verwendet, und dient nur dazu, eine Überladung zu ermöglichen (in C++ können Funktionen nur anhand von Parametern, jedoch nicht anhand des Rückgabewertes überladen werden). Alle Funktionen zur Serialisierung/Deserialisierung sind in der Datei TemplateNode/-Serialize.h zu finden.

```
1  template <typename T>
2  T deserialize(uint8_t*& data, uint8_t* end, T*)
3  {
4      ...
5  }
6
7  template <typename T>
8  std::vector<T> deserialize(uint8_t*& data, uint8_t* end,
       ↪ std::vector<T>*)
9  {
10     ...
11 }
12
13 inline std::string deserialize(uint8_t*& data, uint8_t* end,
       ↪ std::string*)
14 {
15     ...
16 }
17
18 template <typename... Args>
19 nonstd::optional<std::tuple<Args...>> deserialize(uint8_t* data,
       ↪ uint8_t* end)
20 {
21     ...
22 }
```

Abbildung 4.4: Funktionen zur Deserialisierung

4.1.3 Attribute

Für die Implementierung von *Attributen*, werden die Klassen „RemoteValue“ sowie „RemoteValueReadOnly“ definiert. „RemoteValueReadOnly“ wird für den Zugriff auf *Attribute* anderer *Knoten* verwendet, auf die nur lesend zugegriffen werden kann. Die Implementierung der beiden Klassen funktioniert bei BLE und MQTT unterschiedlich, die Schnittstelle ist jedoch gleich,

und wird in Abbildung 4.5 dargestellt.
Obwohl auch „set()“ fehlschlagen kann (keine Verbindung zum Server) wird auf eine Fehlerbehandlung verzichtet, da diese ohne Anpassung der verwendeten BLE-Bibliothek nicht möglich ist. Die zum Schreiben verwendete Funktion („void BLERemoteCharacteristic::writeValue“ lässt schlicht keine Fehlerbehandlung zu.
Ein Anpassen wäre zwar leicht möglich, da die Bibliothek intern bereits den Fehler erkennt und eine Log-Nachricht erstellt.[5] Dies würde jedoch die Installation und Verwendung des Codegenerators erheblich erschweren, da der Anwender manuell die Arduino Bibliothek löschen und die modifizierte Version installieren müsste.
Als Workaround kann der auf dem Server gespeicherte Wert, nach dem Schreiben, über „get()“ abgefragt werden. Wenn Programmierfehler auf Knoten und Server ausgeschlossen werden, kann ein Fehlschlagen der „set()“ Methode ohnehin nur bei nicht bestehender Netzwerkverbindung auftreten.

```
1
2  template<typename T>
3  class RemoteValueReadOnly
4  {
5  public:
6      bool get(T& value)
7      {
8          ...
9      }
10 };
11
12 template<typename T>
13 class RemoteValue : public RemoteValueReadOnly<T>
14 {
15 public:
16     void set(T value)
17     {
18         ...
19     }
20 };
```

Abbildung 4.5: Klassen für Zugriff auf Attribute

4.1.4 Methoden

Für die Implementierung von Methodenaufrufen wurde die abstrakte Klasse „RemoteFunktionAbstract“ erstellt, die unabhängig von BLE und MQTT die nötigen Abläufe implementiert. Die Implementierung der Klasse ist (gekürzt) in Abbildung 4.6 dargestellt.
Dies Klasse verfügt über einen überladenen „()-Operator“, der es ermöglicht, ein Objekt der

[5] Zu finden in der Datei BLERemoteCharacteristic.cpp, Funktion BLERemoteCharacteristic::writeValue.

Klasse mit der gleichen Syntax aufzurufen, wie eine reguläre Funktion. Dieser wird vom Nutzer verwendet, um eine Methode (auf einem anderen Knoten) aufzurufen. Dabei werden zuerst die Funktionsparameter (wenn vorhanden) serialisiert und es wird (wie in Abschnitt 3.8 beschrieben) ein *Call-Tag* sowie eine *Rolling Number* zu den serialisierten Funktionsparametern hinzugefügt.
Anschließend werden die Daten mit der „sendData“ Methode in das zur Methode gehörende Call-Feld des *MKP-Servers* geschrieben. Bei der konkreten Verwendung mit BLE/MQTT wird, in einer abgeleiteten Klasse, die virtuelle Funktion „void sendData(std::vector<uint8_t>& data)“ überschrieben.
Nachdem die Daten geschrieben wurden, wird sichergestellt, dass die verwendete Semaphore reserviert ist (Zeile 26) und mit einem Timeout auf die Freigabe der Semaphore gewartet. Wenn die Semaphore innerhalb des Timeouts freigegeben wird, bedeutet dies, dass ein Rückgabewert empfangen wurde. Ansonsten ist der Fehlerfall eingetreten und es wird ein leeres „Optional“ zurückgegeben.
Das Freigeben der Semaphore wird von der Funktion „pickUpResult“ übernommen. Diese muss, währenddem die Funktion „operator()“ auf Freigabe der Semaphore wartet, von einem anderen Thread aufgerufen werden. Als Parameter bekommt sie Daten zugestellt, die vom Callee in das Return-Feld des Servers geschrieben wurden. Zuerst wird überprüft ob das *Funktionstag* (Abschnitt 3.8) mit dem zuvor gesendeten Aufruf übereinstimmt.
Danach werden die Daten deserialisiert und das Ergebnis im „result“ Attribut der Klasse „RemoteFunctionAbstract“ abgespeichert. Die anschließende Freigabe der Semaphore signalisiert, dass das Ergebnis empfangen wurde.

```
1
2  template <typename R, typename... Args>
3  class RemoteFunctionAbstract
4  {
5      SemaphoreHandle_t semaphore = xSemaphoreCreateBinary();
6      nonstd::optional<R> result = nonstd::nullopt;
7  public:
8      FunctionCallTag callTag;
9      virtual void sendData(std::vector<uint8_t>& data) = 0;
10
11     void pickUpResult(std::vector<uint8_t>& data)
12     {
13         ... // deserialisieren der Daten und ueberpruefen von
               ↪ Call-Tag und Rolling Number
14                 if (deserialized.has_value())
15                 {
16                     result = std::get<0>(deserialized.value());
17                     xSemaphoreGive(semaphore);
18                 }
19             }
20         }
21     }
22
23     nonstd::optional<R> operator()(Args... args)
24     {
25         ... // serialisieren der Parameter und Anfuegen von Call-Tag
               ↪ und Rolling Number
26         sendData(data);
27         xSemaphoreTake(semaphore, (TickType_t) 10);
28
29         if (xSemaphoreTake(semaphore, (TickType_t) portTICK_PERIOD_MS
               ↪ * 5000) != pdTRUE)
30         {
31             return nonstd::nullopt;
32         }
33         return result;
34     }
35
36     RemoteFunctionAbstract() = default;
37 };
```

Abbildung 4.6: Klassen für Aufruf von Methoden

Um die blockierenden Methodenaufrufe zu implementieren, werden zwei nebenläufige Threads (hier FreeRTOS Tasks) verwendet.
Ein Thread führt den Anwendercode aus, welcher, duch Aufuf der *operator()-Funktion*, den Methodenaufruf startet. Nachdem in das „Call-Feld“ geschrieben wurde, wird blockierend auf den Rückgabewert gewartet. Dieser Thread wird im weiteren als „Thread 1“ bezeichnet.
Der andere Thread (Thread 2) empfängt den Rückgabewert des Callee und teilt dies dem

blockierenden „Thread 1“ mit. Um einen Thread beim Warten auf das Ergebnis zu blockieren, wird eine Semaphore verwendet. Thread 1 reserviert die Semaphore und versucht anschließend diese erneut zu reservieren, was dazu führt, dass der Thread blockiert. Thread 2 kann nun die Semaphore freigeben, woraufhin die Blockierung von Thread 1 aufgehoben wird. Der Ablauf eines Methodenaufrufes ist im Sequenzdiagramm 4.7 dargestellt.

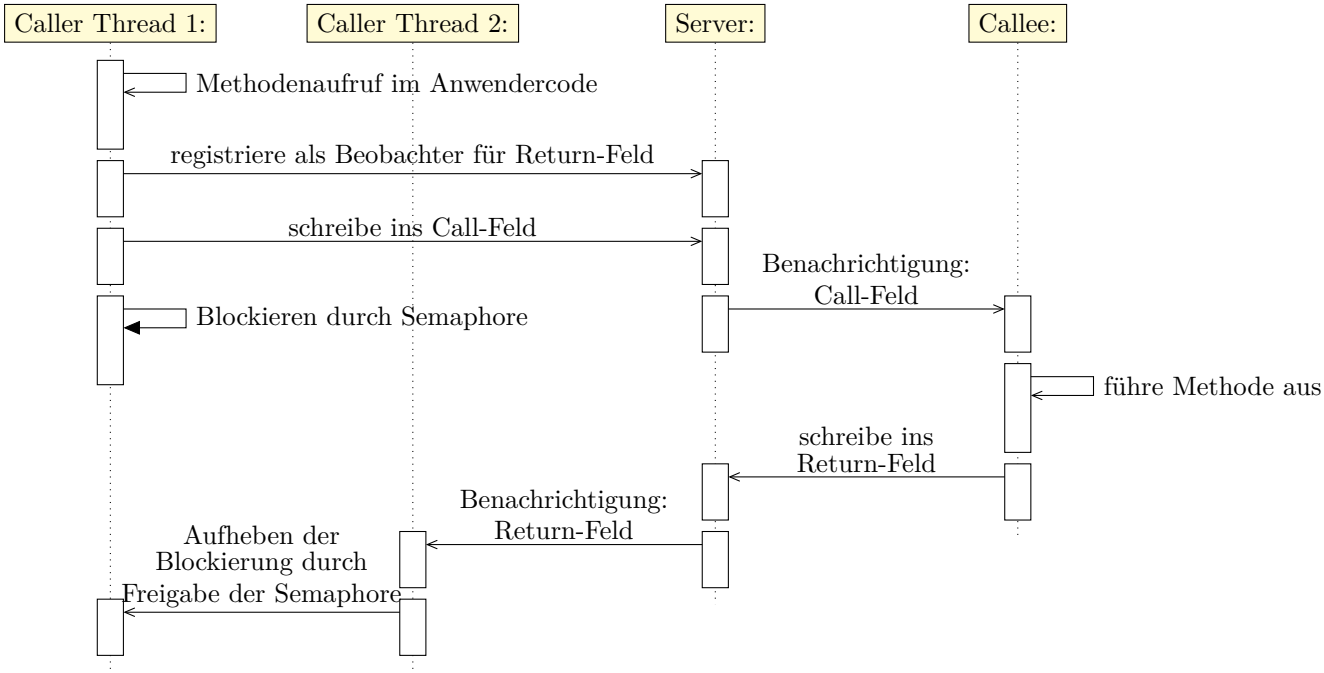

Abbildung 4.7: Sequenzdiagramm Methodenaufruf

Fragmentierung von Attributen

Die in Abschnitt 3.9.1 beschriebene *Fragmentierung von Attributen* wurde mit der Klasse „ComposedAttributeAbstrace“ in der Datei „ComposedAttributeAbstract.h“ implmentiert. Die Klasse übernimmt das Aufteilen und Zusammensetzen der Daten. Die Prüfsumme wird mit der SHA256 Hashfunktion berechnet.
Bei der Verwendung mit BLE/MQTT muss in einer abgeleiteten Klasse die „sendData()“ Methode überschrieben werden. Dies wurde jedoch nur für BLE implementiert, was ausreicht, um den im Konzept beschriebenen Ansatz anhand des Beispielprojektes 5.2 zu demonstrieren.

Bereitstellung für Anwender

Damit der Anwender auf *Attribute* und *Methoden* zugreifen kann, werden mittels Jinja2-Templates für jedes *Attribut* eine Instanz der Klasse „RemoteValue“ und für *Methoden* eine Instanz der Klasse „RemoteFunction“ als globale Variable erzeugt.
Diese globale Variable besitzt den Namen des Attributes (bzw. der Methode) und befindet sich

in einem „Namespace“, der den Namen des zugehörigen Knotens besitzt.
Dadurch kann der Nutzer auf ein Attribut (bzw. Methode) im C++ Code über <Knotenname>::<Attributname> (bzw. <Knotenname>::<Methodenname>) zugreifen.

Die Template Dateien, in denen diese globalen Variablen deklariert/definiert werden, finden sich unter TemplateNode/RemoteValues.h bzw. TemplateNode/RemoteValues.cpp. Durch includen der RemoteValues.h Datei kann somit im Anwendercode auf *Attribute* und *Methoden* zugegriffen werden. Abbildung 4.8 zeigt, wie die Deklarationen mittels Jinja2-Templates generiert werden.

```
1   {% for node in otherNodes %}
2 namespace {{node.name}}
3 {
4     {% for v in node.variables %}
5         {% if v.composed is defined %}
6 extern ComposedAttribute<{{v.type}}> {{v.name}};
7         {% else %}
8 extern RemoteValueReadOnly<{{v.type}}> {{v.name}};
9         {% endif %}
10        {% if v.isObserved is defined %}
11 void onChange_{{v.name}}({{v.type}});
12        {% endif %}
13    {% endfor %}
14
15    {% for fun in node.get('functions', []) %}
16        {% if fun.returnType is defined %}
17 extern RemoteFunction<{{ ([fun.returnType] + fun.get('params', []
    ↪ )|map(attribute='type')|list) |join(',') }}> {{fun.name}};
18        {% else %}
19 extern RemoteFunctionVoid<{{ fun.get('params', []
    ↪ )|map(attribute='type')|join(',') }}> {{fun.name}};
20        {% endif %}
21    {% endfor %}
22 }
```

Abbildung 4.8: Deklaration von Objekten zum Zugriff auf Attribute und Methoden

4.1.5 MQTT

Der *MKP-Server* entspricht unter MQTT dem MQTT-Broker, weshalb die Installation eines beliebigen Brokers bereits zur Bereitstellung eines lauffähigen Servers genügt. Als MQTT-Broker wird in den Beispielprojekten die Software „Eclipse Mosquitto“ eingesetzt. Für die Implementierung der MQTT Version eines Knotens wurde die „PubSubClient.h“ Bibliothek [4] verwendet. Diese stellt die Klasse „PubSubClient“ zur Verfügung, über deren Methoden „bool PubSubClient::subscribe(const char* topic)“ und „bool PubSubClient::publish(const char* topic, const uint8_t * payload, unsigned int plength, boolean retained)“ Nachrichten abboniert (engl. subscribed) und veröffentlicht (engl. published) werden können.

Mit der Methode „PubSubClient::setCallback()“ kann eine Callback Funktion hinterlegt werden, welche immer dann aufgerufen wird, wenn eine neue Nachricht vom Broker erhalten wurde. Dafür wird die, in der Datei *Internal.cpp* definierte, Funktion „void updateValues(char* topic, byte* message, unsigned int length)“ verwendet.
Über die Methode „PubSubClient::setServer(const char * domain, uint16_t port)“ wird eingestellt, über welche IP-Adresse/URL und über welchen Port eine Verbindung mit dem MQTT-Broker hergestellt werden kann. Dazu muss eine Netzwerkverbindung bestehen, wofür die *Arduino Wifi*-Bibliothek verwendet wird [5].

Attribute

Ein „MKP-Datenfeld“ entspricht unter MQTT einer MQTT-Nachricht zu einem bestimmten Topic. Ein Topic wird eindeutig durch einen String identifiziert, der aus dem Knotennamen und dem Namen des Attributes gebildet wird, welche durch einen Schrägstrich getrennt werden (<Knotenname>/<Attributname>).
Damit die letzte Nachricht zu einem bestimmten Topic vom Broker gespeichert wird, muss das „Retain-Flag“ auf *true* gesetzt werden. Wenn ein Knoten (hier MQTT-Client) ein Topic abboniert, wird diesem im Anschluss die gespeicherte Nachricht zugestellt.

Es gibt nun zwei Wege das Lesen eines Attributes umzusetzen:
Der Erste besteht darin, das zum Attribut gehörende Topic nach Verbindung mit dem Broker dauerhaft zu abbonieren. Jede Änderung führt zu einer Nachricht des Brokers mit dem neuen Wert, welcher daraufhin abgespeichert wird. Ein Aufruf der „get()“ Methode liefert dann lediglich den letzten gespeicherten Wert zurück.
Alternativ dazu kann das Topic nur zum Lesen abonniert und danach deabonniert werden. Es werden zum Lesen eines Attributes jedoch drei Nachrichten nötig (abonnieren, deabonnieren und der Wert).
Welcher Ansatz effizienter ist, hängt davon ab, wie oft das Attribut gelesen/geschrieben wird. Hier wurde der Einfachheit halber der erste Weg gewählt, da der hier implementierte Codegenerator als „Proof of Concept“ dient.

Die Klasse „RemoteValueReadOnly“ wurde bei MQTT wie in Abbildung 4.9 dargestellt implementiert. Das Empfangen eines neuen Wertes wird von der Funktion „void updateValues(char* topic, byte* message, unsigned int length)“ (in der Datei *Internal.cpp*) übernommen.
Diese wird als Callback bei der *PubSubClient*-Bibliothek hinterlegt. Immer wenn der Broker dem Client eine Nachricht zustellt, wird diese Callbackfunktion ausgeführt. Anhand des Topics muss entschieden werden, was mit der Nachricht passiert. Aus diesem Grund wird die Funktion *updateValues()* mit Jinja2-Templates generiert. Für jedes Attribut, auf welches der Knoten zugreifen kann, wird ein Codeblock erzeugt. Dieser überprüft zuerst ob das Topic zu dem Attribut gehört (<Knotenname>/<Attributname>). Ist dies der Fall werden die Daten deserialisiert und im

zugehörigen Objekt (der Klasse RemoteValueReadOnly) unter „cachedValue“ gespeichert, sowie „hasValue“ auf *true* gesetzt.
Soll eventgesteuert auf eine Änderung des Attributes reagiert werden, wird außerdem die Funktion <Knotenname>::onChange_<Attributname>() ausgeführt. Abbildung 4.10 zeigt den Teil der C++ Template Datei, der für das Lesen von Attributen zuständig ist.

```
1     template<typename T>
2 class RemoteValueReadOnly
3 {
4     friend void updateValues(char* topic, byte* message, unsigned int
          ↪ length);
5 protected:
6     T cachedValue;
7     bool hasValue = false;
8 public:
9     bool get(T& value)
10    {
11        if (hasValue)
12        {
13            value = cachedValue;
14            return true;
15        }
16        return false;
17    }
18
19    RemoteValueReadOnly() = default;
20 };
```

Abbildung 4.9: Zugriff auf Attribute bei MQTT

```
1
2  void updateValues(char* topic, byte* message, unsigned int length)
3  {
4  {% for node in otherNodes + [thisNode] %}
5      {% for v in node.variables %}
6      if (!strcmp(topic, "{{node.name}}/{{v.name}}"))
7      {
8          {{node.name}}::{{v.name}}.cachedValue =
9              std::get<0>(deserialize<{{v.type}}>(message));
10         {{node.name}}::{{v.name}}.hasValue = true;
11         {% if v.isObserved is defined %}
12         {{node.name}}::onChange_{{v.name}}(
13             {{node.name}}::{{v.name}}.cachedValue);
14         {% endif %}
15         return;
16     }
17     {% endfor %}
18
19 {% endfor %}
20     ....
21 }
```

Abbildung 4.10: Verarbeiten von empfangenen Nachrichten bei MQTT

Methoden

Für die Implementierung von *Methoden* über MQTT wird die Klasse „RemoteFunction“ von der Klasse „RemoteFunctionAbstract“ abgeleitet und das MQTT spezifische senden von Daten implementiert (Abbildung 4.11). Außerdem werden in der Datei *Internal.cpp* mittels Jinja2-Templates Codeblöcke erzeugt, die für das Initialisieren des Objektes (in der Funktion „initializeValues()“), sowie das Zustellen eines Empfangenen Rückgabewertes (in der Funktion „updateValues()“) zuständig sind. Dies ist analog zu dem, in Abbildung 4.10 dargestellten, Empfangen der Werte eines Attributes, und wird daher nicht genauer beschrieben.

```
1
2  template <typename R, typename... Args>
3  class RemoteFunction : public RemoteFunctionAbstract<R, Args...>
4  {
5  public:
6      PubSubClient* client;
7      const char* topic;
8
9      virtual void sendData(std::vector<uint8_t>& data)
10     {
11         client->publish(topic, data.data(), data.size());
12     }
13
14     RemoteFunction() = default;
15 };
```

Abbildung 4.11: MQTT spezifisches Senden von Methodenaufruf

4.1.6 BLE

Bei Bluetooth Low Energy muss der Codegenerator, neben den Arduino Projekten für die Knoten, auch Arduino Projekte für (einen oder mehrere) BLE-Server erzeugen. Für die Implementierung der BLE-Knoten, als auch des BLE-Servers, wurde die „ESP32 BLE Arduino“ Bibliothek [3] verwendet. Um einen BLE-Server mit anderen BLE-Servern oder MQTT-Knoten zu verbinden, wird (wie in Abschnitt 3.6 erläutert) außerdem eine BLE-Bridge benötigt.

Der BLE-Server

Als BLE-Server wird (wie für die Knoten) ein ESP32 verwendet, auf den der vom Codegenerator erzeugt Code (ohne Modifikation) geladen wird. Wie bereits im Konzept erläutert wurde, kann es mehrere BLE-Server geben, damit BLE-fähige Geräte an unterschiedlichen Standorten betrieben werden können.
Der BLE-Server enthält einen „BLE Service“, welcher wiederum für jedes *MKP-Datenfeld* eine BLE-Characteristic enthält. Bei Bluetooth Low Energy wird jeder Service bzw. jede Characteristic durch einen 128-Bit großen Integer identifiziert, welcher UUID genannt wird. Die UUID muss somit als *MKP-Datenfeld-ID* des minimalen Kommunikationsprotokolles verwendet werden.
Um eindeutige UUID's zu generieren, wird die MD5 Hashfunktion verwendet. Dazu wird zuerst aus Knotenname und Attributname (bzw. Methodenname) ein String gebildet (<Knotenname>::<Attributname>) und aus diesem, zusammen mit einem zufällig gewählten Startwert (genannt *seed*), ein Hashwert errechnet. Die Wahrscheinlichkeit einer Kollision ist dabei so ge-

ring, dass diese nicht weiter betrachtet wird.[6]
Der Vorteil eine UUID mittels Hashfunktion zu erzeugen, liegt darin, dass bei einer erneuten Ausführung des Codegenerators die UUIDs gleich bleiben. Dies ermöglicht es z.B. später ein Gerät hinzuzufügen, ohne die bereits bestehenden Geräte neu programmieren zu müssen. Auch die UUID des BLE Service wird mittels der MD5 Hashfunktion aus dem „Servernamen“ gebildet. Der Server gibt durch „Advertising“ die UUID des Service bekannt. Dadurch kann ein *Knoten* beim Scannen der Umgebung erkennen, dass dieses Gerät der gesuchte BLE-Server ist.

Für jeden BLE-Server wird ein Arduino Projekt erzeugt, welches eine Datei namens <Servername>.ino enthält. Das Projekt kann auf einen ESP32 geladen werden, in dem er als BLE-Server agiert.

Attribute

Um den Wert eines *Attributes* vom Server zu lesen, wird (wie in Abbildung 4.12 dargestellt) die „BLE::RemoteCharacteristic::readValue()“ Methode verwendet.

```
1  template<typename T>
2  class RemoteValueReadOnly
3  {
4  protected:
5      BLERemoteCharacteristic* characteristic = nullptr;
6  public:
7      bool get(T& value)
8      {
9          std::string rawValue = characteristic->readValue();
10         if (rawValue.size() == 0)
11         {
12             return false;
13         }
14         value =
               ↪ std::get<0>(deserialize<T>((uint8_t*)rawValue.data()));
15         return true;
16     }
17 };
```

Abbildung 4.12: Zugriff auf Attribute bei BLE

Soll ein Attribut beobachtet werden, muss auf „Notifications“ vom Server reagiert werden. Dies wird durch hinterlegen einer Callbackfunktion bei einem zur Characteristic gehörendem Objekt (vom Typ BLERemoteCharacteristic) realisiert.

[6] Bei zwei Hashwerten liegt die Wahrscheinlichkeit einer Kollision bei $\frac{1}{2}^{128}$. Bei 2^{64} Werten besteht eine 50 prozentige Chance mindestens eine Kollsion zu erhalten.

Hier musste eine Besonderheit der verwendeten Arduino-Bibliothek beachtet werden: Die Bibliothek stellt bei Notifications lediglich die ersten 20 Bytes an Daten zu. Daher muss bei größeren Datenmengen, der Wert, nach Erhalt einer Notification, manuell gelesen werden.
Ein Aufruf der „BLERemoteCharacteristic::readValue()“-Funktion innerhalb des Callbacks führt jedoch zu einem Deadlock. Um dies zu vermeiden wird innerhalb des Callbacks eine Lambda Funktion erstellt. Die Funktion wird (durch Semaphoren geschützt) in einer Liste abgespeichert und später in einem anderen Thread ausgeführt.
Abbildung 4.13 zeigt die in der Datei *Internal.h* definierte Callbackfunktion. Die bei Änderung des Attributes aufgerufene Funktion (<Knotenname>::onChange_<Attributname>) wird beim hinterlegen des Callbacks als Templateparameter übergeben.

```
1  template<typename T, void (*Fun)(T)>
2  void notifyCallback(BLERemoteCharacteristic* remoteCharacteristic,
       ↪ uint8_t* data, size_t length, bool isNotify)
3  {
4      taskBuffer.addTask([=](){
5          std::vector<uint8_t> rawData;
6          if (length >= 20)
7          {
8              std::string longData = remoteCharacteristic->readValue();
9              rawData = std::vector<uint8_t>(longData.begin(),
                   ↪ longData.end());
10         }
11         else
12         {
13             rawData = std::vector<uint8_t>(data, data + length);
14         }
15         auto deserialized = deserialize<T>(rawData);
16         if (deserialized.has_value())
17         {
18             Fun(std::get<0>(deserialized));
19         }
20     });
21 }
```

Abbildung 4.13: Callbackfunktion für beobachtete Attribute

Methoden

Auch beim Aufruf von *Methoden* werden Notifications verwendet, um ein Beschreiben von Call-Feld und Return-Feld zu signalisieren. Da die Signatur der Callbackfunktion von der verwendeten Arduino BLE-Bibliothek vorgegeben wird, werden alle zusätzlich benötigten Informationen als Template Parameter beim hinterlegen des Callbacks übergebeben.
Abbildung 4.14 zeigt die (in der Dati *Internal.h* definierten) Callback-Funktionen für Call-Feld und Return-Feld. Wie bei der Callback-Funktion für *Attribute* beschrieben, wird hier ebenfalls

eine Lambda-Funktion erzeugt, die später in einem anderen Thread ausgeführt wird.

```cpp
template<typename Fun, Fun* remoteFunction>
void notifyReturnCallback(BLERemoteCharacteristic*
    ↪ remoteCharacteristic, uint8_t* data, size_t length, bool
    ↪ isNotify)
{
    taskBuffer.addTask([=](){
        std::vector<uint8_t> rawData;
        if (length >= 20)
        {
            std::string longData = remoteCharacteristic->readValue();
            rawData = std::vector<uint8_t>(longData.begin(),
                ↪ longData.end());
        }
        else
        {
            rawData = std::vector<uint8_t>(data, data + length);
        }
        remoteFunction->pickUpResult(rawData);
    });
}

template<BLERemoteCharacteristic** returnCharacteristic, typename
    ↪ Fun, Fun* localFunction, typename... Args>
void notifyCallCallback(BLERemoteCharacteristic*
    ↪ remoteCharacteristic, uint8_t* data, size_t length, bool
    ↪ isNotify)
{
    taskBuffer.addTask([=](){
        std::vector<uint8_t> rawData;
        if (length >= 20)
        {
            std::string longData = remoteCharacteristic->readValue();
            rawData = std::vector<uint8_t>(longData.begin(),
                ↪ longData.end());
        }
        else
        {
            rawData = std::vector<uint8_t>(data, data + length);
        }
        auto result = processFunctionCall<Fun*, Args...>(rawData,
            ↪ localFunction);
        if (!result)
        {
            return;
        }
        (*returnCharacteristic)->writeValue(result.value().data(),
            ↪ result.value().size());
    });
}
```

Abbildung 4.14: Callbackfunktion für Methodenaufrufe

4.2 BLE-Bridge

Die *BLE-Bridge* wurde in Python implementiert und ist eine vom Codegenerator unabhängige Software. Die Implementierung der BLE-Bridge befindet sich in der Datei „BLEBridge.py“ im Verzeichnis „BLEBridge“. Ein ESP32 lässt sich für die *BLE-Bridge* nicht ohne weiteres verwenden, da beim ESP32 Wifi und BLE mit dem selben Sender und Empfänger arbeiten und dadurch nicht gleichzeitig aktiv sein können.
Daher wird für den Betrieb der BLE-Bridge ein Gerät benötigt, dem es möglich ist, sich mit dem BLE-Server zu verbinden und gleichzeitig über TCP/IP eine Verbindung mit dem MQTT-Broker aufrechtzuerhalten.[7]
Für die Kommunikation über BLE und MQTT wurden die Python Bibliotheken „bleak“ und „phao-mqqt“ verwendet. Diese sind mit den Betriebssystemen Linux, Mac OS und Windows kompatibel. Bei Verwendung der „bleak“-Bibliothek gibt es jedoch einen Linux-spezifischen Funktionsaufruf, so dass die hier implementierte BLE-Bridge nur auf Linux Rechnern ausgeführt werden kann.

Die BLE-Bridge liest die Konfigurationsdatei und verbindet sich anschließend mit einem BLE-Server und dem MQTT-Broker. Es werden die BLE-Characteristics auf dem BLE-Server beobachtet (durch Notifications) und bei einer Änderung wird eine entprechende Nachricht an den MQTT-Broker erzeugt. Analog dazu werden MQTT-Topics beobachtet („subscribe“) und bei Erhalt einer Nachricht, der Wert, in die entsprchende Characteristic auf dem BLE-Server geschrieben.

4.3 Das Webinterface

Viele IoT-Anwendungen (z.B. Smart Home) bieten die Möglichkeit über ein „Webinterface“ auf IoT-Geräte zuzugreifen, deren Status zu überwachen und Einstellungen vorzunehmen. Es wurde eine als „Webinterface“ bezeichnete Software implementiert, die ein solches Webinterface realisiert.
Dieses speichert die Werte aller Attribute in einer SQLite[29] Datenbank. Über einen Webbrowser kann zu jedem Attribut ein Diagramm aufgerufen werden, das eine Zeitreihe der Werte anzeigt. Außerdem ist es über den Webbrowser möglich, die Methoden von Knoten aufzurufen, wodurch eine manuelle Steuerung ermöglicht wird.

Das Webinterface ist eine vom Codegenerator unabhängige Software und die Verwendung ist rein optional. Da alle Daten über den MQTT-Broker übertragen werden (auch bei BLE durch die BLE-Bridge), kann das Webinterface über den MQTT-Broker die Attribute aller Knoten

[7] In Kapitel 5 (Anwendung) wird zu diesem Zweck ein Raspberry Pi verwendet.

beobachten und Methoden aufrufen. Dazu werden die in der Konfigurationsdatei enthaltenen Informationen benötigt.

Da das Webinterface lediglich dazu dient, das in dieser Arbeit vorgestellt Konzept zu ergänzen, stand bei der Implementierung im Vordergrund, die gewünschte Funktionalität möglichst einfach umzusetzen.
Aus diesem Grund wurde das Webinterface in Python mit Hilfe des Web-Frameworks „Flask“[27] implementiert. Dieses verwendet für die dynamische Generierung von HTML Dokumenten die (auch vom Codegenerator verwendete) Template Engine Jinja2.
Für die Gestaltung der grafischen Oberfläche wurde die Javascript-Bibliothek „htmx“[32] verwendet. Diese erweitert HTML um Funktionen, für die normalerweise JavaScript verwendet werden muss (z.B. POST Request bei drücken eines Buttons). Dadurch ist es möglich die grafische Oberfläche im deklarativen Stil von HTML, ohne eigenen JavaScript Code, zu implementieren.
Für das Design wurde die CSS-Bibliothek „picnic-css“[26] verwendet. Die Biblitohek hat bereits vordefinierte Designs für häufig verwendete HTML-Elemente (wie button, table, lable ...).
Für das Erzeugen der Diagramme wurde die Python Bibliothek „Bokeh“ [14] benutzt. „Bokeh“ ermöglicht es, einen interaktiven Plot als HTML-Dokument zu exportieren. Dadurch ist es möglich in das, auf der Website dargestellte, Diagramm zu zoomen und gewählte Ausschnitte als Bilddatei abzuspeichern.

Das Webinterface kann im Verzeichnis „Webinterface“ gefunden werden und durch das Ausführen der Datei gateway.py gestartet werden.
Die Datei serialize.py enthält Code für das Serialisieren/Deserialisieren von Daten, analog zu der in Abschnitt 4.1.2 beschriebenen Implementierung in C++.
Im Unterverzeichnis „templates“ befinden sich HTML-Dateien, die Jinja2 Templates enthalten. Diese werden zur Erzeugung der grafischen Oberfläche der GUI verwendet.

5 Anwendung des Codegenerators

Anhand der Implementierung sollen die Vor- und Nachteile des vorgestellten Konzepts betrachtet werden, indem die Funktion des Codegenerators an realer Hardware untersucht wird. Folgende Geräte werden verwendet:

- **ESP32 Mikrokontroller (4 Stück):**[8] Hardware für IoT-Geräte sowie BLE-Server
- **Raspberry Pi 3:**[9] Wird für die BLE-Bridge verwendet
- **Wlan Router (Fritz Box):** benötigt für WiFi-Verbindung zwischen MQTT-Geräten
- **Windows 10 Desktop Rechner:** Betrieb von MQTT-Broker (Eclipse Mosquitto), *Webinterface* und Webbrowser
- **MCP9808 Temperatursensor (2 Stück):** Wird über I2C mit dem ESP32 verbunden.
- **SG51R Mikro-Servomotor:** Wird über PWM vom ESP32 angesteuert.
- **16*2 LCD Display (Hersteller Joy-IT):** Wird über I2C mit dem ESP32 verbunden.

Mit vier ESP32 können, je nachdem ob ein BLE-Server benötigt wird, maximal drei bis vier Knoten in einem Besipiel verwendet werden. Üblicherweise besitzen IoT-Anwendungen über wesentlich mehr Geräte, aus Kosten- und Resourcengründen wurde jedoch auf den Einsatz von mehr Geräten verzichtet.

Es werden drei Beispielprojekte vorgestellt. Die „temperaturabhängige Fenstersteuerung“ zeigt eine Anwendung die zwei Sensoren, einen Aktor und ein Gerät mit Steuerlogik beinhaltet. Bei dem an *Smart Home* angelehnten Projekt wird gezeigt, wie mit der implementierten Software (Codegenerator, Webinterface und BLE-Bridge) eine IoT-Anwendung realisiert werden kann.
Anhand von zwei weiteren kurzen Beispielprojekten wird die, in Abschnitt 3.9 diskutierte, Übertragung von größeren Datenmengen zwischen zwei Knoten gezeigt. Ein Beispielprojekt veranschaulicht die in 3.9 beschriebene „Fragmentierung von Attributen“. Das Andere zeigt, wie mit einer *Methode*, beliebig große Datenmengen zwischen zwei Knoten übertragen werden können.

[8] Hier eingesetzt wird das Entwicklungsboard ESP32-DevKitC Ver. D
[9] genau: Raspberry Pi3 Model B mit Betriebssystem: Raspbian GNU/Linux 11 (bullseye)

5.1 Temperaturabhängige Fenstersteuerung

In der Beispielanwendung soll eine Fenstersteuerung entwickelt werden, die abhängig von der Außen- und Innentemperatur automatisch ein Fenster öffnet bzw. schließt. Dazu hat der Anwender 4 Geräte vorgesehen. Bei den Geräten handelt es sich um einen Außen- und einen Innentemperatursensor, den Motor zur Fenstersteuerung, und eine Steuereinheit. Außen- und Innentemperatursensor werden mit einem MCP9808 Temperaturfühler realisiert. Das Öffnen bzw. Schließen des Fensters wird simuliert, indem ein kleiner Servomotor angesteuert wird. Die Steuereinheit zeigt die Temperaturwerte über ein LCD-Display an.

5.1.1 Erstellen des Modells

Wie bereits erwähnt, wurde das Datenmodell an die objektorientierte Programmierung angelehnt. Der Anwender (hier der Programmierer der Fenstersteuerung) kann, mit dem in 5.1 Abbildung dargestellten UML-Diagramm, die Interaktion der Geräte modellieren.
Jedes Gerät (analog zu Objekt) besitzt dementsprechend eine Definition seiner Schnittstelle (analog zu Klassendefinition). Das Diagramm kann daher als Objektdiagramm angesehen werden, da es die Interaktion zwischen den Geräten beschreibt.
Es werden gerichtete Assoziationen verwendet, da bei dieser Anwendung lediglich die Steuereinheit auf andere Geräte zugreifen muss.

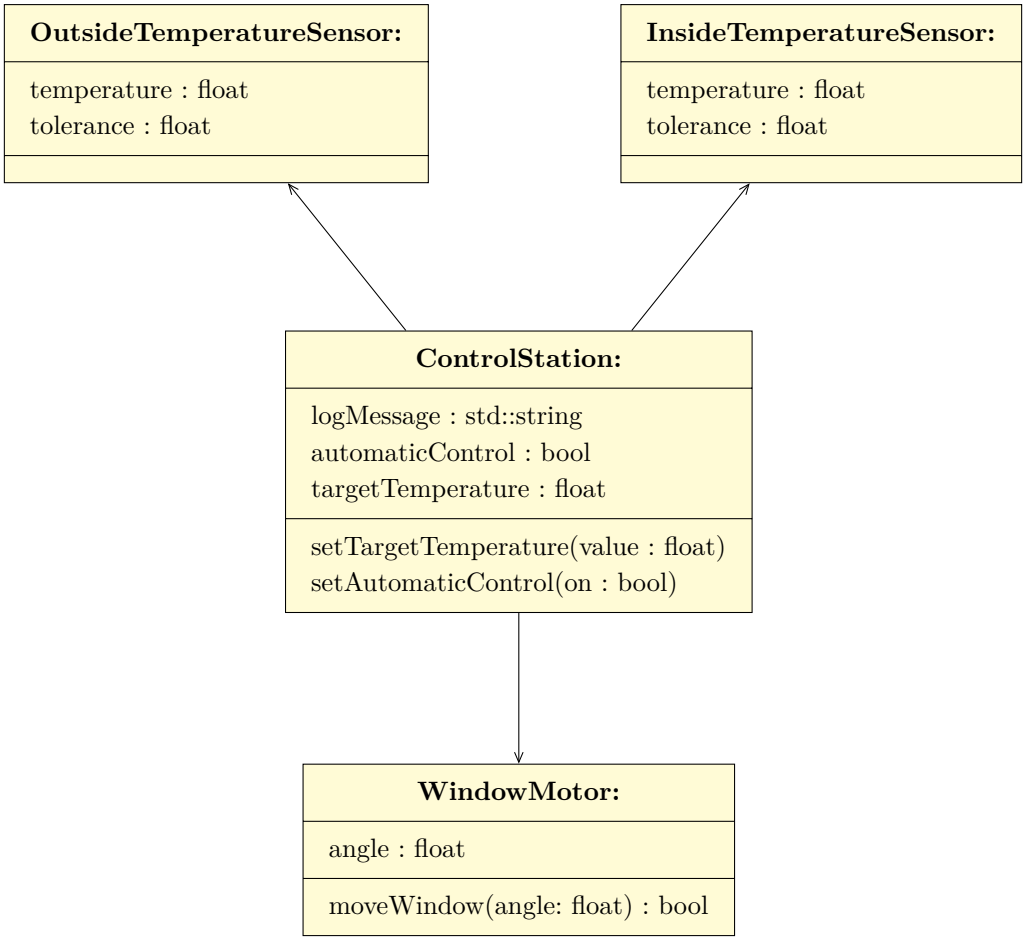

Abbildung 5.1: Objektdiagramm Fenstersteuerung

Abhängig von der gemessenen Außen- und Innentemperatur soll der Fenstermotor gesteuert werden. Wenn die Temperatur im Außenbereich näher an einer vom Nutzer der IoT-Anwendung definierten Zieltemperatur liegt, soll das Fenster geschlossen sein. Liegt die Außentemperatur hingegen näher an dieser Zieltemperatur, soll das Fenster geöffnet werden. Um dies zu erreichen, überwacht die Kontrollstation die Geräte mit den verbauten Temperatursensoren und sendet dem Fenstermotor, abhängig der Messergebnisse, Befehle zum öffnen oder schließen. Die Zieltemperatur kann mit der setTargetTemperatur-Methode eingestellt werden. Mit setAutomaticControl (on/off) lässt sich die automatische Steuerung aktivieren oder deaktivieren.
Aufgrund der gerichteten Assoziationen kann kein Gerät im Modell auf die *Attribute* und *Methoden* der Kontrollstation zugreifen. Durch Verwendung des Webinterfaces kann der Nutzer jedoch manuell *Methoden* aufrufen und die Kontrollstation bedienen.
Das Attribut „logMessage“ ist für die Fehlerbehandlung vorgesehen. Bei Verwendung des Webinterfaces speichert dieses die Werte aller Attribute in einer Datenbank (siehe Abschnitt 4.3). Dadurch können alle Log-Nachrichten über das Webinterface (zusammen mit einem Zeitstempel) eingesehen werden. Das Logging wird nur bei einem Knoten gezeigt, um das Beispiel nicht

unnötig zu verlängern.

5.1.2 Erstellen der Konfigurationsdatei für die Fenstersteuerung

Der Nutzer erstellt eine JSON-Konfigurationsdatei, in der beschrieben wird, über welche *Attribute* und *Methoden* die Knoten der Fenstersteuerung verfügen und wie diese miteinander interagieren können.
Das Gerüst des Außentemperatursensors wird wie in der folgenden Abbildung (5.2) erzeugt. Die Konfiguration des Innentemperatursensors ist, bis auf den Namen, welcher zur eindeutigen Identifizierung dient, identisch und wird daher nicht explizit gezeigt.
Unter „communication-protocol“ (Zeile 3) werden Informationen für die Netzwerkverbindung (z.B. zu verwendendes Protokoll) hinterlegt. Dies wird später genauer erläutert und daher bei den folgenden *Knoten* zur Vereinfachung des Beispiels ausgelassen.
Der Außentemperatursensor verfügt lediglich über zwei *Attribute*, „temperature“ und „tolerance“, welche beide vom Typ „float“ sind.

```
1  {
2      "name" : "OutsideTemperatureSensor",
3      "communication_protocol" :
4      {
5          ...
6      },
7      "variables" :
8      [
9          {
10             "name" : "temperature",
11             "type" : "float"
12         },
13         {
14             "name" : "tolerance",
15             "type" : "float"
16         }
17     ]
18 }
```

Abbildung 5.2: Konfiguration Außentemperatursensor

Der Motor zur Fenstersteuerung (Abbildung 5.3) stellt über die *Methode* „bool moveWindow(float angle)“ eine Schnittstelle zur Ansteuerung des Fensters zur Verfügung. Über den Parameter „angle“ wird der Öffungswinkel angegeben. Die Funktion soll *true* zurückgeben, wenn das Fenster erfolgreich zum neuen Öffnungswinkel bewegt wurde. Das *Attribut* „angle“ zeigt den aktuellen Öffnungswinkel des Fensters an.

```
{
    "name" : "WindowMotor",
    "variables" :
    [
        {
            "name" : "angle",
            "type" : "float"
        }
    ],
    "functions" :
    [
        {
            "name" : "moveWindow",
            "params" :
            [
                {
                    "name" : "targetAngle",
                    "type" : "float"
                }
            ],
            "returnType" : "bool"
        }
    ]
}
```

Abbildung 5.3: Konfiguration Fenstermotor

Die Steuereinheit (Abbildung 5.4) soll die beiden Temperaturwerte vergleichen und je nach Wetterlage das Fenster öffnen bzw. schließen. Um auf andere *Knoten* im Netz zugreifen zu können, muss dies in der Konfiguration unter „using“ festgelegt werden. Es kann nur auf *Attribute* und *Methoden* von Knoten zugegriffen werden, die unter „using“ aufgelistet sind.
Für die bereits beschriebene Möglichkeit, eventgesteuert auf die Änderung von Werten zu reagiern, werden unter „observe“ alle *Attribute* aufgelistet, die „beobachtet“ werden sollen.

```
1
2 {
3     "name" : "ControlStation",
4     "variables" :
5     [
6         ...
7     ],
8     "functions" :
9     [
10        ...
11    ]
12    "using" : [
13        "OutsideTemperatureSensor",
14        "InsideTemperatureSensor",
15        "WindowMotor"
16    ],
17    "observe" : [
18        "OutsideTemperatureSensor::temperature"
19    ]
20 },
```

Abbildung 5.4: Konfiguration Steuereinheit

5.1.3 Befüllen der Programmgrundgerüste

Nachdem die Konfigurationsdatei erstellt wurde, führt der Anwender den Codegenerator aus. Der Codegenerator erzeugt daraufhin ein Arduino-Projekt für jeden *Knoten*. In der Datei CustomCode.cpp kann der Anwender C++ Code einfügen um die gewünschte Funktionalität des Knotens implementieren. Es können auch eigene .cpp und .h Datein angelegt werden, um große Projekte besser zu strukturieren.

Die Temperatursensoren müssen die Werte der *Attribute* „temperature“ und „tolerance“ bereitstellen. Die CustomCode.cpp der Temperatursensoren sieht nach dem Befüllen der Funktionsrümpfe durch den Nutzer wie in Abbildung 5.5 aus. Da sich die Toleranz nicht ändert, wird der Wert nur einmal beim Verbindungsaufbau gesetzt. Solange der Temperatursensor mit dem Netzwerk verbunden ist, stellt er alle 5 Sekunden einen neuen Temperaturwert zur Verfügung. Wie genau der Temperaturwert gelesen wird, hängt von der verwendeten Hardware ab und wird hier nicht genauer dargestellt.

```
1  void Loop()
2  {
3      if (isConnected())
4      {
5          float currentTemp = tempsensor.readTempC();
6          Serial.print("New temperature is: ");
7          Serial.println(currentTemp);
8          temperature.set(currentTemp);
9      }
10     else
11     {
12         Serial.println("No connection to server");
13     }
14     delay(10000);
15 }
16
17 void OnConnect()
18 {
19     Serial.println("Connected to server");
20     tolerance.set(0.1);
21 }
```

Abbildung 5.5: CustomCode.cpp Temperatursensor

Beim Fenstermotor (Abbildung 5.6) enthält die Datei „CustomCode.cpp“ zusätzlich einen Funktionsrumpf für die Methode „moveWindow“. Mit hardwarespezifischem Code wird der Motor angesteuert (Zeile 4) und anschließend wird der Wert des Attributes „angle“ aktualisiert (Zeile 5).

```
1 bool WindowMotor::moveWindow(float targetAngle)
2 {
3     float degree = targetAngle * 360 - 180;
4     servoMotor.write(degree);
5     angle.set(targetAngle);
6     return true;
7 }
```

Abbildung 5.6: CustomCode.cpp Fenstermotor

Die Steuereinheit liest nun die Temperaturwerte und nutzt die „moveWindow()“ Methode um das Fenster zu öffnen oder zu schließen. Da die Werte von Attributen auf dem Server gespeichert werden, ist es möglich, dass der Außentemperatursensor seit langer Zeit defekt ist und ein veralteter Wert gelesen wird. Um zu wissen, wie aktuell der Wert des Außentemperatursensors ist, wird das *Attribut* beobachtet. Wird ein neuer Wert erhalten, erstellt die Steuereinheit einen Zeitstempel. Abbildung 5.7 zeigt, wie der Zeitstempel erstellt und überprüft wird.

```
unsigned long outsideTempLastUpdate = 0;

void Loop()
{
        ...

        if (millis() - outsideTempLastUpdate > 60000)
        {
            logMessage.set("Outside temperature is not actual.");
        }
        ...
}

void OutsideTemperatureSensor::onChange_temperature(float value)
{
    outsideTempLastUpdate = millis();
}
```

Abbildung 5.7: CustomCode.cpp ControlPanel

Damit eine Änderung des Wertes nicht verpasst wird, muss die Steuereinheit stets mit dem Server verbunden sein. Eine Alternative zur Beobachtung besteht darin, den Wert über eine Methode (z.B.„float OutsideTemperatureSensor::getTemperature()“) abzufragen um sicherzustellen, dass der Wert aktuell ist. Dies erfordert jedoch, dass der Außentemperatursensor jederzeit erreichbar ist um den Aufruf der Methode zu erhalten.

Attribute mit Zeitstempeln

Das im Kapitel der Implementierung vorgestellte *Webinterface* stellt ein zusätzliches Tool dar, welches die Werte von allen Attributen mit Zeitstempel versehen, in einer Datenbank abspeichert. Diese Funktionalität wurde in ein optionales Tool ausgelagert, da dies einen recht hohen Overhead erzeugt. Die verwendeten Microcontroller besitzen keine eingebaute Uhr, daher müssten diese, bei der Verwendung von Zeitstempeln, nach dem Start zuerst die aktuelle Zeit von einem anderen Gerät abfragen. Es ist nicht möglich, dass dies der *Server* des *minimalen Kommunikationsprotokolles* übernimmt, da hier bei MQTT der MQTT-Broker verwendet wird, dessen Funktionalität nicht angepasst werden kann.

5.1.4 Auswahl der eingesetzten Kommunikationsprotokolle

Die in Abschnitt 5.1.2 beschriebene Konfigurationsdatei benötigt noch Informationen über die von den Knoten verwendeten Kommunikationsprotolle. Das Beispielprojekt enthält dabei zwei unterschiedliche Versionen (in den Dateien „Konfiguration1.json“ und „Konfiguration2.json“). In einer Version verwenden alle Knoten MQTT, in der Anderen verwenden der Außentempera-

tursensor und die Fenstersteuerung BLE.

Mit diesen beiden Versionen wird die „Variation der Kommunikationsprotokolle“ gezeigt. Es ist möglich, dass Geräte, die unterschiedliche Kommunikationsprotokolle verwenden, miteinander kommunizieren. Durch erneutes ausführen des Codegenerators, kann auch das von einem Gerät verwendete Kommunkikationsprotokoll geändert werden, ohne das manuell Quellcode geändert werden muss.

In den Abbildungen 5.8 und 5.9 wird dargestellt, wie die verwendeten Geräte, in den beiden unterschiedlichen Versionen, miteinander kommunizieren. Da Bluetooth Low Energy sowohl die physikalische Datenübertragung als auch die Anwendungsebene spezifiziert, wird die physikalische Verbindung in der Abbildung mit „BLE phys“ bezeichnet.

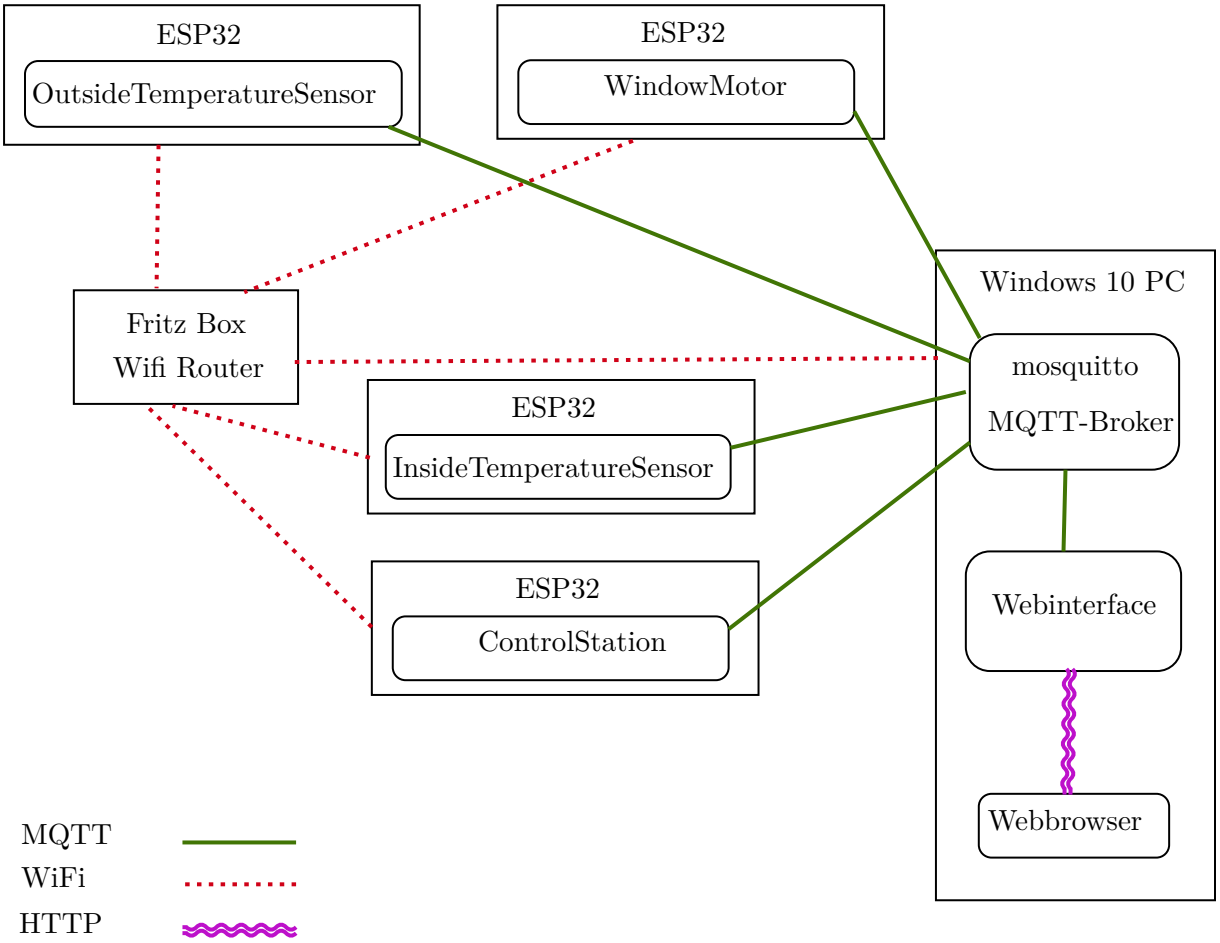

Abbildung 5.8: verwendete Geräte und Komponenten Version 1

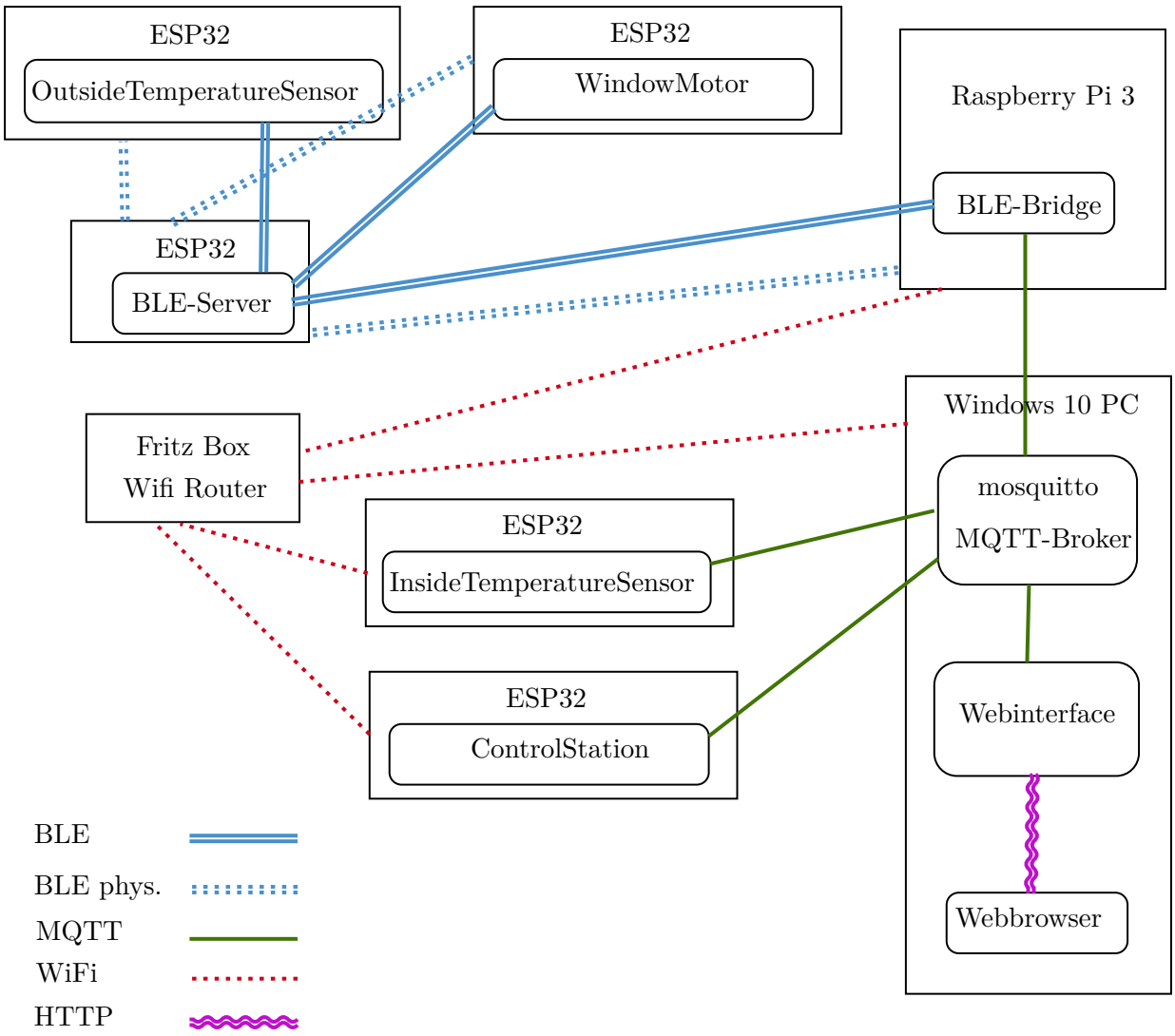

Abbildung 5.9: verwendete Geräte und Komponenten Version 2

Damit Knoten, die BLE verwenden, mit Knoten kommunizeren können, die MQTT verwenden, wird in der zweiten Version die *BLE-Bidge* verwendet, die den *BLE-Server* mit dem *MQTT-Broker* verbindet.

5.1.5 Testaufbau

Abbildung 5.10 zeigt ein Bild des Testaufbaus, während des Betriebes der Anwendung. Zu sehen sind (von links nach rechts) die *Steuereinheit* mit LCD-Display, der *BLE-Server*, der *Fenstermotor*, sowie Innen- und Außentemperatursensor.

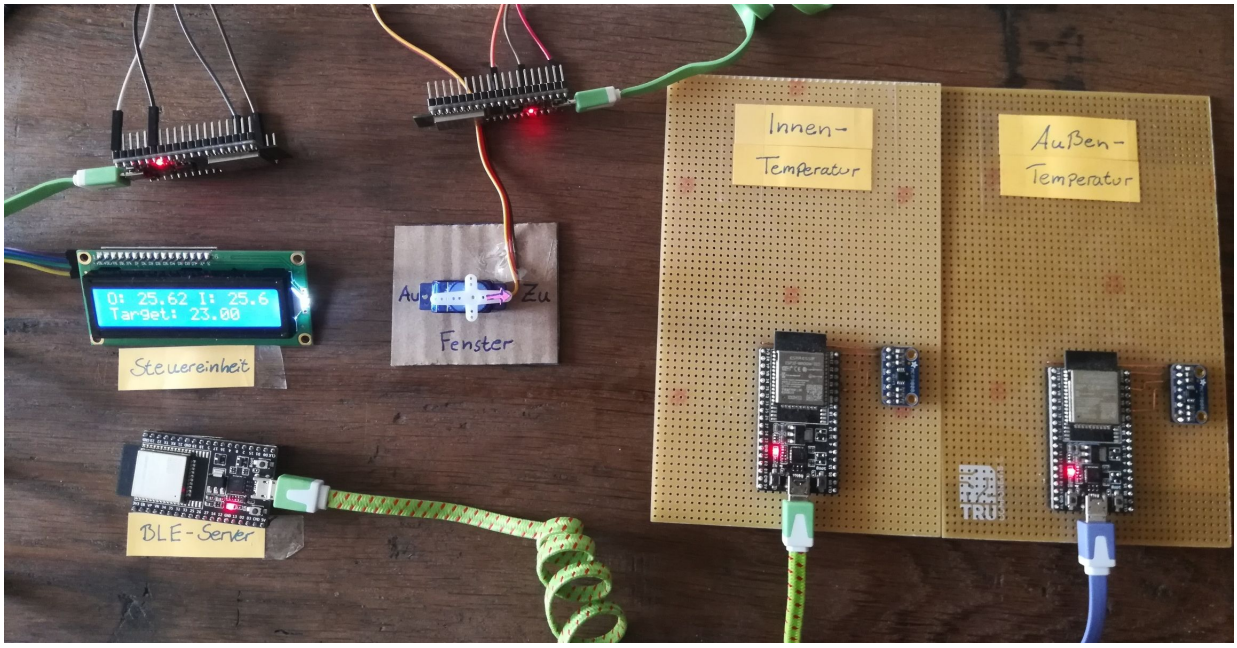

Abbildung 5.10: Testaufbau

5.1.6 Webinterface

Nachdem die Programme auf die Mikrokontroller geladen wurden und alle Geräte eingeschaltet sind, kann das *Webinterface* über einen Webbrowser geöffnet werden.
Auf der Website erfolgt eine Auflistung von *Attributen* und *Methoden* aller Knoten. Durch Anklicken eines Attributes wird die (in der Datenbank gespeicherte) Zeitreihe angezeigt. Bei Zahlenwerten wird ein interaktiver Plot mit Zoomfunktion dargestellt (Abbildung 5.12). Arrays und Strings werden in Form einer Liste, mit den dazugehörigen Zeitstempeln, ausgegeben (Abbildung 5.13).
Beim Anklicken einer *Methode* wird ein Formular geöffnet (Abbildung 5.14). Der Nutzer kann die *Methode* manuell aufrufen, indem er zuerst die Werte der Parameter (wenn vorhanden) einträgt und danach den Call-Button betätigt.

IoT Dashboard

Variable Monitor

OutsideTemperatureSensor	
temperature	20.375
tolerance	0.10000000149011612
InsideTemperatureSensor	
temperature	26.75
tolerance	0.10000000149011612
ControlStation	
logMessage	Connected to server
automaticControl	True
targetTemperature	1.0
WindowMotor	
angle	1.0

Functions

ControlStation
void setTargetTemperature(float value)
void setAutomaticControl(bool on)
WindowMotor
bool moveWindow(float targetAngle)

Abbildung 5.11: Webschnitstelle

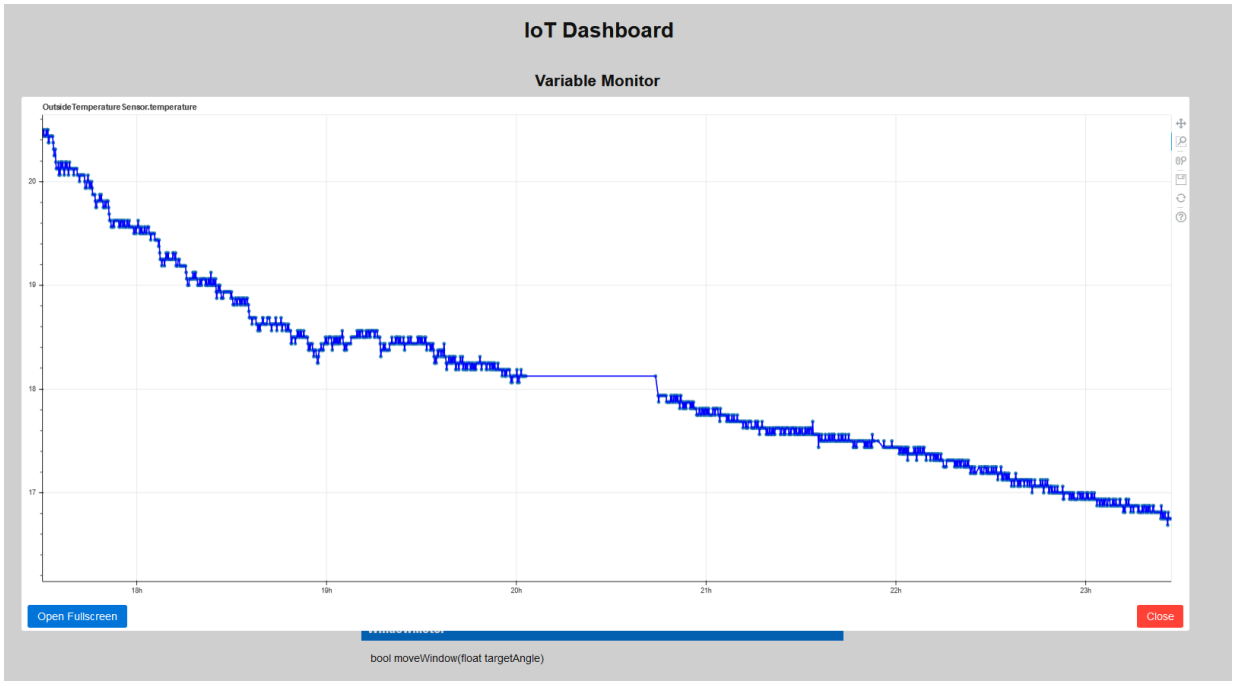

Abbildung 5.12: interaktiver Plot

ControlStation logMessage

Time	Value
2022-10-12 20:20:12	Open Window.
2022-10-12 20:20:12	Outside temperature is not actual.
2022-10-12 20:20:14	Open Window.
2022-10-12 20:20:14	Outside temperature is not actual.
2022-10-12 20:20:15	Open Window.
2022-10-12 20:20:16	Outside temperature is not actual.
2022-10-12 20:20:18	Open Window.
2022-10-12 20:20:18	Outside temperature is not actual.
2022-10-12 20:20:19	Open Window.
2022-10-12 20:20:19	Outside temperature is not actual.
2022-10-12 20:20:20	Open Window.
2022-10-12 20:20:21	Outside temperature is not actual.
2022-10-12 20:20:22	Open Window.
2022-10-12 20:20:22	Outside temperature is not actual.
2022-10-12 20:20:23	Open Window.
2022-10-12 20:20:24	Outside temperature is not actual.
2022-10-12 20:20:26	Open Window.

Abbildung 5.13: Log Nachrichten

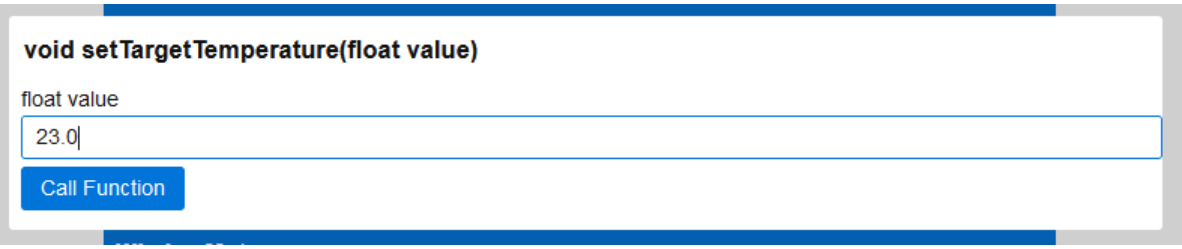

Abbildung 5.14: manueller Methodenaufruf über Formular

5.2 Fragmentierung

Bei diesem Projekt wird die in Abschnitt 3.9 beschriebene „Fragmentierung von Attributen“ getestet. Der Knoten mit dem Namen „Sender“ verfügt über ein Attribut vom Typ *std::vector<int>*, welches von dem Knoten namens „Receiver“ gelesen wird.
Damit für das *Attribut* mehrere *MKP-Datenfelder* verwendet werden, wird in der Konfigurationsdatei die Information hinterlegt, wie viele Datenfelder verwendet werden sollen und wie viel Bytes ein Datenfeld speichern kann. Dazu wird, wie in Abbildung 5.15 dargestellt, unter „composed“ die Anzahl („length“) und Kapazität („size“) der Datenfelder eingetragen.

Der „Sender“ verlängert den Vektor, indem er jede Sekunde eine Zahl anhängt. Der für das Attribut verfügbare Speicher auf dem *MKP-Server* ist jedoch begrenzt, weshalb es nach einer gewissen Zeit zu einem Fehler beim Lesen des Attributes kommt.
Der „Receiver“ liest den Wert des Attributs und zeigt diesen über die serielle Schnittstelle an. Wenn die Größe des Vektors den zur Verfügung stehenden Speicher übertrifft, kann der Fehler beim Lesen an dem übermittelten Hashwert (beschrieben in Abschnitt 3.9.1) erkannt werden.
Über die *log()* Funktion, übermittelt der generierte Code eine Fehlermeldung, die auch über die serielle Schnittstelle ausgegeben wird. Abbildung 5.16 zeigt eine mögliche Ausgabe des Programmes des Receivers über die Arduino Konsole. Im Beispiel stehen für das *Attribut* 5 * 64 = 320 Bytes zur Verfügung. Da 32 Bytes für den Hashwert verwendet werden und 4 Bytes für die Länge des Vektors, stehen 320 - 32 - 4 = 284 Bytes für die Elemente des Vektors zur Verfügung. Der Vektor kann somit maximal 284 / 4 = 71 Elemente enthalten, was mit der Ausgabe des Programmes übereinstimmt.
Da die „Fragmentierung von Attributen“ nur für BLE Knoten implementiert wurde, kann dieses Projekt nicht mit MQTT verwendet werden.

```
{
    "name" : "Sender",
    "communication_protocol" :
    {
        "name" : "BLE",
        "server" : "Server"
    },
    "variables" :
    [
        {
            "name" : "testVector",
            "type" : "std::vector<int>",
            "composed" : {
                "size" : 64,
                "length" : 5
            }
        }
    ]
}
```

Abbildung 5.15: Fragmentierung von Attributen: Konfigurationsdatei

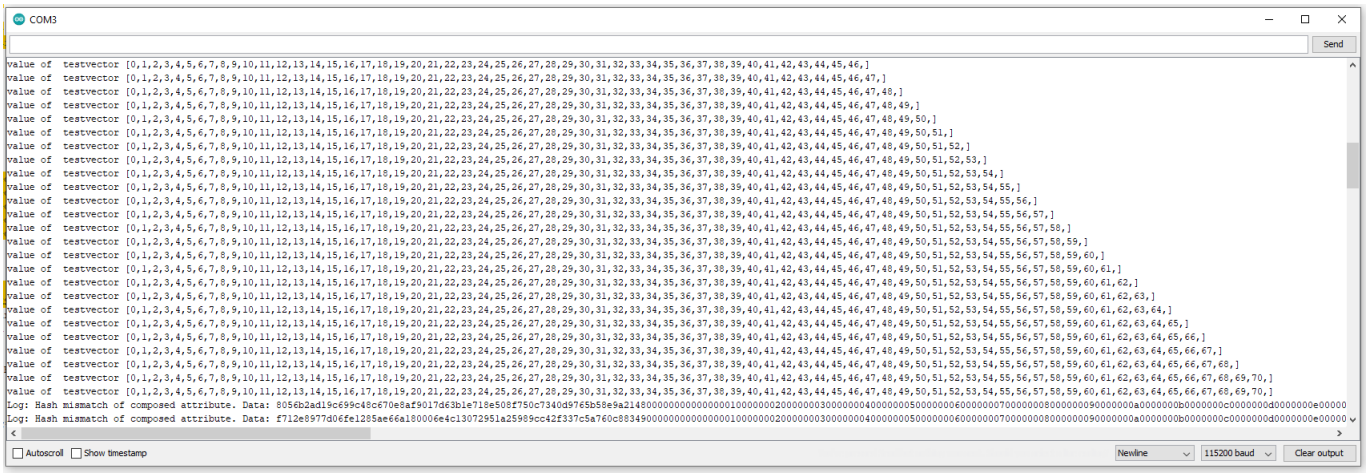

Abbildung 5.16: Ausgabe des „Receiver“ über Arduino Konsole

5.3 Streaming

Bei dieser Beispielanwendung wird der in Abschnitt 3.9.3 beschriebene Ansatz, zur Übertragung von beliebig große Datenmengen zwischen zwei Knoten, implementiert. Die Daten werden von einem Knoten („Sender“) zu einem anderen Knoten („Receiver“) durch den wiederholten Aufruf der Methode „inputStream()“ gesendet.
Der Receiver gibt die empfangenen Daten über die serielle Schnittstelle aus. Daran kann erkannt werden, dass die Daten erfolgreich übertragen wurden.

Abbildung 5.17 zeigt die vom „Receiver“ bereitgestellte Funktion „inputStream()“. In der Variable „buffer“ werden die empfangenen Daten angesammelt. Wenn es sich um das erste Datenpaket handelt, wird der Inhalt des Buffers gelöscht (Zeile 10). Wenn das letzte Datenpacket erhalten wurde, werden die empfangenen Daten in der Variable „receivedData“ abgespeichert (Zeile 22).

```
1
2  std::vector<uint8_t> buffer;
3  std::vector<uint8_t> receivedData;
4  constexpr int maxCapacity = 10000;
5
6  bool Receiver::inputStream(std::vector<uint8_t> data, bool
       ↪ isFirstPackage, bool isLastPackage)
7  {
8      if (isFirstPackage)
9      {
10         buffer.clear();
11     }
12
13     if (buffer.size() + data.size() > maxCapacity)
14     {
15         Serial.println("Error␣max␣buffer␣capacity␣exeeded.");
16         return false;
17     }
18
19     buffer.insert(buffer.end(), data.begin(), data.end());
20     if (isLastPackage)
21     {
22         receivedData = buffer;
23         printReceivedData();
24     }
25     return true;
26 }
```

Abbildung 5.17: Auszug CustomCode.cpp „Receiver“

6 Ergebnisse

Anhand des implementierten Codegenerators, sowie dem damit realisierten Beispielprojekt, wurde gezeigt, dass das vorgestellte Konzept zur „Variation der Kommunikationsprotokolle“ funktioniert und realisierbar ist. In dem Testprojekt „temperaturabhängige Fenstersteuerung“ wurde mit Hilfe des Codegenerators eine „Variation der Kommunikationsprotokolle“ erreicht. Einerseits indem IoT-Geräte, die mit unterschiedlichen Protokollen arbeiten, miteinander vernetzt wurden. Andererseits durch die Möglichkeit, durch erneutes Ausführen des Codegenerators, das von einem Gerät verwendete Kommunikationsprotokoll zu ändern.

Eine für batteriebetriebene IoT-Geräte wichtige Eigenschaft der verwendeten Kommunikationsprotokolle konnte beibehalten werden. Diese besteht darin, dass es möglich ist, dass sich ein IoT-Gerät nur kurz mit dem Netzwerk verbindet, Daten austauscht und danach wieder abschaltet.
Das Beispiel „temperaturabhängige Fenstersteuerung“ hat veranschaulicht, dass eine Aussage darüber getroffen werden kann, wie aktuell der (vom nur sporadisch verbundenen Sensor) gelieferte Wert ist. Ein anderes Gerät (im Beispiel die Steuereinheit) muss zu diesem Zweck stets mit dem Netzwerk verbunden sein und den, vom Sensor über ein Attribut bereitgestellten Wert, zu „beobachten“.
Es wäre auch möglich, dass sich die Steuereinheit nur gelegentlich mit dem Netzwerk verbindet. Um sicherzustellen, dass der vom Sensor gelieferte Wert aktuell ist, kann dieser Wert über eine Methode abgerufen werden. In diesem Fall muss jedoch der Sensor stets mit dem Netzwerk verbunden sein, ansonsten scheitert der Aufruf der Methode.

Das gewählte Datenmodell eignet sich damit gut für die Implementierung von Steuerungsaufgaben unter Einsatz von IoT-Geräten, da die Geräte einerseits ressourcenschonend verwendet werden können und es andererseits (durch *Methoden*, *Attribute* und das „Beobachten“ von *Attributen*) weitreichende Möglichkeiten bei der Programmierung der Geräte bietet.

Die Übertragung von größeren Datenmengen zwischen zwei Geräten, (wie in Abschnitt 3.9 diskutiert), wurde mit den Beispielprojekten *Fragmentierung* und *Streaming* getestet. Diese Form der Datenübertragung ist möglich, jedoch stets weniger effizient, als die Verwendung eines Kommunikationsprotokolls, das nicht für das Internet der Dinge konzipiert ist (z.B TCP/IP oder Bluetooth). Ein Grund dafür ist, dass bereits die verwendeten Kommunikationsprotokolle MQTT und BLE nicht dafür optimiert sind große Datenmengen zu übertragen.

Das verwendete Adressierungsschema ist statisch, das heißt, Geräte werden durch ihren Namen

identifiziert und können sich zur Laufzeit nur mit zuvor definierten Geräten verbinden. Dies macht manche IoT-Anwendungen unmöglich.[10]

Der in dieser Arbeit verwendete Ansatz der *modellgetriebenen Softwareentwicklung* bietet auch Vorteile, wenn keine „Variation der Kommunikationsprotokolle“ benötigt wird.
Das hier implementierte *Webinterface* kann, durch Einlesen des Modells aus der Konfigurationsdatei, eine Website erzeugen, die auf die im Modell beschriebene Anwendung zugeschnitten ist.
Auch wenn die Automatisierung eines großen öffentlichen Gebäudes oder einer Industrieanlage umgesetzt werden soll, ist es sinnvoll, alle verwendeten Aktoren und Sensoren in einem Dokument aufzulisten. Das Dokument kann zur Abstimmung zwischen der Softwareentwicklung und anderen, an der Entwicklung beteiligten, Personen verwendet werden. Wenn das Dokument gleichzeitig als Konfigurationsdatei dient und daraus Programmgrundgerüste erzeugt werden, erleichtert es die Arbeit in der Softwareentwicklung.

[10] Bei BLE können z.B. durch die in den „use-case-specific profiles“ definierten herstellerunabhängigen Schnittstellen Geräte in Funkreichweite erkannte werden.

7 Zusammenfassung und Ausblick

Es wurde gezeigt, wie es durch „modellgetriebene Softwareentwicklung“ und „generative Programmierung“ möglich ist, die Softwareentwicklung beim Erstellen von IoT-Anwendungen zu erleichtern. Dazu wurde in Form des *Datenmodells* eine zusätzliche Abstraktionsebene oberhalb der verwendeten Kommunikationsprotokolle eingeführt.
Der Softwareentwickler verwendet das Datenmodell über die vom Codegenerator bereitgestellten Programmgrundgerüste. Da es möglich ist, das Datenmodell mit unterschiedlichen Kommunikationsprotokollen zu realisieren, kann dadurch eine „Variation der Kommunikationsprotokolle“ erreicht werden.

Der hier implementiert Codegenerator stellt ein „Proof of Concept“ dar, anhand dessen die Machbarkeit des Ansatzes, sowie Vor- und Nachteile untersucht wurden. Dieser könnte erweitert werden, sodass andere Kommunikationsprotokolle unterstützt werden. Das implementierte *Gateway* zeigt, dass auch ein, in der Programmiersprache Python, geschriebenes Programm mit den anderen IoT-Geräten vernetzt werden kann. Der Codegenerator kann also auch dahingehend erweitert werden, dass er Programmgrundgerüste für unterschiedliche Geräte und in unterschiedlichen Programmiersprachen erzeugt.

Der bei allen mit dem Internet verbundenen Geräten besonders wichtige Sicherheitsaspekt, wurde in dieser Arbeit nicht untersucht. Bei der vorgestellten Implementierung könnte ein böswilliger Akteur, die Werte von Attributen anderer Knoten setzten und in deren Namen auf Methodenaufrufe antworten. Das bedeutet, allen mit dem MQTT-Broker oder BLE-Server verbundenen Geräten muss vertraut werden.
Sowohl BLE als auch MQTT besitzen die Möglichkeit, dass sich Geräte vor Aufbau der Verbindung authentifizieren müssen. Es wäre jedoch durchaus interessant zu untersuchen, wie ein Schutz vor böswilligen Akteuren erreicht werden kann, der nicht auf den darunterliegenden Kommunikationsprotokollen basiert.

Literaturverzeichnis

[1] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE communications surveys & tutorials*, 17(4):2347–2376, 2015.

[2] J Pedro Amaro, Sérgio Patrão, Fernando Moita, and Luís Roseiro. Bluetooth low energy profile for mpu9150 imu data transfers. In *2017 IEEE 5th Portuguese Meeting on Bioengineering (ENBENG)*, pages 1–4. IEEE, 2017.

[3] Arduino. Esp32 ble arduino - arduino reference. `https://www.arduino.cc/reference/en/libraries/esp32-ble-arduino/`. [Online; accessed 2022-10-07].

[4] Arduino. Pubsubclient - arduino reference. `https://www.arduino.cc/reference/en/libraries/pubsubclient/`. [Online; accessed 2022-10-07].

[5] Arduino. Wifi - arduino reference. `https://www.arduino.cc/reference/en/libraries/wifi/`. [Online; accessed 2022-10-07].

[6] Kevin Ashton. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[7] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[8] Tim Bray. The javascript object notation (json) data interchange format, 2017. doi: 10.17487. *RFC8259*, 2017.

[9] Frederick Brooks and H Kugler. *No silver bullet.* April, 1987.

[10] Robert Bryce, Thomas Shaw, and Gautam Srivastava. Mqtt-g: A publish/subscribe protocol with geolocation. In *2018 41st international conference on telecommunications and signal processing (TSP)*, pages 1–4. IEEE, 2018.

[11] Gilles Callebaut, Guus Leenders, Chesney Buyle, Stijn Crul, and Liesbet Van der Perre. Lora physical layer evaluation for point-to-point links and coverage measurements in diverse environments. *arXiv preprint arXiv:1909.08300*, 2019.

[12] Arduino CC. About arduino. `https://www.arduino.cc/en/about`. [Online; accessed 2022-09-28].

[13] Arduino CC. What is arduino? https://www.arduino.cc/en/Guide/Introduction. [Online; accessed 2022-09-28].

[14] Bokeh Contributors. Bokeh documentation. https://docs.bokeh.org/en/latest/. [Online; accessed 2022-10-02].

[15] Fredrik Dahlqvist, Mark Patel, Alexander Rajko, and Jonathan Shulman. *Growing opportunities in the Internet of Things*. McKinsey and Company, 2019.

[16] esperrif. Esp32 overview. https://www.espressif.com/en/products/socs/esp32. [Online; accessed 2022-09-28].

[17] Raspberry Pi Foundation. Raspberry 3. https://www.raspberrypi.com/products/raspberry-pi-3-model-b/. [Online; accessed 2022-09-28].

[18] Daniel Giusto, Antonio Iera, Giacomo Morabito, and Luigi Atzori. *The Internet of Things - 20th Tyrrhenian Workshop on Digital Communications*. Springer, New York, NY, 2010.

[19] Yuri F Gomes, Danilo FS Santos, Hyggo O Almeida, and Angelo Perkusich. Integrating mqtt and iso/ieee 11073 for health information sharing in the internet of things. In *2015 IEEE International Conference on Consumer Electronics (ICCE)*, pages 200–201. IEEE, 2015.

[20] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, 2003.

[21] Ecma International. Ecma-404—the json data interchange format, 2013.

[22] JSON. Einführung in json. https://www.json.org/json-de.html. [Online; accessed 2022-10-07].

[23] Rishika Mehta, Jyoti Sahni, and Kavita Khanna. Internet of things: Vision, applications and challenges. *Procedia computer science*, 132:1263–1269, 2018.

[24] Dae-Hyeok Mun, Minh Le Dinh, and Young-Woo Kwon. An assessment of internet of things protocols for resource-constrained applications. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 555–560. IEEE, 2016.

[25] Maria Rita Palattella, Ridha Soua, André Stemper, and Thomas Engel. Aggregation of mqtt topics over integrated satellite-terrestrial networks. *ACM SIGMETRICS Performance Evaluation Review*, 46(3):96–97, 2019.

[26] Francisco Presencia. Picnic css. https://picnicss.com/. [Online; accessed 2022-10-07].

[27] Pallets Projects. Flask documentation. https://flask.palletsprojects.com/en/2.2.x/. [Online; accessed 2022-10-02].

[28] Pallets Projects. Template designer documentation. https://jinja.palletsprojects.com/en/3.0.x/templates/. [Online; accessed 2022-10-02].

[29] Hwaci Applied Software Research. Sqlite home page. https://www.sqlite.org/index.html. [Online; accessed 2022-10-02].

[30] Diego RC Silva, Guilherme MB Oliveira, Ivanovitch Silva, Paolo Ferrari, and Emiliano Sisinni. Latency evaluation for mqtt and websocket protocols: an industry 4.0 perspective. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 01233–01238. IEEE, 2018.

[31] Stefan Sobernig and Uwe Zdun. Inversion-of-control layer. In *Proceedings of the 15th European Conference on Pattern Languages of Programs*, pages 1–22, 2010.

[32] Big Sky Software. Htmx. https://htmx.org/. [Online; accessed 2022-10-07].

[33] OASIS Standard. Mqtt version 5.0. *Retrieved June*, 22:2020, 2019.

[34] Kevin Townsend, Carles Cufí, Akiba, and Robert Davidson. *Getting Started with Bluetooth Low Energy*. O'Reilly Media, Inc., USA, 2014.