



Bachelorarbeit

**Entwicklung einer
Softwarelösung zur Nutzung von
Smartphones als Sensor- und
Aktor**

Marius Cerwenetz

Abschlussarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

Vorgelegt von	Marius Cerwenetz
am	08. Juli 2022
Referent	Prof. Dr. Peter Barth
Korreferent	Prof. Dr. Jens-Matthias Bohli

Schriftliche Versicherung laut Studien- und Prüfungsordnung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mannheim, 08. Juli 2022

Marius Cerwenetz

Zusammenfassung

Interaktive Programmieraufgaben eignen sich besonders gut zum Programmierenlernen für angehende Programmiererinnen oder Programmierer. Auf Mikrocontroller-Schaltungen basierende Projekte bieten durch angeschlossene Sensoren ein ideale Interaktivität, benötigen allerdings auch zusätzliches Fachwissen und müssen zusätzlich beschafft werden. Gegenüber ihnen bietet der Einsatz von Smartphones für interaktive Aufgaben Vorteile gegenüber Mikrocontroller-Schaltungen. Die Geräte bieten einen vergleichbaren Sensorumfang und können Ausgaben virtuell darstellen. Ihre Anbindung in Programmierumgebungen ist im Vergleich zu Mikrocontrollern jedoch nicht äquivalent unterstützt.

Diese Arbeit behandelt die Entwicklung einer Softwarelösung namens Smartbit, welche Sensoren von Smartphones über eine Programmierumgebung ausliest und Ausgaben auf dem Smartphone ausführt. Anforderungen an Smartbit werden anhand von Beispielaufgabenstellungen spezifiziert und für die Implementierung berücksichtigt. Die Evaluation erfolgt durch eine Demonstration einer Beispielaufgabe und einer Messung für drei unterschiedliche Nutzungsszenarien.

Smartbit kann für den beschriebenen Einsatz verwendet werden. Die Einbindung in bestehende Programmierumgebungen ist möglich. Bei Aufrufen auftretende Latenzen zwischen Smartphone und Programm beeinträchtigen nicht die Nutzung.

Inhaltsverzeichnis

1	Einführung	3
2	Smartphones als Mikrocontroller-Ersatz für Programmieraufgaben	6
2.1	Beispielprogrammieraufgaben	6
2.2	Anforderungen an Smartbit	8
3	Smartbit-Architektur und Spezifikation	11
4	Nachrichtenformate der Komponenten	16
5	Aufbau der Programmierumgebung	19
6	Aufbau und Funktionsweise der Kontrollanwendung	21
7	Aufbau und Funktionsweise der Android-Anwendung	25
7.1	Startvorgang	25
7.2	Sensoren	28
8	Evaluation der Smartbit-Lösung	31
8.1	Verwendungsbeispiel	31
8.2	Latenzmessung	35
9	Fazit	39
	Abkürzungsverzeichnis	44
	Literaturverzeichnis und Online-Quellen	45

A Nachrichtenformate**47**

Kapitel 1

Einführung

Viele angehende Programmiererinnen und Programmierer mühen sich beim Programmieren lernen zum Anfang mit der Semantik von Programmiersprachen und grundlegenden algorithmischen Konzepten. Akademische Übungsaufgaben senken durch rein virtuelle Aufgabenstellungen ohne Interaktionsmöglichkeiten die Lernmotivation. Aufgaben in Verbindung mit Mikrocontroller-Schaltungen stellen dagegen eine praktische, fordernde und spielerische Einstiegsmöglichkeit dar, um Programmieren zu lernen. In den Mikrocontroller-Projekten werden kleine Programme realisiert, die durch die Verwendung von Sensoren Programmiererinnen und Programmierer einladen, sich an den Aufgabenstellungen auszuprobieren. Die in Mikrocontroller-Schaltungen integrierten Sensoren sind Voraussetzung, um physikalische Eigenschaften in der realen Welt zu messen. Von Programmiererinnen und Programmierern entwickelte Programme können auf dem Mikrocontroller die verbundenen Sensoren auslesen und auf Änderungen der gemessenen Werte reagieren. Zusammen ermöglichen Sensoren, Mikrocontroller und ein entwickeltes Programm eine Bedienung der Schaltung. Während der Entwicklung treten jedoch häufig Probleme auf. Selten verhält sich das Programm beim ersten Versuch korrekt. Eine Anpassung des Codes ist nötig, bis das Fehlverhalten beseitigt ist. Diese kontinuierliche Weiterentwicklung mindert Ängste vor Code-Änderungen, schafft Routine in der Entwicklung und damit ein tieferes Verständnis und Hintergrundwissen für die Problemstellung. Sinnvoll sind diese Projekt-Eigenschaften insbesondere für Programmiererinnen und Programmierer mit wenig Vorwissen, wie Schüler oder Erstsemester-Studierende. Praktische Programmieraufgaben bieten für sie den größten Lerneffekt und motivieren am Meisten [2].

Mikrocontroller-Projekte benötigen allerdings teure Einstiegs-Kits. Die Kosten eines Arduino-Development-Boards belaufen sich im Arduino-Shop über 80,00 € [3]. Ein Großteil der Kosten entfällt zwar auf den konkreten Mikrocontroller, ein nicht uner-

heblicher Teil jedoch auf Peripherie wie Breadboards, Verbindungskabel und Erweiterungsboards. Die Peripherie-Anbindung setzt zusätzliches Hintergrundwissen in elektrotechnischen Bereichen voraus, zum Beispiel bei den Verschaltungskonventionen von Breadboards oder der Benutzung von Lötkolben. Fehlerhafte Verdrahtungen oder ein Fehlgebrauch resultieren in zerstörten Komponenten oder Verletzungen. Dies stellt ebenfalls eine Einstiegshürde dar, die die eigentliche interaktive Lernerfahrung hinauszögert und die Motivation senkt.

Smartphones dienen für interaktive Aufgaben als Alternative zu herkömmlichen Mikrocontroller-Schaltungen für Programmieraufgaben. Ein Vorteil ihrer Nutzung gegenüber Mikrocontrollern bei Programmierprojekten liegt in ihrer Verfügbarkeit. 2022 besaßen weltweit 5,2 Mrd. Menschen ein Smartphone [7]. Sie sind insbesondere unter Jugendlichen und jungen Erwachsenen weit verbreitet und werden im Alltag für Chats, Social-Media oder Spiele verwendet. Sie sind also häufig bereits in Gebrauch und müssten für die Nutzung von Programmierprojekten nicht zusätzlich beschafft werden. Projekte in denen Smartphones anstatt Mikrocontroller-Schaltungen verwendet werden reduzieren das Risiko von Verdrahtungsfehlern einer Schaltung dadurch, dass sie außer einem Ladeanschluss keine elektronischen Schnittstellen anbieten. Ihr Sensoren-Umfang ist mit dem von Mikrocontroller-Schaltungen vergleichbar. Zahlreiche Sensortypen wie Lagesensoren, Gyroskop oder Annäherungssensoren sind bereits integriert. Neben kabelgebundenen Übertragungsschnittstellen wie USB oder Lightning besteht auch die Möglichkeit, drahtlose Verbindungsmöglichkeiten wie WLAN oder Bluetooth zu nutzen. Die Geräte sind zudem batteriebetrieben, was Lösungen ermöglicht, die von einer Spannungsversorgung unabhängig sein können. Eine Einbindung von Smartphones in Entwicklungsumgebungen wird in den meisten Fällen jedoch nicht unterstützt. Visuelle und haptische Ausgaben auf dem Smartphone erfordern zudem eine mobile Anwendung, da Smartphone-Betriebssysteme keine nativen Ausgabemethoden bieten.

In dieser Arbeit wurde eine Softwarelösung mit dem Namen Smartbit entwickelt. Sie ermöglicht Smartphones statt Mikrocontroller-Schaltungen für die Entwicklung von interaktiven Projekten zu nutzen. Sensoren sollen ausgelesen und Ausgaben auf dem Smartphone ausgelöst werden können. Für die Verwendung von Smartbit werden angehenden Programmiererinnen und Programmierern Beispielaufgaben dargereicht, deren Bewältigung eine Bereitstellung von Benutzungsmöglichkeiten seitens Smartbit erfordert. Die Aufgaben, sowie aus ihnen ableitbare Anforderungen und Rahmenbedingungen, sind in Kapitel 2 zu aufgeführt. Die drei Komponenten Smartphone-App, Kontrollanwendung und Programmierumgebung werden in Kapitel 3 vorgestellt. Damit die Komponenten zusammenarbeiten können müssen sie Nachrichten austauschen. Die Einsatzzwecke der Nachrichtentypen und der Nachrichtenaustausch werden in Kapitel 4 erklärt beziehungsweise exemplarisch

veranschaulicht. Die Nutzung und Funktionsweise der Schnittstellen der Programmierumgebung werden in Kapitel 5 erklärt. Der Aufbau und das Verhalten, der zwischen Smartphone-App und Programmierumgebung vermittelnden Kontrollanwendung, wird in Kapitel 6 erklärt. Kapitel 7 behandelt die Funktionsweise und den Aufbau der Smartphone-App im Detail. In Kapitel 8 wird untersucht, ob die vorgegebenen Anforderungen erfüllt wurden und die Verwendung von Smartbit anhand einer Beispielaufgabe vorgestellt. Bei der Entwicklung aufgetretene Schwierigkeiten, Verbesserungs- und Erweiterungsmöglichkeiten werden in Kapitel 9 diskutiert.

Kapitel 2

Smartphones als Mikrocontroller-Ersatz für Programmieraufgaben

Smartphones sind in sich geschlossene, technische Geräte, die neben vordefinierten Verbindungsschnittstellen wie einem USB- bzw. Lightning-Port, WLAN und Bluetooth keine weiteren Schnittstellen bieten, um externe Hardware und Schaltungen anzuschließen. Gegenüber Mikrocontroller-Schaltungen bieten sie jedoch einen vergleichbaren Sensorumfang. Integrierte Lautsprecher können die in einer Mikrocontroller-Schaltung manuell angeschlossenen Piepser ersetzen. Zudem müssen Smartphones bei Ausgaben nicht auf Mehrzweck-Ausgabemöglichkeiten wie LED-Grids zurückgreifen, da zweckgebundene UI-Elemente wie Textfelder, Textausgaben oder Bildausgaben während der App-Entwicklung beliebig platziert werden können.

In diesem Kapitel werden Beispielprogrammieraufgaben vorgestellt. Diese können sowohl mit Mikrocontroller-Schaltungen, als auch mit Smartbit gelöst werden. Des Weiteren werden die aus den Beispielprogrammieraufgaben hervorgehenden Anforderungen aufgestellt und erörtert.

2.1 Beispielprogrammieraufgaben

Praxisnahe Programmieraufgaben motivieren mit interessanten Aufgabenstellungen Programmiererinnen und Programmierer. Die in diesem Abschnitt vorgestellten Beispielaufgaben definieren das von Sensormesswerten abhängige Verhalten interaktiver Programme. Die Aufgabenstellungen sind in Tabelle 2.1 aufgeführt. Für jede

Aufgabe werden die benötigten Sensortypen und Ausgabeschnittstellen beschrieben. Leserinnen und Leser werden in der Entscheidungsfindung durch die Angabe eines dreistufigen Schwierigkeitsgrades unterstützt. Dies soll verhindern, dass sich unerfahrene Programmiererinnen und Programmierer zu Anfang mit komplizierten Aufgaben überfordern.

Name der Aufgabe	Benötigte Sensoren	Verwendete Ausgabe-schnittstellen	Schwierigkeitsgrad
Disco	-	LED	Einfach
Würfeln	Lagesensor	Textfeld	Einfach
Diebstahl-Alarm	Annäherungssensor	Textfeld, LED, Vibration	Mittel
Klatsch-Zähler	Mikrofon	Textfeld	Mittel
Dreh-Zähler	Lagesensor	Textfeld	Schwer

Tabelle 2.1: Beispielprogrammieraufgaben

In der Aufgabe *Disco* soll eine virtuelle LED für eine Zeitdauer von 500 ms grün und anschließend 500 ms rot leuchten. Sensoren werden nicht verwendet, da der LED-Farbwechsel unabhängig von Sensormesswertänderungen ausgeführt wird, weshalb die Aufgabe als *Einfach* eingestuft wird.

In der Aufgabe *Würfeln* soll ein Schütteln des Geräts erkannt werden. Verwendet wird ein Beschleunigungen messender Lagesensor. Wird ein Schütteln erkannt, soll auf dem PC eine Zufallszahl generiert werden, welche anschließend auf dem Endgerät in einem Textfeld ausgegeben werden soll. Der Schwierigkeitsgrad wird ebenfalls auf *Einfach* eingeschätzt, da die Aufgabe unter Verwendung lediglich eines Sensortyps und einer Ausgabe lösbar ist.

In der Aufgabe *Diebstahl-Alarm* soll unter Verwendung des Näherungssensors das örtliche Umfeld des Geräts auf eine Annäherung überprüft werde. Im diesem Falle soll ein alarmierender Text im Textfeld ausgegeben werden. Zusätzlich soll die LED wie in Aufgabe *Disco* die Farbe wechseln. Neben den visuellen Ausgaben wird die Vibrationsfunktion als haptisches Feedback benötigt, die das Gerät fünf mal für 1000 ms vibrieren lässt. Der Schwierigkeitsgrad der Aufgabe ist als *Mittel* eingestuft, da hier zwischen Alarm- und Normalzustand unterschieden werden muss. Entfernt sich eine Person, muss der Alarm-Zustand verlassen und alle Ausgaben auf ihren Initialwert zurückgesetzt werden.

Bei der Aufgabe *Klatsch-Zähler* muss für einen selbstdefinierten Zeitraum die Anzahl der Händeklatscher gemessen werden. Diese Anzahl soll anschließend im Textfeld des Geräts ausgegeben werden. Verwendet wird das Mikrofon als Ein- und das Text-

feld als Ausgabe. Die Schwierigkeit ist auf *Mittel* angesetzt, da die Messwerte von Händeklatschern unterschiedliche Intensitäten aufweisen. Um einen Händeklatscher zu identifizieren, ist es nötig Grenzwerte zu ermitteln um sie von normalen Hintergrundgeräuschen abzugrenzen. Eine Visualisierung der Messdaten kann bei dieser Aufgabe helfen.

In der Aufgabe *Drehzähler* soll die Programmiererin oder der Programmierer das Gerät innerhalb eines selbstdefinierten Zeitraums drehen und ein Programm entwickeln, das anhand der Messdaten die Anzahl der Umdrehungen ermittelt. Diese Anzahl soll anschließend im Textfeld auf dem Gerät ausgegeben werden. Verwendet werden hier ebenfalls der Lagesensor als Sensoreingabe und das Textfeld als Ausgabe. Die Aufgabe ist als *Schwer* bewertet, da hier Wiederholungen in einer Werteabfolge erkannt werden müssen, welche jedoch wie bei der Aufgabe *Klatsch-Zähler* in ihrer Größe variieren können. Abhängigkeiten unter den Messweltergebnissen erschweren eine Erkennung der Umdrehungen zusätzlich.

2.2 Anforderungen an Smartbit

Aus der vorangegangenen Beschreibung von Beispielaufgaben lassen sich Anforderungen an Smartbit ableiten. Die Zielgruppe besteht, wie in der Einleitung beschrieben, aus IT-affinen Studieninteressierten oder angehenden Programmierern oder Programmierern, mit Hauptfokus auf sich im Grundstudium befindende Studierende der Fakultät Informationstechnik der Hochschule Mannheim. Voraussetzung der Nutzung der App ist der Besitz eines PCs und eines Android-Smartphones mit Internetzugang.

Anhand der Beispielaufgaben werden Anwendungsfälle spezifiziert, die die Nutzeranforderungen definieren. Eine Übersicht der Anwendungsfälle ist in Abbildung 2.1 dargestellt. Nutzerinnen und Nutzer möchten durch Smartbit Ausgaben auf dem

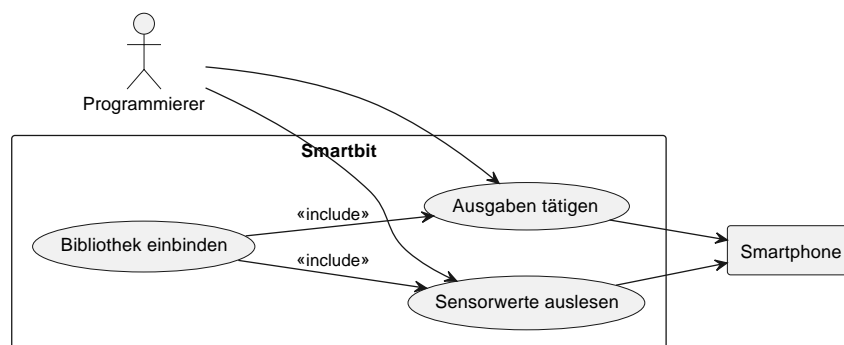


Abbildung 2.1: Anwendungsfälle für die Benutzung von Smartbit

Smartphone tätigen und Sensorwerte des Smartphones auslesen können. Ausgaben sollen visuell über ein Textfeld, eine LED und haptisch über eine Vibrationsausgabe möglich sein. Sensor-Anfragen sollen, zur Erfüllung der Aufgaben, für den Lage- und Annäherungssensor sowie dem Mikrofon möglich sein. Für die Bedienung aller Ausgabemöglichkeiten und der Abfrage der spezifizierten Sensortypen, müssen aus bestehenden Programmcode aufrufbare Funktionen existieren. Die Verwendung soll verständlich und ohne Hintergrundwissen möglich sein. Auf serialisierte Ausgabeformate ist zu verzichten. Funktionen müssen Rückgabewerte in Form von primitiven Datentypen zurückgeben. Um die Kompatibilität mit, von Programmierinnen oder Programmierern entwickelten Programmen, zu erhöhen, müssen die Funktionen in mehreren Programmiersprachen zur Verfügung stehen. Smartbit soll für Nutzerinnen und Nutzer, neben der Einbindung in Programme, auch leicht in Entwicklungsumgebungen integrierbar und mit wenig Boilerplate-Code verwendbar sein. Der Einsatz der Entwicklungsumgebung *Eclipse IDE* [8] ist an der Fakultät Informationstechnik der Hochschule Mannheim weit verbreitet und wird in den Lehrveranstaltungen des Grundstudiums umfassend genutzt. Es ist davon auszugehen, dass Studierende mit der Benutzung dieser Entwicklungsumgebung vertraut sind. Die Smartbit-Einbindung soll daher insbesondere für diese Entwicklungsumgebung unterstützt werden. Da es sich um eine plattformunabhängige IDE (Integrated Development Environment) handelt, muss Smartbit auch plattformübergreifend, also für Unix und Windows-Systeme, in diese Entwicklungsumgebung eingebunden werden können.

Um die Benutzbarkeit sicherzustellen, müssen unter der Verwendung von Smartbit Programme entwickelt werden können, die ein responsives Verhalten aufweisen. Responsivität trägt zur Lernerfahrung bei, da sich Änderungen physikalischer Umgebungseigenschaften unmittelbar auf das Verhalten des entwickelten Programms auswirken. Um eine Responsivität zu garantieren, gilt es Latenzzeiten zu minimieren. Latenzen werden durch die RTT (Round Trip Time) gemessen. Diese beschreibt die Zeitdauer der Übermittlung einer Nachricht über Hin- und Rückweg eines Hosts zu einem Anderen. Initial müssen Sensordaten vorliegen, um sie vom Smartphone zu übermitteln. Das Starten von Sensormessprozessen erfordert jedoch Zeit, was Latenzen erhöht. Anforderung an Smartbit ist es deshalb Sensordaten rechtzeitig zum Zwecke erhöhter Responsivität bereitzustellen. Smartphones bieten keine kabelgebunden Netzwerkschnittstellen, so dass drahtlose Netzwerke genutzt werden müssen. Der Versand der Sensorwerte wird durch den Transportweg zwischen Smartphone und Programmierungsumgebung und deren lokale Gegebenheiten verzögert. Latenzzeiten können je nach Mobilfunk- bzw WLAN-Standard, Umwelteinflüssen und der Anzahl der registrierten Geräte in Funkzellen variieren. Kollisions-Präventionsprinzipien wie CSMA/CA (Carrier Sense Multiple Access/Collision Avoidance) verhindern

bei WLAN-Verbindungen zwar, das Geräte gleichzeitig senden, erhöhen dadurch allerdings die Latenzen. Die Latenzen sind ebenfalls abhängig vom Standard des mobilen Netzwerks. Bei WLAN 802.11b beträgt die Latenz ca. 10 ms, kann jedoch beispielsweise bei Verbindungen mit UMTS (Universal Mobile Telecommunications System) auf 300 ms bis 400 ms ansteigen [1]. Zudem fallen die Latenzzeit bei mehreren Sendungen unterschiedlich aus, was bei einer synchronen Übertragung zu für Nutzer nicht nachvollziehbaren Einbrüchen in der Übertragungsgeschwindigkeit führt. Bei der Übertragung von Sensormessdaten ist auf den Gebrauch ihrem Einsatzzweck entsprechender Protokolle zu achten. Um die Sicherheit zu erhöhen müssen Nachrichten auf den Transportwegen verschlüsselt übertragen und eine Ausführung von Schadcode auf dem Smartphone verhindert werden. Bei der Kommunikation zwischen Smartphone-Anwendung und Programmierumgebung kann es zu Komplikationen kommen. Um die Wartbarkeit zu gewährleisten, müssen interne Abläufe transparent sein. Smartbit muss Nachrichtenabläufe nachvollziehbar darstellen, um fehlerhafte Konfigurationen zu vermeiden.

Kapitel 3

Smartbit-Architektur und Spezifikation

In diesem Kapitel werden die Smartbit-Komponenten vorgestellt und ihr Verhalten spezifiziert. Der Fokus liegt auf den Anforderungen berücksichtigenden Designentscheidungen während der Implementierung und den zur interkomponentellen Kommunikation verwendeten Protokollen. Des Weiteren wird das Verhalten der Smartbit-Lösung exemplarisch anhand des Ablaufs eines Funktionsaufrufs spezifiziert.

Smartbit besteht aus den drei Komponenten Programmierungsumgebung, Kontrollprogramm und Android-Anwendung, die zur korrekten Funktionsweise miteinander kommunizieren. Eine Übersicht des Aufbaus ist in Abbildung 3.1 dargestellt. Um mit dem Smartphone zu interagieren bietet die Programmierungsumgebung eine programmiersprachenunabhängige Bibliothek mit Schnittstellen in Form von Stub-Funktionen an, welche die API (Application Programming Interface) der Kontrollanwendung nutzen. Die Bibliothek kann in bestehenden Programmcode auf dem lokalen PC eingebunden werden, um die angebotenen Funktionen zu verwenden. Zur Verbesserung der Verständlichkeit für angehende Programmiererinnen und Programmierer, wird auf Funktionen mit asynchronen Rückgabewerten, wie beispielsweise Futures, verzichtet. Die in der Programmiersprache Python implementierte Kontrollanwendung, welche ebenfalls auf dem lokalen PC betrieben wird, dient der Nachrichtenvermittlung zwischen Smartphone-App und Programmierungsumgebung. Latenzen zwischen Smartphone und Programmierungsumgebung werden durch den Einsatz eines Zwischenspeichers für Sensorwerte vor der Programmiererin oder dem Programmierer verschleiert. Das Problem variierender, durch die Verwendung von Funknetzen verursachter, Übertragungsgeschwindigkeiten wird dadurch umgangen. Das Verwalten des Zwischenspeichers ist ebenfalls Aufgabe der Kontrollanwendung.

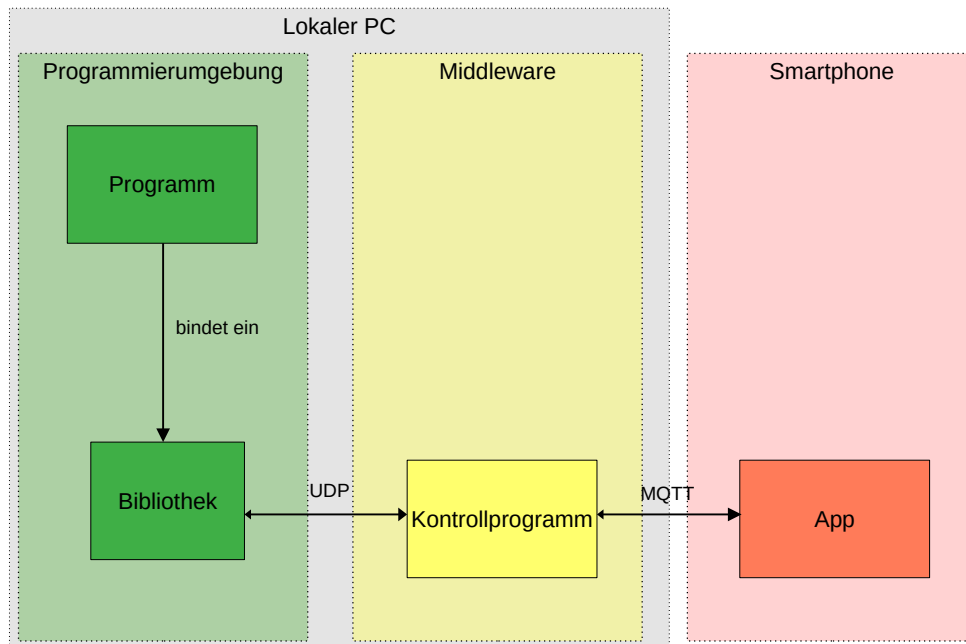


Abbildung 3.1: System-Aufbau

Ausgabemöglichkeiten werden von der, für Android-Smartphones implementierten, mobilen Anwendung angeboten. Diese reagiert auf, von Funktionen der Bibliothek gesendeten, Ausgabe-Anweisungen und sendet kontinuierlich Sensorwerte zur Zwischenspeicherung an die Kontrollanwendung. Um Latenzzeiten beim Start der Sensormessprozesse zu reduzieren, werden diese bereits zum Start der Smartphone-App ausgeführt, so dass die Kontrollanwendung bereits vor der Ausführung des Programms der Programmiererin oder des Programmierers auf Sensorwerte zwischengespeichert hat. Die Komponenten stehen untereinander in einer 1:1-Beziehung, Programmiererinnen und Programmierer können mit *einem* Programm und *einer* Kontrollanwendung *ein* Smartphone ansteuern. Für das Ansteuern mehrerer Smartphones wird jeweils eine Kontrollanwendung benötigt. Umgebungsparameter für die Komponenten können in den zwei Konfigurationsdateien *config.json* und *protocol.json* eingestellt werden. In Ersterer sind grundlegende Konfigurationsparameter wie Hostname und Port des MQTT-Brokers oder der Name des Topics definiert. In der zweiten Datei werden neben den Schlüsselwörtern und Abkürzungen für Sensortypen und Ausgabekommandos auch Vorlagen für Nachrichtenformate aufgeführt.

Zur Übermittlung der Nachrichten werden die Protokolle MQTT (Message Queuing Telemetry Transport) und UDP (User Datagram Protocol) eingesetzt. MQTT ist ein

auf dem Observer-Pattern basierendes Client-Server-Protokoll. Durch einen 2 Byte großen Header und eine maximale Payload-Größe von 260 MB [5] ist es leichtgewichtig und gleichzeitig flexibel. Nachrichten werden zur Publizierung auf einem Topic an einen MQTT Broker gesendet. Dieser leitet sie dann an alle Topic-Abonnenten weiter. Die Verwendung von MQTT ist vom Referenten vorausgesetzt. Um die Latenzzeiten zwischen Programmierumgebung und Kontrollprogramm zu reduzieren, wird die UDP-Kommunikation zwischen Bibliothek und Kontrollprogramm auf dem lokalen PC über ein Loopback-Interface umgesetzt. Das Loopback-Interface ist eine virtuelle Netzwerk-Schnittstelle des Betriebssystems eines PCs. Pakete werden nicht über externe Netzwerk-Schnittstellen wie Netzwerkkarten versendet, sondern verbleiben im Netzwerk-Stack des Betriebssystems. Latenzzeiten beim Versand betragen dadurch weniger als eine Millisekunde. Zur weiteren Minimierung der Latenzen wird UDP als verbindungsloses Protokoll eingesetzt. Die Verwendung verbindungsorientierter Protokolle wie TCP würde neben ihrer Übertragungssicherheit, durch eine insgesamt größere Anzahl an zu sendenden Nachrichten, auch die Latenzzeiten erhöhen. Da die Reduzierung der Latenzen, zum Zweck einer besseren Lernerfahrung, eine höhere Priorität hat als die Gewährleistung der Übertragungssicherheit wird auf verbindungsorientierte Protokolle verzichtet. Als Nachrichtenformat wird JSON (JavaScript Object Notation) verwendet. Das menschenlesbare, kompakte Nachrichtenformat ist weit verbreitet und wird von vielen Softwarebibliotheken unterstützt. Es ermöglicht die Darstellung von Zahlen, Strings, Booleans und Arrays und die Schachtelung von Objekten als Schlüssel-Wert-Paare [15]. Als Character-Encoding wird UTF-8 verwendet, wodurch beispielsweise auch eine Textausgabe von Emojis auf dem Smartphone ermöglicht wird.

Die beschriebenen Komponenten bilden ein, über Nachrichten kommunizierendes, heterogenes verteiltes System. Um das von den Nachrichten abhängige Verhalten der Komponenten zu spezifizieren, wird im Folgenden der Ablauf einer Beispielnachricht vorgestellt. In Abbildung 3.2 ist eine schematische Darstellung der Funktionsweise einer Ausgabe-Funktion abgebildet. Dies könnte beispielsweise eine Funktion sein, die einen Text auf dem Textfeld der Smartphone-App anzeigen soll. Die Bibliothek mit den, zur Interaktion mit dem Smartphone geeigneten, Stub-Funktionen kann in Programmcode der Programmiersprachen C, Java und Python eingebunden werden und ermöglicht dadurch eine Entwicklung eines mit dem Smartphone interagierenden Programms(1). Führt eine Programmiererin oder ein Programmierer einen Funktionsaufruf der Bibliothek aus (2), übermittelt die Bibliothek eine Anfrage per MQTT an das Kontrollprogramm (3). Dieses entscheidet über die weitere Verfahrensweise. Anfragen, die Sensormessdaten des Smartphones fordern, werden direkt bedient. Dieser Weg ist in der Abbildung nicht dargestellt. Ausgabe-Anfragen werden per MQTT an das Smartphone übertragen (4). Die Smartphone-App nimmt die Anfrage

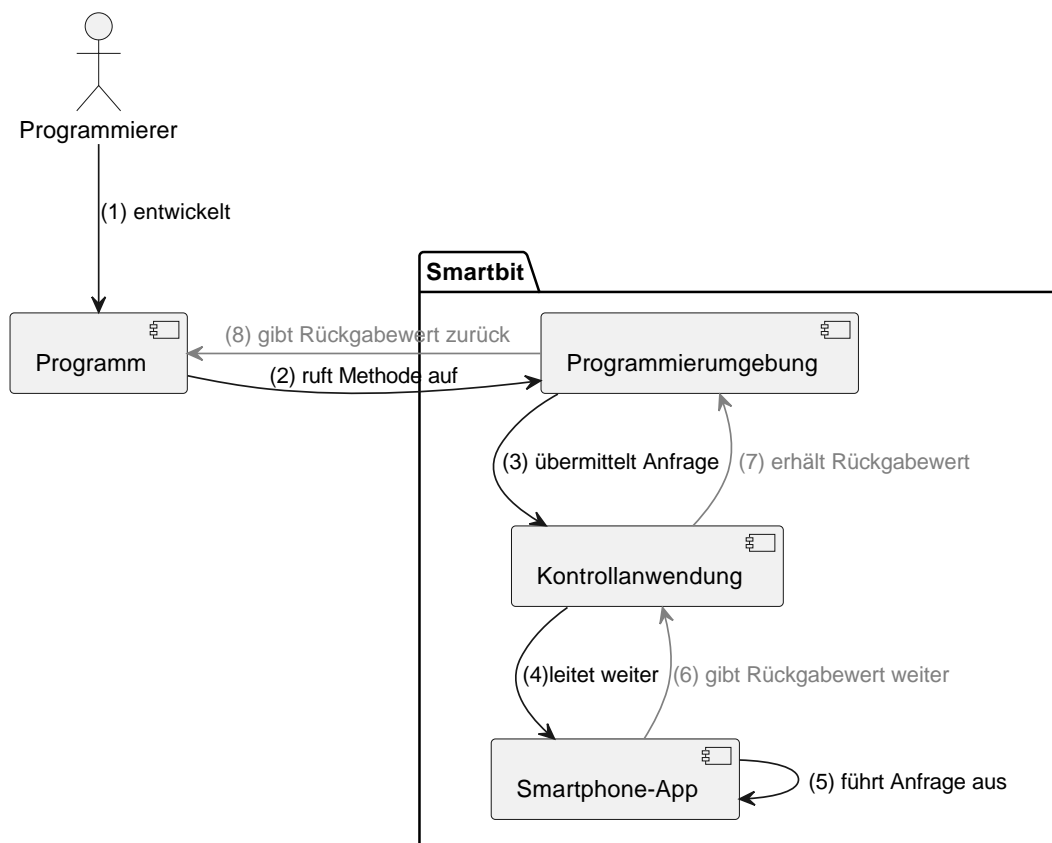


Abbildung 3.2: Nachrichtenablauf einer Ausgabe

an und führt sie aus (5). Der Ablauf von Ausgaben, die Rückgabewerte zurückgeben, ist in der Abbildung grau dargestellt. In diesem Fall sendet die App den Rückgabewert an die Kontrollanwendung (6), welche ihn an die Bibliothek weiterleitet (7). Die Bibliothek gibt den Rückgabewert zum Schluss an das aufrufende Programm zurück (8)

Kapitel 4

Nachrichtenformate der Komponenten

Ein einheitliches Nachrichtenformat ist Voraussetzung für den Nachrichtenaustausch. Der ausgearbeitete Nachrichtenstandard definiert die Nachrichten in einem Klartextformat. Nachrichten-Vorlagen sind in einer Datei gespeichert und werden in der Bibliothek, dem Kontrollprogramm und in der Smartphone-App eingelesen, wodurch sie in allen Komponenten kongruent vorliegen. Es gibt unterschiedliche Nachrichtentypen, welche in Tabelle 4.1 aufgeführt sind. Zur besseren Nachvollziehbarkeit der Kommunikation sind zusätzlich Quelle, Ziel und das verwendete Netzwerkprotokoll angegeben. Eine Liste der vollständigen Nachrichtentypen, inklusive ihrer Felder, ist im Anhang aufgeführt.

Nachrichtentyp	Quelle	Ziel	Netzwerkprotokoll
sensor_request	Bibliothek	Kontrollprogramm	UDP
sensor_response	Kontrollprogramm	Bibliothek	UDP
update_request	Smartphone	Kontrollprogramm	MQTT
rpc_request	Bibliothek, Kontrollprogramm	Smartphone	UDP/MQTT
rpc_response	Smartphone, Kontrollprogramm	Bibliothek	UDP/MQTT

Tabelle 4.1: Nachrichten-Typen

`sensor_requests` kommen bei Sensormesswert-Abfragen zum Einsatz. Dieser Nachrichtentyp wird von der Bibliothek an das Kontrollprogramm gesendet, in der die vom Smartphone übermittelten Sensorwerte zwischengespeichert wurden. Nach Eingang ermittelt das Kontrollprogramm, durch Angabe des gewünschten Sensortyp-Kürzels im Feld `sensor_type`, den gespeicherten Sensormesswert. Sensortyp-Kürzel

sind für alle Sensoren definiert und dienen in der Kontrollanwendung als Schlüssel der Adressierung der Messwerte im Zwischenspeicher der Kontrollanwendung. Die Kürzel sind in Tabelle 4.2 aufgelistet. Ist für den Sensortyp ein Sensormesswert

TYPE-Kürzel	Beschreibung
accel_{x,y,z}	Lagesensor für die X, Y oder Z-Richtung
gyro_{x,y,z}	Gyroskop-Sensor für die X, Y oder Z-Richtung
prox	Näherungssensor

Tabelle 4.2: Sensor-Kürzel mit Beschreibung

im Zwischenspeicher vorhanden, wird das Ergebnis in einer `sensor_response`-Nachricht, im Feld `sensor_value`, zurückgesendet. Die Antwort wird von der Bibliothek angenommen und an die aufrufende Funktion zurückgegeben. Um stets aktuelle Sensormesswerte im Zwischenspeicher zu erhalten, müssen diese von der Smartphone-App in regelmäßigen Zeitabständen übermittelt werden. Diese sendet daher in periodischen Abständen Nachrichten des Typs `update_request` an das Kontrollprogramm. Übermittelt werden zur Einspeicherung und Zuordnung im Zwischenspeicher sowohl der Sensortyp als auch der Sensormesswert. Der gesamte Ablauf der Sensordatenübertragung ist in Abbildung 4.1 dargestellt. Aufgeführt sind

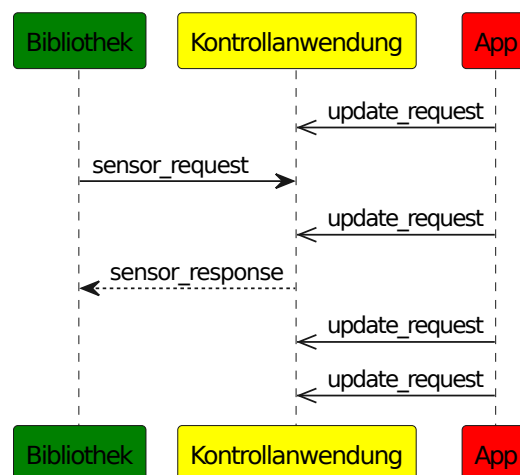


Abbildung 4.1: Nachrichtenablauf der Sensordatenübermittlung

die drei Komponenten Bibliothek, Kontrollprogramm und Smartphone-App. Die Bibliothek sendet `sensor_requests` per UDP an das Kontrollprogramm. Dieses antwortet über UDP mit einer `sensor_response`. Währenddessen werden der Kontrollanwendung von der Smartphone-App fortwährend neue Sensormesswerte durch `update_requests` übertragen.

Neben Nachrichten die die Sensordatenübermittlung betreffen, existieren zur Umsetzung von Ausgaben auf der Smartphone-App auch Ausgabe-Nachrichten. Die

app-seitigen Ausgaben sind unterscheidbar in Ausgaben mit und ohne Rückgabewert. Für erstere gibt es den Nachrichtentyp `rpc_request`.

RPC (Remote Procedure Call) bezeichnet clientseitige Funktionsaufrufe, die auf einem Server ausgeführt werden. Die Bezeichnung entspricht nicht exakt dem Konzept, da ein Kommunikationspartner, Smartphone-App oder Kontrollanwendung, in diesem Fall die Rolle des Servers einnehmen würde. Die Voraussetzung einer Client-Server-Anwendung ist durch die beidseitige Nachrichtenübertragung zwischen Smartphone-App und Kontrollanwendung nicht gegeben. Die Bezeichnung wurde unter dem Fokus auf der entfernten Ausführung einer Funktion gewählt und entspricht eher einer RMI (Remote Method Invocation).

Der Nachrichtentyp `rpc_request` enthält die Felder `command` und `value`. Diese spezifizieren die Ausgabe und falls vorgesehen eine mit der Ausgabe verbundener Parameter. Für die Vibrationsausgabe wären die Inhalte des `rpc_requests` für eine Vibration von einer Sekunde beispielsweise `vibrate` und `1000`. Die Nachricht wird von der Bibliothek per UDP an die Kontrollanwendung und von dort aus per MQTT an das Smartphone gesendet. Die Smartphone-App nimmt die Anfrage an und führt die Ausgabe aus. Manche Ausgabe-Anweisungen geben zusätzlich einen Rückgabewert zurück. Damit dieser vom Smartphone zurück an die Bibliothek gesendet werden kann, gibt es das Nachrichtenformat `rpc_response`. Eine Nachricht dieser Nachrichtenart wird erst per MQTT an das Kontrollprogramm und von dort aus per UDP an die Bibliothek gesendet. Der Nachrichtenablauf wird in Abbildung 4.2 dargestellt. Aufgeführt sind die drei Komponenten Bibliothek, Kontrollprogramm

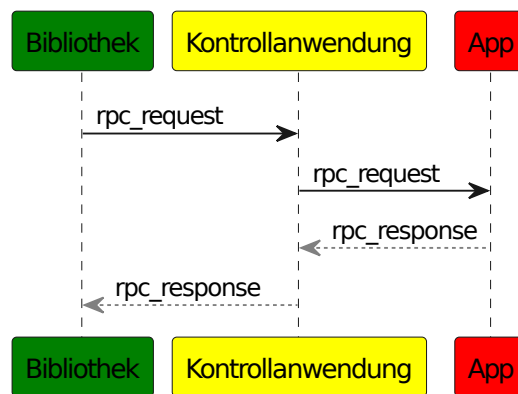


Abbildung 4.2: Nachrichtenablauf der RPC-Anfragen

und Android-App. Die Bibliothek sendet ein `rpc_request` per UDP an das Kontrollprogramm. Dieses leitet die Nachricht per MQTT weiter an die Smartphone-App, wo das Kommando ausgeführt wird. Ein eventueller Rückgabewert wird durch eine `rpc_response` vom Smartphone per MQTT zurück an die Kontrollanwendung gesendet, welche die Nachricht dann per UDP weiter an die Bibliothek weiterleitet.

Kapitel 5

Aufbau der Programmierumgebung

Die Programmierumgebung ist die Schnittstelle, die Programmiererinnen und Programmierer für die Interaktion mit dem Smartphone in ihren Programmen verwenden. Sie besteht aus einer Bibliothek, die Funktionen anbietet, mit denen Sensormesswerte eines Smartphones eingelesen, oder Ausgaben auf dem Smartphone getätigt werden können. In diesem Kapitel wird der Aufbau der Bibliothek und der Ablauf der angebotenen Funktionen beschrieben.

Die Bibliothek existiert, zur Kompatibilität, für Programme der Programmiersprachen C, Java und Python. In Programmen in Java und Python ist sie zudem plattformunabhängig. Für C-Programme gibt es zwei Bibliotheken: Eine für Unix- und eine für Windows-Systeme. In C ist die Bibliothek prozedural mit statischen Methoden, in Java und Python objektorientiert implementiert. Der TIOBE-Index stellt, anhand der Häufigkeit von Suchbegriffen der 25 wichtigsten Suchmaschinen im Internet, ein Popularitätsranking für Programmiersprachen dar. Die Sprachen C, Java und Python belegen im Juni 2022 die ersten drei Plätze [4].

Beim Aufruf der bereitgestellten Funktionen, werden standardisierte Anfragen im JSON-Format generiert und an die Kontrollanwendung gesendet. Diese sendet die Daten gegebenenfalls an das Smartphone weiter oder antwortet direkt. Eine Übersicht des System-Kontexts der Bibliothek ist in Abbildung 5.1 dargestellt.

Alle Anfragen werden über das UDP-Protokoll unter IPv4 versendet. Das Kontrollprogramm ist auf dem Port 5006 erreichbar, Bibliotheken auf dem Port 5005. Beide kommunizieren über die localhost-Adresse 127.0.0.1. Dadurch werden Datagramme über das Loopback-Interface gesendet. Zum Senden und Empfangen von Anfragen werden Sockets verwendet. Für das Empfangen von Paketen müssen diese gebunden werden, für Sendevorgänge nicht.

Bei der Erstellung eines Phone-Objekts in Python und Java werden die Nachrich-

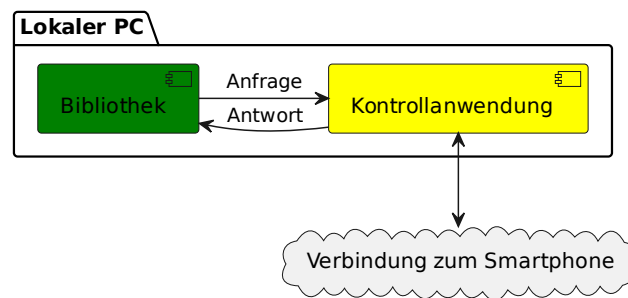


Abbildung 5.1: System-Kontext der Bibliothek

tenvorlagen für Anfragen und Antworten aus der Datei `protocol.json` geladen. Die Datei muss sich im Dateisystem im gleichen Ordner befinden wie die Bibliothek. Für die C-Bibliothek sind alle Methoden statisch definiert. Es gibt somit keinen Startpunkt, zu dem die Datei `protocol.json` eingelesen werden kann. Damit die Datei nicht für jeden Funktionsaufruf kontinuierlich eingelesen werden muss, muss ihr Inhalt von der Programmiererin oder dem Programmierer einmal zum Start des Programms als C-String eingelesen werden. Anschließend muss dieser C-String für jeden Aufruf einer Funktion der Bibliothek als Parameter angegeben werden. Die Methode `get_file_content` kann, unter der Angabe des Dateipfades der `protocol.json`-Datei, aufgerufen werden, um den Dateinhalt einzulesen und den Inhalt als C-String zurück zu erhalten. Diese Lösung verringert die Anzahl der Lesevorgänge und damit die Latenzzeiten. Nach aufruf der Einlese-Funktion ist der Datei-Inhalt auf dem Heap des Arbeitsspeichers gespeichert. Programmiererinnen und Programmierer müssen diesen am Ende des Programms durch den Aufruf der `free`-Funktion wieder freigeben.

Für die Verwendung unter C- und Java-Programmen muss zusätzlich ein JSON-Parser eingebunden werden, um die Anfragen an die Kontrollanwendung versenden und Antworten von ihr parsen zu können. Unter C wird die unter OpenSource-Lizenz stehende Bibliothek `cJSON` von Dave Gamble [9] verwendet. Diese kann manuell kompiliert und gelinkt werden. Unter Debian steht mit `libcjson1` [16] allerdings auch ein installierbares Paket zur Installation bereit. Um ein Programm erfolgreich zu kompilieren, muss gcc mit der `-I`-Option der Pfad zum `cJSON`-Header angegeben werden. Dieser ist bei der Installation des Paketes unter `/usr/lib/cjson` zu finden. Um die Dateien zu linkern, muss gcc mit der `-l`-Option der Name der Library (`cjson`) mitgeteilt werden. Da Java ebenfalls keinen integrierten JSON-Parser besitzt, wird daher `JSON in Java` [12] von Sean Leary verwendet. Dieser kann als JAR-Datei heruntergeladen und Projekten im Classpath angegeben werden. Eine Integration über Gradle oder Maven ist jedoch auch möglich [13].

Kapitel 6

Aufbau und Funktionsweise der Kontrollanwendung

Die Kontrollanwendung vermittelt als Middleware zwischen Anfragen der Programmierungsumgebung und Anfragen der mobilen Anwendung. Durch sie können Ausgaben in der Android-App ausgeführt und Sensordaten abgefragt werden. In diesem Kapitel wird der Aufbau der in Python entwickelten Anwendung anhand ihrer Komponenten dargestellt und die Funktionsweise der Bestandteile im Detail an exemplarisch erläutert.

Die Kontrollanwendung besteht aus mehreren Thread-Klassen, welche beim Start der Kontrollanwendung gestartet werden. Sie ist mit den Klassen DataHandler, MQTT-HandlerThread und SensorDB aufgeteilt in einen Daten verwaltenden Bestandteil, eine MQTT-Anbindung und einem Zwischenspeicher für Sensormesswerte. SensorDB bietet einen threadsicheren Zugriff auf ein intern referenziertes Python-Dictionary. Eine Übersicht über die Komponenten der Kontrollanwendung ist in Abbildung 6.1 dargestellt.

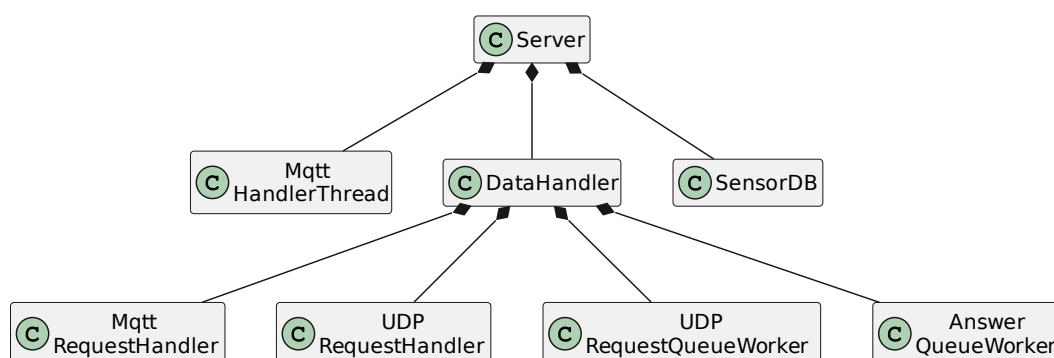


Abbildung 6.1: Übersicht der Komponenten der Kontrollanwendung

Die Komponente DataHandler besteht wiederum aus vier Sub-Komponenten, die ebenfalls als Threads nebenläufig gestartet werden: MqttRequestHandler, UDPRequestHandler, UDPRequestQueueWorker und AnswerQueueWorker. Ziel der Aufteilung, der Arbeitsschritte in Threads, ist die Gewährleistung der Verfügbarkeit der Kontrollanwendung sowohl für die Programmierungsumgebung, als auch die Smartphone-App. Die Thread-Kommunikation wird durch die threadsichere Warteschlangen-Klasse *queues* [14] der C-Python Implementierung realisiert. Nachrichten werden von einem Thread in einer queue abgelegt und in einem anderen aus der queue entnommen. Eine, durch polling verursachte, erhöhte CPU-Last kann durch blockierende Operationen beim Ablegen und Herausnehmen der Nachrichten abgewendet werden. Die Funktionsweise und Zwecke der Subkomponenten von DataHandler werden im Folgenden anhand zweier Beispiele erläutert.

Abbildung 6.2 wird zur Erläuterung für das erste Beispiel diskutiert. Dargestellt

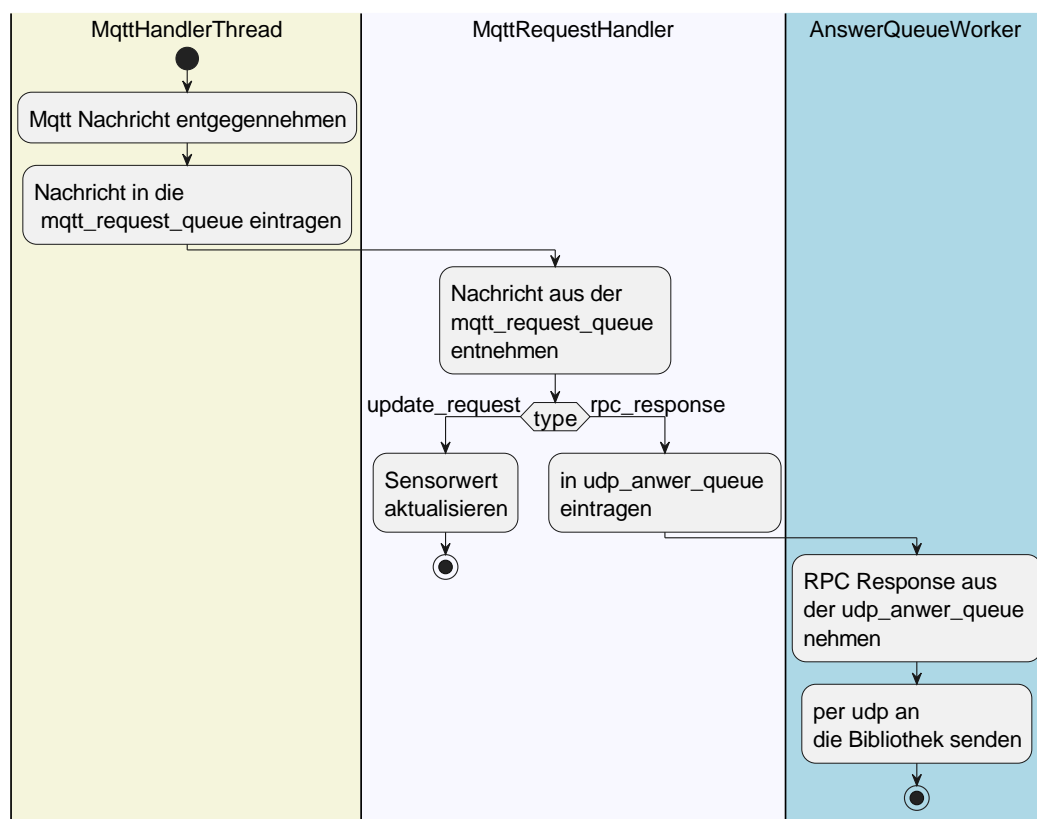


Abbildung 6.2: Ablaufdiagramm MQTT Request

ist der Ablaufplan bei Empfang einer von der mobilen Anwendung an die Kontrollanwendung per MQTT versandten Nachricht. Dies könnte beispielsweise ein *update_request* sein. Für die MQTT-Verbindung wird die, unter OpenSource-Lizenz stehende, MQTT-Library Paho [6] der Eclipse-Foundation verwendet. Erreicht

eine Nachricht per MQTT die Anwendung, wird sie vom `MQTTHandlerThread` entgegengenommen. Dieser hat sich mit einem, unter der Hostadresse `pma.inftech.hs-mannheim.de` MQTT-Broker, auf ein, in der Datei `config.json` angegebenes, Topic verbunden und reagiert mit einer Callback-Methode auf eingehende Nachrichten. Erreicht eine Nachricht den `MQTTHandlerThread`, wird die Callback-Funktion ausgelöst und die Payload der Nachricht in eine Queue eingetragen. Der `MqttRequestHandler` wartet, bis ein Eintrag in der Queue vorhanden ist, entnimmt gegebenenfalls eine Nachricht und bestimmt den Nachrichtentyp der Anfrage. Prinzipiell können von der Smartphone-App nur zwei Nachrichtentypen an die Kontrollanwendung versandt werden. Handelt sich um ein `update_request`, also einen neuen Sensorwert, muss dieser in der Datenbank aktualisiert werden. Handelt es sich hingegen um eine `rpc_response`, also um eine Antwort auf eine vorausgegangenes `rpc_request`, muss diese Nachricht per UDP an die Bibliothek zurückgesendet werden, wofür sie in eine andere queue abgelegt wird. Der `AnswerQueueWorker` wartet, bis der `MqttRequestHandler` die Nachricht abgelegt hat und sendet diese dann per UDP über die localhost-Adresse `127.0.0.1` auf dem Port `5005` zurück an die Bibliothek.

Das zweite Beispiel befasst sich mit dem Ablauf eines UDP-Requests, also einer mit UDP versandten Nachricht von der Programmierung. Der Ablauf ist in Abbildung 6.3 dargestellt. Erreicht eine Nachricht per UDP die Anwendung, wird sie vom `UDPRequestQueueWorker` entgegengenommen. Dieser bindet beim Start einen Socket für die localhost-Adresse `127.0.0.1` und den Port `5005` ein. Geht auf dem Socket eine Nachricht ein, wird sie in eine queue abgelegt. Der `UDPRequestHandler` entnimmt die Nachricht aus der queue und bestimmt den Typ der Anfrage. Handelt es sich um eine Nachricht des Typs `rpc_request`, also einer Ausgabe-Anfrage für das Smartphone, muss sie per MQTT an dieses versandt werden, wofür sie in eine queue eingetragen wird. Der `MQTTHandlerThread` entnimmt die Nachricht und sendet sie per MQTT ab. Ist Nachricht hingegen ein `sensor_request`, also die Anfrage eines Sensormesswerts, muss eine Nachricht des Typs `sensor_response` erstellt und der ausgelesene Wert eingefügt werden. Dieser wird unter Verwendung der `SensorDB`-Klasse ermittelt. Die erstellte, befüllte Antwort muss anschließend an die Programmierung zurückgesendet werden. Hierfür wird die Anfrage vom `UDPRequestHandler` in eine Queue eingetragen, aus der der `AnswerQueueWorker` sie wieder entnimmt und sie an die Programmierung zurücksendet.

Zusammenfassend erfüllen die Komponenten folgende Aufgaben. Der `MQTTHandlerThread` nimmt Nachrichten per MQTT an und sendet sie ab. Das Gegenstück dazu bilden für UDP-Anfragen der `UDPRequestQueueWorker`, der Anfragen von der Programmierung annimmt und der `AnswerQueueWorker`, der Anfragen an diese zurücksendet. Aufgabe des `MqttRequestHandler` ist die Bearbeitung von MQTT Requests.

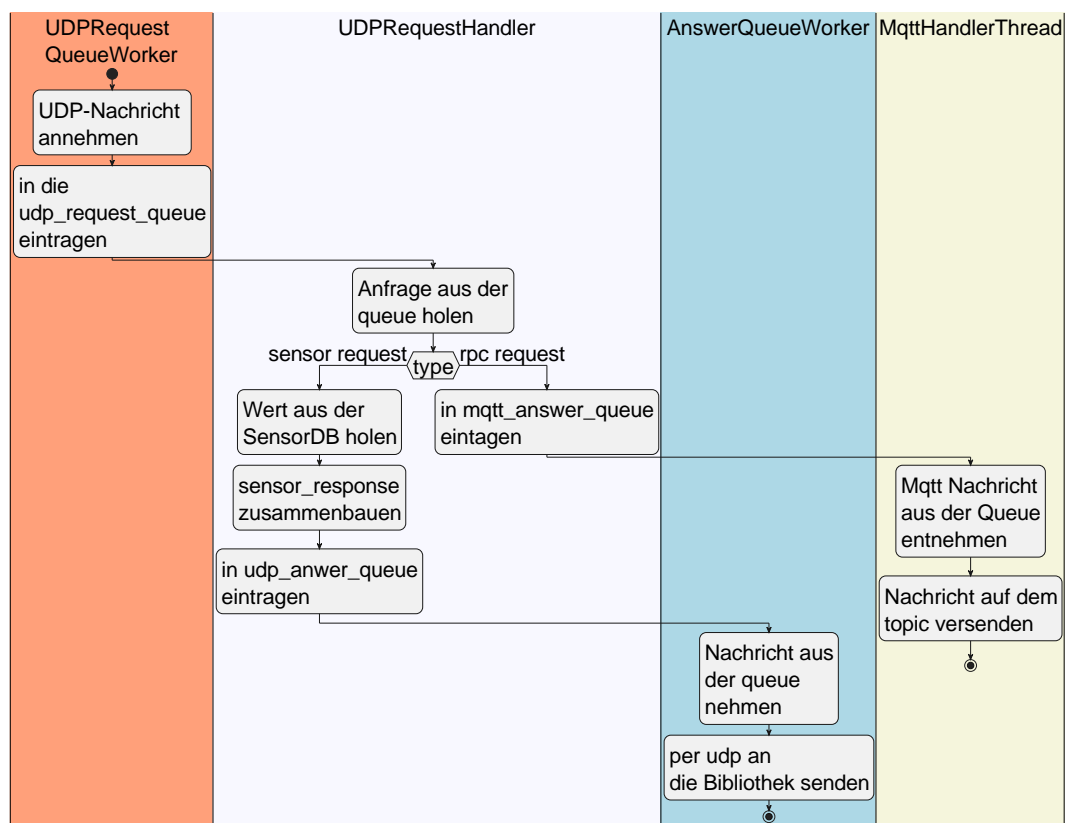


Abbildung 6.3: Ablaufdiagramm UDP Request

Kapitel 7

Aufbau und Funktionsweise der Android-Anwendung

Die Android-Anwendung ist eine der drei Bestandteile des Frameworks. Sie dient dazu Sensormessprozesse zu starten, Sensordaten zu übermitteln und Ausgabe-Kommandos auszuführen. Der Aufbau und die internen Abläufe der App, in Bezug auf Nachrichtenempfang und automatisierter Übertragung der Sensormesswerte, werden in diesem Kapitel im Detail beschrieben.

Als visuelle Ausgabemöglichkeiten bietet die Android-Anwendung eine Signal-LED, ein Textfeld und zwei Buttons in einer RootActivity an. Neben UI-Elementen gibt es zusätzlich noch eine Vibrationsausgabe. Die Anwendung initiiert zum Start Sensormessprozesse und sendet unter Verwendung von `SensorEventListener` die Sensormessdaten in periodischen Zeitabständen an die Kontrollanwendung. Zur Verwendung kommende Sensortypen umfassen beispielsweise den Lagesensor oder das Gyroskop des Smartphones. Die gemessenen Werte werden über einen im Hintergrund ausgeführten MQTT-Service versandt, welcher ebenfalls auf eingehende Nachrichten reagiert, um Ausgaben auf dem Smartphone auszulösen.

7.1 Startvorgang

Ein Ablaufplan des Startvorgangs ist in Abbildung 7.1 dargestellt. In der RootActivity werden anfänglich alle UI-Elemente zur programmiertechnischen Ansteuerung eingebunden. Anschließend werden die zwei Konfigurationsdaten *config.json* und *protocol.json* eingelesen. Nach dem Einlesen der Konfigurationen wird der zur Kommunikation verwendete MQTT-Service gestartet und asynchron eingebunden. Über eine `ServiceConnection` wird beim erfolgreichen Einbinden über eine Callback-Methode

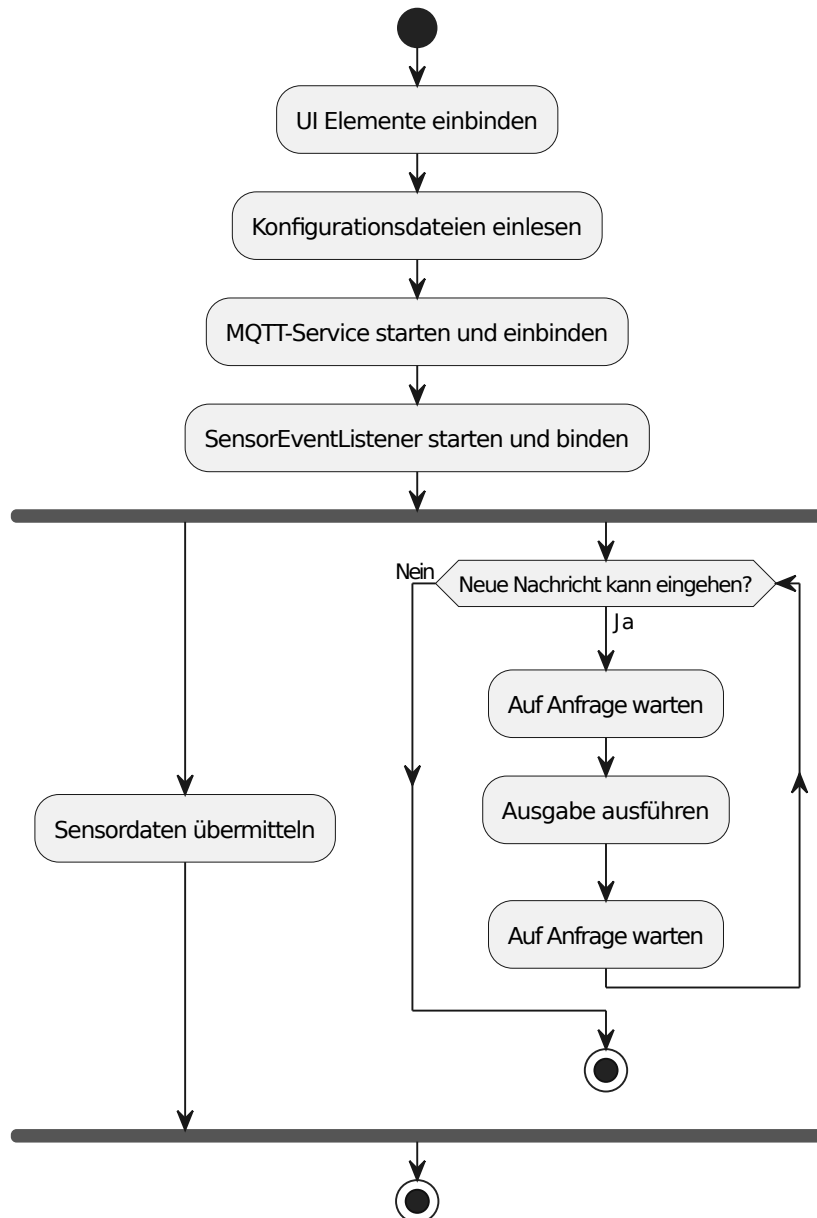


Abbildung 7.1: Ablaufdiagramm Android-Anwendung

der weitere Ablauf der Anwendung definiert, der erst ausgeführt werden soll, wenn eine MQTT-Verbindung hergestellt ist. RootActivity und MQTT-Service tauschen jeweils ihre Objekt-Referenzen aus, da Methoden des MQTT-Service um Ausgaben zu tätigen, Zugriff auf Methoden der RootActivity benötigen. Die in der RootActivity referenzierten SensorEventListener benötigen wiederum eine Referenz des MQTT-Services, da sie Sensormesswerte über diesen `update_requests` versenden.

Der Nachrichtenempfang im MQTT-Service wird nicht im UI-Thread behandelt. Dies ist jedoch Voraussetzung um UI-Elemente, wie zum Beispiel die Farbe der Signal-LED, im Service verändern zu können. Android unterbindet Änderungen des UIs aus Nicht-UI-Threads. Dieses Problem wird durch die Methode `runOnUiThread` umgangen, welche die UI-Änderungen in der Ausführungswarteschlange des UI-Threads einreicht. Eine Übersicht des Vorgangs in Abbildung 7.2 dargestellt. Der MQTT-Service

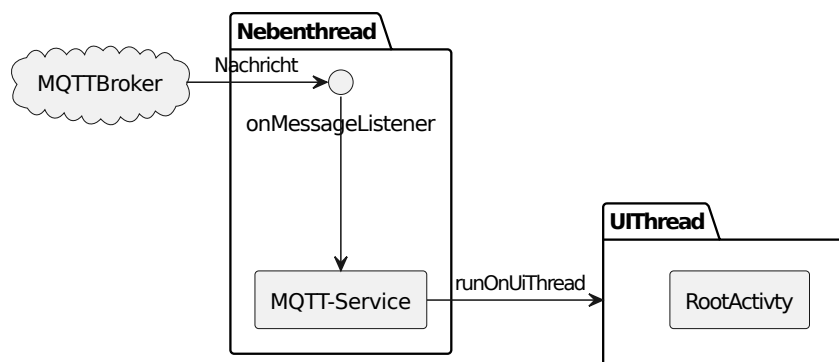


Abbildung 7.2: Ausführung auf dem UI-Thread

baut eine Verbindung zu, einem in der Datei `config.json` definierten, MQTT-Broker auf und abonniert ein, ebenfalls in der Konfigurationsdatei angegebenes, Topic. Ist der MQTT-Service eingebunden sind alle Voraussetzungen für eine Übertragung erfüllt und Sensormessprozesse können gestartet werden.

Die Funktionsweise der Sensormessung wird in Abbildung 7.3 dargestellt. SensorEventListener werden, nach Prüfung der Verfügbarkeit der jeweiligen Sensoren, gestartet und zentral in der Instanz der Klasse `SmartBitEventListenerContainer` gespeichert. Diese Klasse beinhaltet SensorEventListener für alle unterstützten Sensoren und dient ihrer Datenhaltung. Aufgabe der SensorEventListener ist es, auf Sensorwert-Änderungen zu reagieren und gegebenenfalls die Callback-Funktion `onSensorChanged` auszuführen, in der die Sensorwerte über den MQTT-Service per `update_requests` an die Kontrollanwendung übermittelt werden. Statische Methoden der Klasse `JSONMessageWrapper` ermöglichen die Erstellung der Nachrichtenformate und setzen das Messergebnis in das entsprechende Feld in der JSON-Nachricht ein. Die so generierte Nachricht wird anschließend über den gebundenen

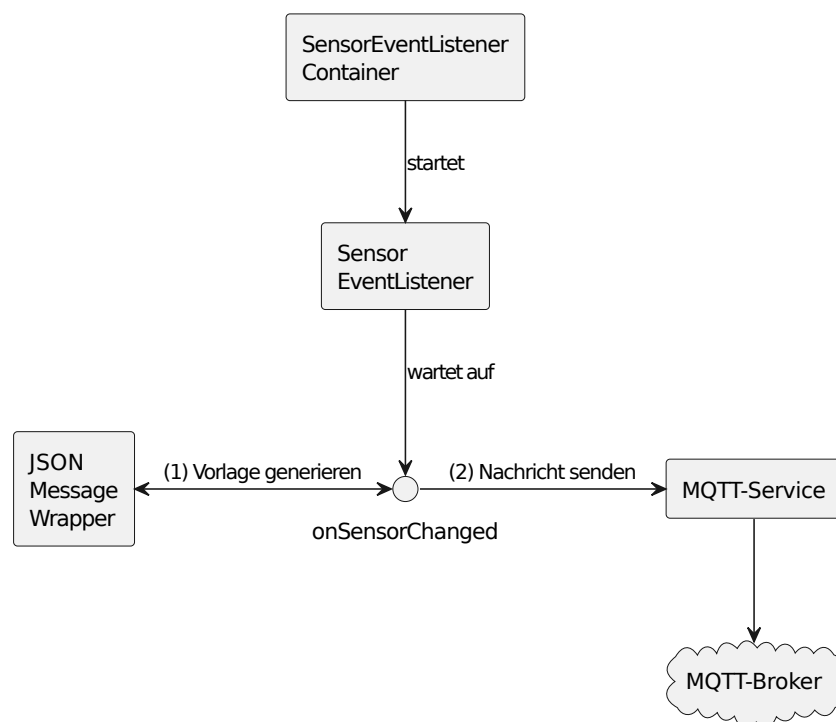


Abbildung 7.3: Ablauf SensorEventListener

MQTT-Service an das vorher definierte Topic versandt. Ist der Erhalt von Nachrichten möglich und werden Sensormessdaten automatisch an die Kontrollanwendung gesendet, ist die Startroutine der mobilen Anwendung abgeschlossen. Auf Nachrichten wird nun nur noch im MQTT-Service in einem `MessageListener` mit entsprechendem Callback reagiert und gegebenenfalls eine Ausgabe ausgeführt. Die Anwendung ist nun betriebsbereit und sendet `update_requests` an die Kontrollanwendung. Übermittelt werden die Sensordaten an den Broker mit einer QoS (Quality of Service) von 0. Auf dieser Stufe wird der Empfang der Nachricht von den Kommunikationspartnern nicht bestätigt. Verluste von `update_requests` sind unproblematisch, da je nach Taktung der `SensorEventListener` innerhalb kurzer Zeit neue Sensormesswerte zur Übertragung vorliegen. Eine exakte Zustellung ist hier nicht notwendig. Höhere QoS-Stufen würden den Übertragungsprozess verlangsamen und Latenzen erhöhen.

7.2 Sensoren

Smartphones beinhalten Sensoren, die Daten über die Umgebungseigenschaften erfassen. Dazu zählen beispielsweise Beschleunigung, Annäherung, aber auch Temperatur oder Luftdruck. Für unterschiedliche Aufgaben werden unterschiedliche

Sensoren benötigt, wie beispielsweise der Näherungssensor für *Diebstahl-Alarm* oder der Lagesensor für *Dreh-Zähler*. Für die in dieser Arbeit enthaltenen Übungsaufgaben werden folgende Sensortypen verwendet: Beschleunigungssensor, Gyroskop und Annäherungssensor.

Beschleunigungs- bzw. Lagesensoren messen die Beschleunigung in m/s^2 für die drei Bewegungsrichtungen: X-, Y- und Z-Achse in einem festgelegten Zeitraum. Zur Übersicht sind diese in Abbildung 7.4 dargestellt. Die Erdbeschleunigung ist auch in den

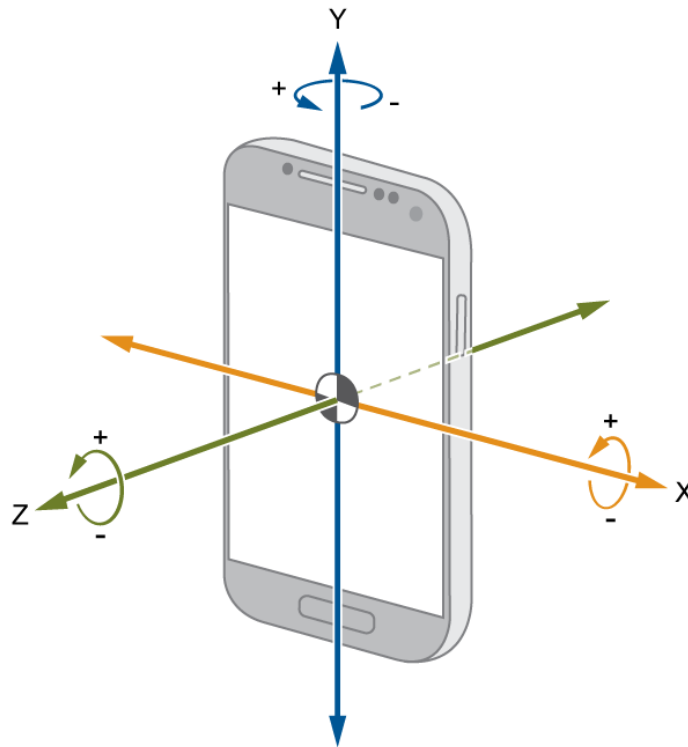


Abbildung 7.4: Android-Koordinatensystem

Messwerten des Lagesensors enthalten. Diese muss für die bereinigten, realen Werte von den aufgenommenen Werten subtrahiert werden[10]. Messeinheiten unterscheiden sich je nach Sensor. Das Gyroskop misst beispielsweise keine Beschleunigung, sondern die aktuelle Geschwindigkeit in rad/s der gleichen Achsen.

Die Frequenz, mit der Messwerte erfasst werden, kann manuell angegeben werden. Hierfür stehen vier Stufen zur Auswahl, welche in Tabelle 7.1 dargestellt sind. In der Android-App erfolgen alle Messungen unter der Verzögerungsstufe `SENSOR_DELAY_NORMAL`. Die Stufe gilt für Android jedoch nicht als festes Limit, sondern wird eher als Richt-Frequenz behandelt. Android kann die reale Frequenz auch erhöhen.

Bezeichnung	Maximale Verzögerung
SENSOR_DELAY_FASTEST	Keine. Verwendet die Frequenz des Sensors.
SENSOR_DELAY_GAME	20 ms
SENSOR_DELAY_UI	60 ms
SENSOR_DELAY_NORMAL	200 ms

Tabelle 7.1: Sensor-Taktgeschwindigkeiten[11]

Kapitel 8

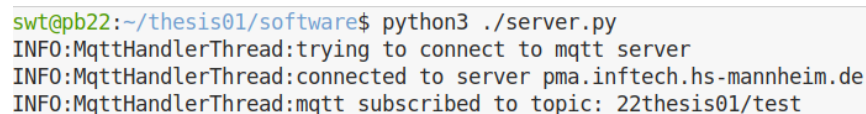
Evaluation der Smartbit-Lösung

Die Umsetzung der an die implementierte Anwendung gestellten Anforderungen wird in diesem Kapitel überprüft. Durch Betrachtung eines Verwendungsbeispiels werden die Anforderungen qualitativ untersucht und anschließend bewertet. Geringen Latenzzeiten wurden in der Konzeptionsphase eine hohe Priorität zugeordnet. Diese werden anhand der Smartbit-Lösung für drei Nutzungsszenarien gemessen und bewertet.

8.1 Verwendungsbeispiel

Die Verwendung der Smartbit-Lösung wird anhand der Beispielaufgabe *Alarmanlage* vorgestellt. Gezeigt wird wie die Aufgabe in der Programmierumgebung unter Verwendung der Python-Bibliothek gelöst wurde und wie sich Ausgaben auf dem Smartphone äußern. Die Android-Anwendung dieses Beispiels wurde auf den Smartphones Motorola G6 (Android 9), OnePlus 3T (Android 11) und Google Pixel 4 (Android 12) getestet.

Eine gestartete Kontrollanwendung ist Voraussetzung für einen Nachrichtenaustausch. Mit dem Befehl `python ./server.py` wird sie in einer Shell gestartet. Der Vorgang wird in Abbildung 8.1 dargestellt. Die Anwendung meldet eine er-



```
swt@pb22:~/thesis01/software$ python3 ./server.py
INFO:MqttHandlerThread:trying to connect to mqtt server
INFO:MqttHandlerThread:connected to server pma.inftech.hs-mannheim.de
INFO:MqttHandlerThread:mqtt subscribed to topic: 22thesis01/test
```

Abbildung 8.1: Start der Kontrollanwendung

folgreiche Verbindung mit dem MQTT-Broker und gibt das abonnierte Topic aus.

Neben diesen Start-Informationen werden auch eingehende `rpc_requests` und `rpc_responses` ausgegeben. Häufig versandte Nachrichtentypen wie `sensor_requests`, `sensor_responses` und `update_requests` werden nicht geloggt.

Die Implementierung der Lösung der Aufgabe ist in Listing 8.1 dargestellt.

```
1 from time import sleep
2 import smartbit
3
4 p = smartbit.Phone()
5
6 while True:
7     prox_val = float(p.get_proxy())
8     if prox_val == 0.0:
9         p.write_text("ALARM")
10        for _ in range(5):
11            p.vibrate(1000)
12            p.toggle_led()
13            sleep(0.2)
14        sleep(0.5)
```

Listing 8.1: Alarmanlage-Beispiel

Die Bibliothek wird in Zeile 2 in das vom Programmierer geschriebene Programm importiert, wofür sich die Datei `smartbit.py` und das entwickelte Programm im gleichen Ordner befinden müssen. In Zeile 4 wird ein `Phone`-Objekt erstellt, über welches Sensor-Auslesemethoden wie `get_x_accel()` oder Smartphone-Ausgaben wie `vibrate()` aufgerufen werden können. Das Programm soll nur im Falle eines `KeyboardInterrupts` angehalten werden. In einer Endlosschleife wird der Näherungssensorwert kontinuierlich abgefragt. Zusätzlich wird der Ablauf für 500 ms pausiert um einer Nachrichtenflut und somit einer Nichtverfügbarkeit der Kontrollanwendung vorzubeugen. Der Annäherungssensor sendet im Falle einer Annäherung den Wert 0.0 zurück auf den in Zeile 8 reagiert wird. Ist die Bedingung erfüllt, wird mit der Methode `write_text()` der Text `ALARM` ausgegeben. Die Methode `vibrate` lässt das Smartphone für die Dauer von 1000 ms vibrieren. Zum Schluss wird mit der Methode `toggle_led` noch der Farbwert der Signal-LED von grün auf rot geändert.

Die Bibliothek übermittelt für jede Ausgabe korrespondierende `rpc_requests` über die Kontrollanwendung an das Smartphone. Bei Empfang werden die Anfragen in der Kontrollanwendung geloggt. Eine Übersicht der gesendeten Nachrichten des Beispiels ist in Abbildung 8.2 aufgeführt. Zu erkennen sind die unterschiedlichen Ausgabekürzel der Anfragen, welche im Feld `command` abgebildet sind. Für die Textausgabe entspricht das Kürzel `write_text`, für Vibrationen `vibrate` und für das

```
swt@pb22:~/thesis01/software$ python3 ./server.py
INFO:MqttHandlerThread:trying to connect to mqtt server
INFO:MqttHandlerThread:connected to server pma.inftech.hs-mannheim.de
INFO:MqttHandlerThread:mqtt subscribed to topic: 22thesis01/test
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'write_text', 'value': 'ALARM'}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'vibrate', 'value': '1000'}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'led_toggle', 'value': ''}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'vibrate', 'value': '1000'}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'led_toggle', 'value': ''}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'vibrate', 'value': '1000'}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'led_toggle', 'value': ''}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'vibrate', 'value': '1000'}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'led_toggle', 'value': ''}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'vibrate', 'value': '1000'}
INFO:DataHandler:rpc request: {'type': 'rpc_request', 'command': 'led_toggle', 'value': ''}
```

Abbildung 8.2: Nachrichtenversand der Kontrollanwendung

Umschalten der LED-Farbe `led_toggle`. Pro Ausgabe-Kommando kann zusätzlich ein Parameterwert angegeben werden. Für `write_text` bestimmt er den anzuzeigenden Text und für `vibrate` die Vibrationsdauer in Millisekunden. Da es nicht vorgesehen ist die Farbe der Signal-LED manuell festzulegen, wird für `led_toggle` kein Parameterwert angegeben.

Wird die App auf dem Smartphone gestartet, befindet sie sich im Initialmodus in dem sie bereits Sensorwerte misst und an die Kontrollanwendung sendet. Das Userinterface ist in Abbildung 8.3 dargestellt. Es besteht aus zwei mit A und B beschrifteten

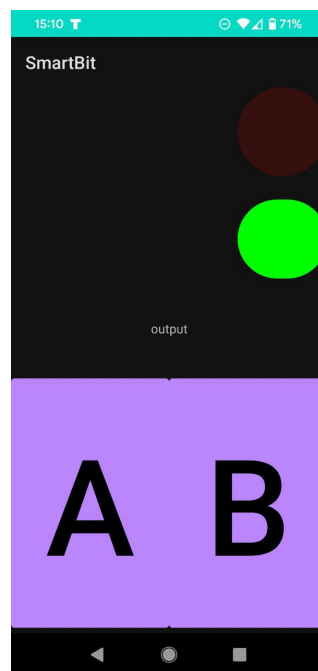


Abbildung 8.3: Initialzustand der mobilen Anwendung

Buttons, einem Textfeld in der Mitte, einer Signal-LED welche zu Beginn grün leuchtet

und einer Vorgangs-LED, welche während der Ausführung von Ausgaben aufleuchtet.

Im Alarmfall werden die vom Programm gesendeten Ausgaben entsprechend umgesetzt und das Aussehen des Userinterfaces verändert. Das Resultat ist in Abbildung 8.4 dargestellt. Das Textfeld stellt nun den Text *ALARM* dar und die Farbe der Signal-

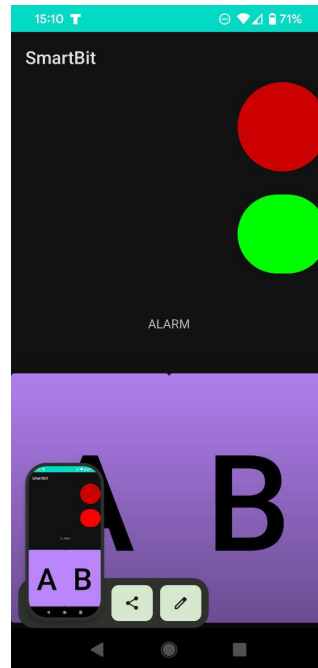


Abbildung 8.4: Alarmzustand der mobilen Anwendung

LED hat sich von grün auf rot geändert. Die Vorgangs-LED leuchtet rot um zu signalisieren dass gerade eine Ausgabe ausgeführt wird. Diese ist nicht sichtbar, denn es handelt sich um das haptische Vibrationsfeedback.

Nutzerinnen und Nutzer können alle in den Anforderungen aufgestellten Ausgabemöglichkeiten und Sensortypen nutzen. Implementierte Funktionen kommunizieren zwar mit der Kontrollanwendung in serialisierten Nachrichtenformaten, geben an die aufrufenden Programme jedoch primitive Datentypen zurück. Die Bibliothek ist portabel nutzbar, da die Stub-Funktionen jeweils für die Programmiersprachen C, Java und Python implementiert wurden. Eine Einbindung in Eclipse IDE ist möglich, jedoch durch die zusätzlich benötigten JSON-Parser-Bibliotheken für C- und Java-Projekte nicht trivial. Da Java und Python plattformunabhängige Programmiersprachen sind, erübrigt sich die zweifache Implementierung für Unix- und Windows-Systeme. Smartbit ermöglicht ein responsives Verhalten der Komponenten. Im vorangegangenen Beispiel *Alarmanlage* betrug die Reaktionszeit unter einer Sekunde. Smartphonesseitige Latenzen werden architekturbedingt durch die

Zwischenspeicherung von Kontrollanwendung verschleiert. Von Umgebungseigenschaften abhängige Variierungen der Übertragungsgeschwindigkeit ist dadurch für die Programmiererin oder den Programmierer nicht sichtbar. Mit der Verwendung von MQTT und UDP wurden leichtgewichtige, ihrem Einsatzzweck entsprechende Protokolle für die Kommunikation gewählt. Trotz der Übertragung mit einem QoS-Level 0 werden auch Ausgabeanfragen sicher übertragen und ausgeführt. Über die MQTT-Verbindung zwischen Kontrollanwendung und Smartphone-App versandte Nachrichten, werden auf dem Übertragungsweg mit TLS verschlüsselt übertragen. Über die die UDP-Verbindung zwischen Kontrollanwendung und Bibliothek übertragene Nachrichten, werden nicht verschlüsselt. Ein unberechtigtes Mitlesen des Datenverkehrs setzt jedoch voraus, dass Angreifer Nutzerrechte auf dem PC der Nutzerin oder des Nutzers besäßen, da Datagramme lediglich über das Loopback-Interface ausgetauscht werden können. Das Smartphone ist vor einer RCE (Remote Code Execution)-Attacke geschützt, da nur Ausgaben ausgeführt werden können, die von der API spezifiziert wurden. Smartbit setzt durch die Implementierung von Log-Ausgaben in Kontrollanwendung und Smartphone ebenfalls die Anforderungen an die Transparenz um.

8.2 Latenzmessung

Im Zuge der Konzeption wurden Latenzen eine besondere Bedeutung zugemessen. Erhöhte Latenzzeiten führen zu einem verzögerten Verhalten und einer schlechteren Benutzbarkeit. Zum Zwecke der Evaluation der Latenzen werden für drei Nutzungsszenarien Messungen durchgeführt. Da `sensor_requests` lediglich über die Local-Loopback-Schnittstelle zwischen Bibliothek und Kontrollanwendung ausgetauscht werden ist anzunehmen, dass diese Verbindung in der Smartbit-Lösung keine Limitierung darstellt. Auf die Untersuchung der Latenzzeiten zwischen dem Absenden eines `sensor_requests` und dem Erhalt einer `sensor_response` wird daher verzichtet.

Unbekannt ist indes die Latenz zwischen Smartphone-App und Kontrollprogramm. Über diesen Transportweg werden die Nachrichtenformate `rpc_request`, `rpc_response` und `update_request` übermittelt. Durch die marginale Latenz zwischen Kontrollanwendung und Bibliothek können Sensorwerte zwar häufig abgefragt werden, jedoch wird bis zum Eintreffen eines neuen Sensormesswerts der zuletzt eingespeicherte zurückgegeben. Sensordatenabfragen sind daher abhängig von der MQTT-Verbindung zwischen Smartphone und Kontrollanwendung. Um ein realistisches Nutzungsverhalten zu simulieren, wird für die Messungen der Transportweg zwischen Bibliothek und Smartphone-App betrachtet. Gemessen wird die Zeitdauer nach Versand eines `rpc_requests` bis zum Erhalt einer `rpc_response`

von der Smartphone-App. Die Smartphone-App sendet währenddessen fortlaufend `update_requests` per MQTT um die reguläre Last auf der Kontrollanwendung zu simulieren. Messungen finden für drei verschiedene Nutzungsszenarien statt: Der Verwendung im Labor des SWT-Instituts an der Hochschule, der Heimarbeit über eine DSL-Verbindung und der Verwendung in einer virtuellen Umgebung. Für alle Messungen erfolgen Internetzugriffe seitens des lokalen PCs kabelgebunden und seitens des Smartphones über WLAN. Die Anzahl der verbundenen Geräte des AccessPoints sind für die Übertragungszeit ebenfalls ausschlaggebend. Die bei CS-MA/CA zum Kollisionsschutz verwendeten randomisierten Sendefenster limitieren mit steigender Geräteanzahl die Sendezeit und erhöhen die Latenzen. Um signifikante Ergebnisse zu diskutieren werden pro Nutzungsszenario 100 Messwerte erhoben und in einem Boxplot dargestellt.

Mit dem SWT-Labor befindet sich der lokale PC im ersten Nutzungsszenario im gleichen Netzbereich wie der MQTT-Broker, was den Transportweg reduziert. Die Messergebnisse sind in Abbildung 8.5 dargestellt. Die Latenzen sind mit Maximal-

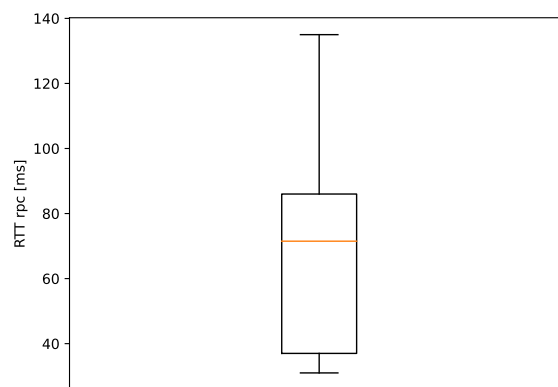


Abbildung 8.5: Messergebnisse der Latenzmessungen im Heimnetz

werten von 138 ms sehr gering. `SensorEventListener` erheben Daten mit einer Verzögerung von maximal 200 ms. Hier könnte auch die eigentliche Sensormessung auf dem Smartphone die Übertragungsgeschwindigkeit begrenzen. Im Labor-Szenario kann die Smartbit-Lösung eingesetzt werden.

Die Messung des zweiten Szenarios erfolgt aus einem Heimnetz mit acht Hops zwischen Broker und lokalem PC. Auf dem Access Point sind zur Zeit der Messung acht Geräte registriert. Die Messergebnisse sind relevant für die Heimarbeit oder für die Verwendung beim Erledigen von Hausaufgaben. Die Ergebnisse sind in Abbildung 8.6 dargestellt. Die Messwerte fallen mit einem Median von etwa 175 ms deutlich höher als im Labor aus, befinden sich allerdings immer noch im tolerablen Bereich und

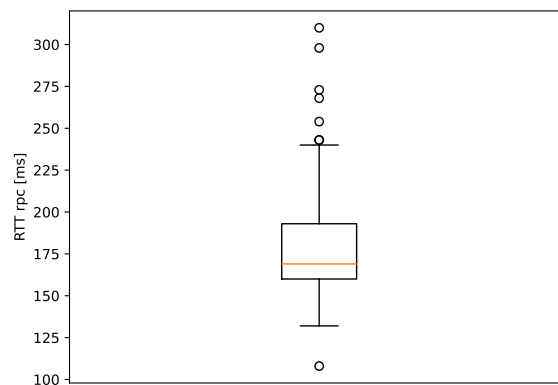


Abbildung 8.6: Messergebnisse der Latenzmessungen im Heimnetz

stellen keine gravierende Beeinträchtigung in der Benutzung der Smartbit-Lösung dar.

Im dritten Szenario wird die Messung von einer in VirtualBox gestarteten, virtuellen Maschine auf dem lokalen PC untersucht. Sowohl Bibliothek und Kontrollanwendung werden in der virtuellen Maschine betrieben. Relevant ist das Szenario ebenfalls für die Heimarbeit. Es bildet die Verwendung der virtuellen Maschine zum Programmieren ab. Die Messergebnisse sind in Abbildung 8.7 dargestellt. In den

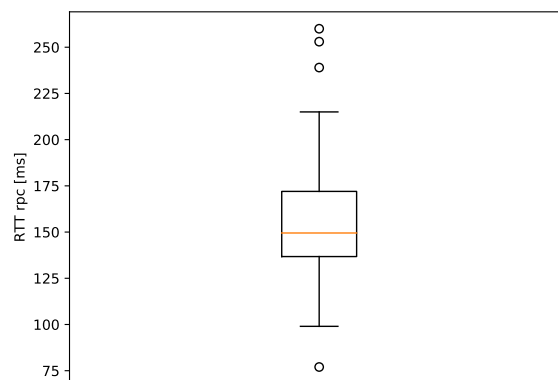


Abbildung 8.7: Messergebnisse der Latenzmessungen im Heimnetz

Lehrveranstaltungen des Instituts wird dies häufig angewandt um Studierenden auf simplem Wege eine vollständige Programmierungsumgebung zur Verfügung zu stellen. Untersucht wird, ob die Latenzzeiten sich durch die NAT-Funktionalität von VirtualBox signifikant vergrößern. Die Latenzzeiten sind im Vergleich zur Messung

auf dem Host-PC nicht wesentlich angestiegen. Dadurch ist ebenfalls von keiner maßgeblichen Beeinträchtigung der Nutzung auszugehen. Smartbit kann auch in virtuellen Entwicklungsumgebungen verwendet werden.

Kapitel 9

Fazit

Die Smartbit-Lösung erfüllt die Anforderungen zufriedenstellend. Die im Vorfeld durchgeführte Konzeption ermöglichte eine problemlose Implementierung. Während der Entwicklung wurden jedoch einige Designentscheidungen getroffen, welche die Smartbit-Lösung beschränken. Sensordaten werden von der Kontrollanwendung nicht für einzelne Geräte eingespeichert. Es ist nicht möglich, Programmcode simultan auf mehreren Smartphones gleichzeitig auszuführen. Gruppen-Sessions können nicht erstellt werden. Für Programme die die Java und Python-Bibliothek nutzen erscheint dieses Konzept kontraintuitiv. Die Bedeutung eines Phone-Objektes lässt auf eine zwischen Einzelgeräten unterscheidende Verwendung schließen. Ein weiterer Nachteil der Benutzerfreundlichkeit besteht in der zwingenden Verwendung von Hilfs-Bibliotheken für den Nachrichtenaustausch über JSON. Sowohl für Java, als auch für C sind zusätzliche JSON-Parser für die Kommunikation zwischen Bibliothek und Kontrollanwendung nötig. Für eine erfolgreiche Kompilierung in C muss der Dateipfad für diesen dem Linker bekannt sein. Für die interne Struktur ist die Datenverwaltung innerhalb des Programms ausreichend. Durch den asynchronen Ansatz des Multithreading können beide Kommunikationspartner gleichzeitig über das Kontrollprogramm kommunizieren und zwischengespeicherte Daten auslesen oder bearbeiten. Die Möglichkeit, mehrere Werte zum internen Zwischenspeicher hinzuzufügen, wird jedoch nicht unterstützt. Häufige Sensor-Anfragen werden mit dem gleichen, einzeln zwischengespeicherten Sensorwert beantwortet, welcher zusätzlich kein Ablaufdatum besitzt. Abbrüche der Übertragung resultieren in der kontinuierlichen Übermittlung des zuletzt eingespeicherten Sensorwerts.

Zur Ablösung von Hilfsbibliotheken eignet sich die Verwendung eines Binärprotokolls. In Enumerationen konvertierte Nachrichtenformate und Parameter könnten statt der implementierten Darstellung mit JSON eingesetzt werden. Dadurch entfielen nicht nur der Nachteil der kontraintuitiven Einbindung. Die Nachrichtengröße

würde ebenfalls verringert werden. Energieeffizienz der Android-App wurde zum Vorteil der Latenzreduzierung während der Implementierung nicht wesentlich berücksichtigt. Zum Start der Anwendung werden die Sensormessprozesse unmittelbar gestartet und Sensorwerte an die Kontrollanwendung gesendet, was einen erhöhten Energieverbrauch bedeutet. Sollten sich die Startzeiten der Sensormessprozesse nicht wesentlich auf die Latenzen auswirken, wäre eine auf Anfragen basierende Sensormessdatenübertragung erwägenswert, um den Energieverbrauch zu reduzieren.

Abbildungsverzeichnis

2.1	Anwendungsfälle für die Benutzung von Smartbit	8
3.1	System-Aufbau	12
3.2	Nachrichtenablauf einer Ausgabe	14
4.1	Nachrichtenablauf der Sensordatenübermittlung	17
4.2	Nachrichtenablauf der RPC-Anfragen	18
5.1	System-Kontext der Bibliothek	20
6.1	Übersicht der Komponenten der Kontrollanwendung	21
6.2	Ablaufdiagramm MQTT Request	22
6.3	Ablaufdiagramm UDP Request	24
7.1	Ablaufdiagramm Android-Anwendung	26
7.2	Ausführung auf dem UI-Thread	27
7.3	Ablauf SensorEventListener	28
7.4	Android-Koordinatensystem	29
8.1	Start der Kontrollanwendung	31
8.2	Nachrichtenversand der Kontrollanwendung	33
8.3	Initialzustand der mobilen Anwendung	33
8.4	Alarmzustand der mobilen Anwendung	34
8.5	Messergebnisse der Latenzmessungen im Heimnetz	36
8.6	Messergebnisse der Latenzmessungen im Heimnetz	37

8.7	Messergebnisse der Latenzmessungen im Heimnetz	37
-----	--	----

Tabellenverzeichnis

2.1	Beispielprogrammieraufgaben	7
4.1	Nachrichten-Typen	16
4.2	Sensor-Kürzel mit Beschreibung	17
7.1	Sensor-Taktgeschwindigkeiten[11]	30

Abkürzungsverzeichnis

API Application Programming Interface.

CSMA/CA Carrier Sense Multiple Access/Collision Avoidance.

IDE Integrated Development Environment.

JSON JavaScript Object Notation.

MQTT Message Queuing Telemetry Transport.

QoS Quality of Service.

RCE Remote Code Execution.

RMI Remote Method Invocation.

RPC Remote Procedure Call.

RTT Round Trip Time.

UDP User Datagram Protocol.

UMTS Universal Mobile Telecommunications System.

Literaturverzeichnis

- [1] Martin Winkler Christoph Lüders. c't 23 - pingpong, 2006.
- [2] Martinha Piteira and Carlos Costa. Learning computer programming: Study of difficulties in learning programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication, ISDOC '13*, page 75–80, New York, NY, USA, 2013. Association for Computing Machinery.

Online-Quellen

- [3] Arduino. Kits-arduino official store. <https://store.arduino.cc/collections/kits>. [letzter Zugriff: 08. Juni. 2022].
- [4] TIOBE Software BV. Rfc 8259. <https://www.tiobe.com/tiobe-index/>. [letzter Zugriff: 06. Juli. 2022].
- [5] Steve Cope. Mqtt client and mosquitto broker message restrictions. <http://www.steves-internet-guide.com/mqtt-broker-message-restrictions/>. [letzter Zugriff: 03. Juli. 2022].
- [6] Eclipse. Paho. <https://www.eclipse.org/paho/>. [letzter Zugriff: 03. Juni. 2022].
- [7] Financesonline. Number of smartphone users worldwide 2022. <https://financesonline.com/number-of-smartphone-users-worldwide/>. [letzter Zugriff: 08. Juni. 2022].
- [8] Eclipse Foundation. Eclipse desktop ides. <https://www.eclipse.org/ide/>. [letzter Zugriff: 03. Juli. 2022].
- [9] Dave Gamble. Github - davegamble/cjson: Ultralightweight json parser in ansi c. <https://github.com/DaveGamble/cJSON>. [letzter Zugriff: 07. Juli. 2022].
- [10] Google. Android developers - accelerometer example. https://developer.android.com/guide/topics/sensors/sensors_motion#sensors-motion-accel. [letzter Zugriff: 04. Juni. 2022].
- [11] Google. Sensors - overview. https://developer.android.com/guide/topics/sensors/sensors_overview#sensors-monitor. [letzter Zugriff: 22. Juni. 2022].

- [12] Sean Leary. Github - stleary/json-java: A reference implementation of a json package in java. <https://packages.debian.org/bullseye/libcjson1>. [letzter Zugriff: 07. Juli. 2022].
- [13] Sean Leary. Maven repository json. <https://mvnrepository.com/artifact/org.json/json/20220320>. [letzter Zugriff: 07. Juli. 2022].
- [14] python. queue. <https://docs.python.org/3/library/queue.html>. [letzter Zugriff: 03. Juni. 2022].
- [15] Ed. T. Bray. Rfc 8259. <https://datatracker.ietf.org/doc/html/rfc8259>. [letzter Zugriff: 05. Juli. 2022].
- [16] Boyuan Yang. Debian – details of package libcjson1 in bullseye. <https://packages.debian.org/bullseye/libcjson1>. [letzter Zugriff: 07. Juli. 2022].

Anhang A

Nachrichtenformate

```
1 {  
2   "type": "update_request",  
3   "sensor_type": "",  
4   "sensor_value": ""  
5 }
```

Listing A.1: Update-Request

```
1 {  
2   "type": "sensor_request",  
3   "sensor_type": ""  
4 }
```

Listing A.2: Sensor-Request

```
1 {  
2   "type": "sensor_response",  
3   "value": ""  
4 }
```

Listing A.3: Sensor-Response

```
1 {  
2   "type": "sensor_request",  
3   "sensor_type": ""  
4 }
```

Listing A.4: RPC-Request

```
1 {  
2   "type": "sensor_response",  
3   "value": ""  
4 }
```

Listing A.5: RPC-Response