



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

Um Programmieraufgaben interaktiv zu gestalten eignen sich Projekte mit Microcontrollern besonders gut. Smartphones bieten einen vergleichbaren Funktionsumfang und müssen meist nicht zusätzlich beschafft werden. In dieser Arbeit wurde eine Softwarelösung erstellt, um Smartphonesensoren über eine Programmierumgebung auszulesen und Ausgaben auf dem Smartphone auszuführen. Hierfür wurde eine Android-Anwendung, eine Kontrollanwendung und eine programiersprachenunabhängige Softwarebibliothek erstellt.

Für die Nutzung der Lösung werden Beispiel-Programmieraufgaben dazugereicht. Programmierer schreiben Programme auf dem PC, welche auf Änderungen von Smartphonesensorwerten wie beispielsweise Beschleunigungssensoren reagieren und die Ausgabemöglichkeiten des Smartphones nutzen.

Inhaltsverzeichnis

1	Einführung	2
2	Smartphones als Microcontroller-Ersatz	5
2.1	Beispielprogrammieraufgaben	5
2.2	Anforderungen der Implementierung	7
3	Architektur	9
4	Nachrichtenformate	11
5	Android Anwendung	15
5.1	Startvorgang	15
5.2	Sensoren	18
6	Kontrollanwendung	20
7	Programmierungsumgebung	23
8	Evaluation	25
8.1	Verwendungsbeispiel	25
8.2	Latenzmessung	28
9	Fazit	31
	Abkürzungsverzeichnis	35
	Literaturverzeichnis und Online-Quellen	36

Kapitel 1

Einführung

Viele Programmierer oder programmierwillige Anfänger mühen sich beim Programmierenlernen mit der Semantik von Programmiersprachen und grundlegenden algorithmischen Konzepten. Akademische Übungsaufgaben senken die Lernmotivation durch rein virtuelle Aufgabenstellungen ohne Interaktionsmöglichkeiten. Projekte mit Microcontrollern dagegen bieten eine praktische, fordernde und spielerische Einstiegsmöglichkeit. Es werden kleine Projekte realisiert, die durch die Verwendung von Sensoren Programmierer einladen sich an Programmieraufgaben auszuprobieren. Diese Eigenschaften sind sinnvoll, insbesondere bei Projekten für Programmierer mit wenig Vorwissen wie Schüler oder Erstsemester-Studierende. Gelerntes kann direkt angewandt werden. Praktische Programmieraufgaben bieten für Programmieranfänger den besten Lerneffekt bei höchster Motivation [2]. Die in Microcontroller integrierten Sensoren sind Voraussetzung um physikalische Eigenschaften in der realen Welt zu messen. Programme auf dem Microcontroller können die Sensoren auslesen und auf Änderungen der gemessenen Werte reagieren. Sensoren, Microcontroller und das entwickelte Programm ermöglichen zusammen eine Bedienung durch Nutzer. 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 Änderungen des Codes, schafft Routine in der Entwicklung und damit ein tieferes Verständnis und Hintergrundwissen für die Problemstellung.

Microcontroller-Projekte benötigen allerdings kostspielige Einstiegs-Kits. Ein Arduino-Development-Board kostet im Arduino-Shop über 80,00 € [3]. Ein Großteil der Kosten entfällt zwar auf den eigentlichen Microcontroller, ein nicht unmittelbarer Teil jedoch auf Peripherie wie Breadboards, Verbindungskabel und Erweiterungsboards. Die Peripherie-Anbindung setzt daneben Hintergrundwissen in elektrotechnischen Bereichen voraus, wie zum Beispiel den Verschaltungskonventionen bei Breadboards. Dies stellt ebenfalls eine Einstiegshürde dar, die die eigentliche interaktive Lernerfahrung

herauszögert und die Motivation senkt.

Smartphones dienen für Einstiegs-Programmierer als Alternative zu herkömmlichen Microcontrollern. Integriert sind zahlreiche Sensoren wie Lagesensoren, Gyroskop oder Näherungssensoren. Der Sensoren-Umfang ist vergleichbar mit dem von Microcontrollern. Elektrische Bauteile konventioneller Microcontroller-Sets erlauben einen Fehlgebrauch, der im schlimmsten Fall in der Zerstörung von Komponenten enden kann. Projekte mit Smartphones reduzieren dieses Risiko dadurch, dass Schaltkreise bereits intern verknüpft und somit von äußerlicher Fehlverwendung geschützt sind. Ein weiterer Vorteil Smartphones gegenüber Microcontrollern liegt in der Verfügbarkeit. Weltweit besaßen 2022 5,2 Mrd. Menschen ein Smartphone [5]. Sie sind gerade unter Kindern, Jugendlichen und jungen Erwachsenen weit verbreitet. Kinder besitzen häufig bereits mit 10 Jahren ein Smartphone [8]. Im Alltag wird es 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. Durch Ihre eingesetzten Sensoren können Sie zuverlässig physikalische Umgebungseigenschaften messen. Neben kabelgebundenen Übertragungsschnittstellen wie USB besteht auch die Möglichkeit sich mit drahtlosen Verbindungsmöglichkeiten wie WLAN zu verbinden. Die Geräte sind zudem batteriebetrieben, was Lösungen ermöglicht die von einer Spannungsversorgung unabhängig sind. Eine Einbindung von Smartphones ist in den meisten Entwicklungsumgebungen jedoch nicht möglich. Visuelle und haptische Ausgaben auf dem Smartphone erfordern zudem eine mobile Anwendung, da Smartphone-Betriebssysteme keine nativen Ausgabemethoden außerhalb von Apps bieten.

Ziel dieser Arbeit ist die Entwicklung einer Softwarelösung, die die Möglichkeiten von Smartphones für die Entwicklung von interaktiven Projekten nutzbar macht. Sensoren sollen ausgelesen und Ausgaben auf dem Smartphone ausgelöst werden können. Für die Verwendung der Lösung werden angehenden Programmierern Beispielaufgaben gereicht. Um die Beispielaufgaben zu bewältigen muss die Lösung auch Benutzungsmöglichkeiten bereitstellen. Hierdurch werden Anforderungen an die in dieser Arbeit implementierte Lösung gestellt. Die Aufgaben, Anforderungen und Rahmenbedingungen sind in Kapitel 2 zu finden. Die drei Komponenten Smartphone-App, Kontrollanwendung und Programmierumgebung und ihr Zusammenspiel werden in Kapitel 3 vorgestellt. Die dafür benötigten Nachrichtenformate werden in Kapitel 4 gezeigt. Ihr Zweck wird erklärt und der Nachrichtenaustausch dort exemplarisch veranschaulicht. Kapitel 5 behandelt die Funktionsweise und den Aufbau der Android-App im Detail. Diese tauscht Nachrichten mit der Programmierumgebung aus. Als Zwischenvermittlung fungiert das zentrale Kontrollprogramm, was in Kapitel 6 erklärt wird. Einbindung, Nutzung, und externe Schnittstellen der Bibliotheken zur Android-App und dem Kontrollprogramm wer-

den in Kapitel 7 erklärt. In Kapitel 8 wird untersucht, ob die vorgegebenen Anforderungen erfüllt wurden. Dort wird zudem die Verwendung der Lösung anhand einer Beispielaufgabe vorgestellt. Schwierigkeiten die bei der Entwicklung auftraten werden in Kapitel 9 diskutiert. Erweiterungsmöglichkeiten und Verbesserungen werden diskutiert.

Kapitel 2

Smartphones als Microcontroller-Ersatz

Smartphones sind in sich geschlossene technische Geräte, die neben vordefinierten Verbindungsschnittstellen wie einem USB-Port, WLAN und Bluetooth keine weiteren Schnittstellen bieten um externe Hardware und Schaltungen anzuschließen und fernzusteuern. Microcontroller-Schaltungen zum Programmierenlernen bieten meistens mehrere Ausgabemöglichkeiten wie LEDs, Lautsprecher oder Piepser. Smartphones können diese Bausteine nicht anschließen aber virtuell darstellen. Gewohnte Ausgabelemente können virtualisiert werden. Die Funktionen sind außerdem nicht nur von Mehrzweck-Ausgaben, wie LED-Grids begrenzt, die zum Beispiel für die Text- oder Bildanzeige verwendet werden können. Zweckgebundene Elemente wie Textfelder, Textausgaben oder Bildausgaben in der App können beliebig kombiniert werden. Darüber hinaus ist die Anordnung der jeweiligen Elemente frei wählbar, so dass das Layout anders als bei Microcontrollern auch im Nachhinein noch geändert werden kann. Die Ausgabemöglichkeiten können in Kombination mit Sensor-Daten verwendet werden, um kleine Programmieraufgaben zu lösen. Einige Beispiele werden in diesem Kapitel vorgestellt. Da diese nicht auf einem Microcontroller, sondern einem Smartphone ausgeführt werden, müssen gewisse Rahmenbedingungen erfüllt sein, wie beispielsweise geringe Latenzen in der Sensordatenübertragung.

2.1 Beispielprogrammieraufgaben

Praxisnahe Programmieraufgaben mit interessanten Aufgabestellungen motivieren Softwareentwickler. 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

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

Name der Aufgabe	Benötigte Sensoren	Verwendete Ausgaben	Schwierigkeitsgrad
Disco	-	Led	+
Würfeln	Lagesensor	Textfeld	+
Diebstahl-Alarm	Näherungssensor	Textfeld, Led, Vibration	++
Klatsch-Zähler	Mikrofon	Textfeld	++
Dreh-Zähler	Lagesensor	Textfeld	+++

Tabelle 2.1: Beispielprogrammieraufgaben

In der Aufgabe *Disco* soll eine virtuelle LED für eine Zeitdauer von 500 ms grün und anschließend 500ms rot leuchten. Die Verwendung Sensoren ist in dieser Aufgabe nicht nötig, da die LED-Ausgabe unabhängig von Sensormesswertänderungen ausgeführt wird. Aus diesem Grund ist die Aufgabe als Einfach eingestuft.

In der Aufgabe *Würfeln* soll ein Schütteln des Geräts erkannt werden. Zur Verwendung kommt dabei ein Lagesensor zum Einsatz welcher Gerätebeschleunigungen messen kann. Wird ein Schütteln erkannt, soll auf dem PC eine Zufallszahl generiert werden. Anschließend wird diese auf dem Gerät in einem Textfeld ausgegeben. Der Schwierigkeitsgrad wird ebenfalls auf Einfach eingeschätzt, da die Aufgabe unter Verwendung eines Sensors und einer Ausgabe lösbar ist.

In der Aufgabe *Diebstahl-Alarm* wird unter Verwendung des Näherungssensors das örtliche Umfeld des Geräts auf eine Annäherung kontrolliert. Wird eine Annäherung festgestellt, wird ein alarmierender Text im Textfeld ausgegeben. Zusätzlich soll die LED die wie in Aufgabe *Disco* die Farbe wechseln. Neben den visuellen Ausgaben wird die Vibrationsfunktion als haptisches Feedback benötigt. Im Falle einer Annäherung, soll das Gerät fünf mal vibrieren. 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 werden können. Alle Ausgaben werden auf ihren initialwert zurückgesetzt.

Bei der Aufgabenstellung *Klatsch-Zähler* muss für einen definierten Zeitraum die Anzahl der Händeklatscher gemessen werden. Diese Anzahl wird anschließend im Textfeld des Geräts ausgegeben. Zur Verwendung kommen hierfür das Mikrofon als Ein- und das Textfeld 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 Festlegung helfen.

Bei der letzten Aufgabe *Drehzähler* soll der Programmierer das Device innerhalb eines definierten Zeitraums drehen und anhand der Messdaten die Anzahl der Umdrehungen ermitteln. Diese Anzahl soll anschließend im Textfeld auf dem Gerät ausgegeben werden. Zur Verwendung kommen 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 zwischen den Messweltergebnissen erschweren eine Umdrehungserkennung zusätzlich.

2.2 Anforderungen der Implementierung

Aus den Beispielaufgaben leiten sich Anforderungen an die Lösung der ab.

Geringe Latenzen sind bei der Sensordatenübermittlung für ein responsives Verhalten von Programmen nötig. Sie tragen zur Lernerfahrung bei, da sich physikalische Änderungen unmittelbar auf das Verhalten des entwickelten Programms auswirken. Eine Verkürzung der Latenzen ist von mehreren Faktoren abhängig.

Für die Übermittlung von Sensorwerten müssen diese initial vorliegen. Sensormessprozesse müssen gestartet und Sensoren ausgelesen werden, was die Latenzzeiten erhöht. Sensormessungen sollten daher nicht erst nach einer Anfrage oder einem Ereignis, sondern direkt zu Anfang gestartet werden und kontinuierlich messen. Der Versand der Sensorwerte wird durch den Transportweg zwischen Smartphone und Programmierungsumgebung und deren lokale Gegebenheiten verzögert. Smartphones bieten keine kabelgebundenen Netzwerkschnittstellen. Latenzen können je nach Mobilfunk- bzw WLAN Standard, Umwelteinflüssen und der Geräteanzahl in Funkzellen variieren. Für Latenzen dient die Round-Trip-Time ([RTT](#)) als Messgröße. Sie beschreibt die Zeitdauer der Übermittlung einer Nachricht über Hin- und Rückweg eines Hosts zu einem Anderen. Präventionsprinzipien wie [CSMA/CA](#) verhindern bei WLAN Verbindungen Interferenzen, erhöhen allerdings die Latenzen. Während sie bei Standards wie WLAN 802.11b ca. 10 ms beträgt, kann Sie bei [UMTS](#) auf 300 ms bis 400 ms ansteigen [\[1\]](#). Zudem ist Sie variabel, was bei einer synchronen Übertragung zu ungewollten Verzögerungen führt. Um die Latenz des Transportweges geringer darzustellen müssen Sensorwerte zwischengespeichert werden, um einen Puffer aufzubauen auf den in Fällen erhöhter Latenz zugegriffen werden kann. Eine weitere Latenz-Optimierung ist durch den Einsatz effizienter Protokolle möglich. Der Verlust weniger Sensorwerte ist für die Funktionsweise des Programms unerheblich.

Verbindungsorientierte Protokolle erhöhen die Menge der zu sendenden Nachrichten pro Sensorwert. Verbindungslose Protokolle sind deshalb zu bevorzugen.

Die implementierte Lösung muss einen benutzerfreundlichen Nutzungszugang für Programmierer und die Anbindung ihrer Programme bieten. Boilerplate-Code soll reduziert werden, um die Verwendung zu erleichtern. Die angebotene Funktionalität soll mit möglichst wenig Vorwissen nutzbar sein. Auf serialisierte Ausgabeformate muss verzichtet werden. Funktionen müssen Rückgabewerte in Form von primitiven Datentypen zurückgeben. Für die Bedienung aller Ausgabemöglichkeiten des Smartphones wird muss die Lösung eine Funktion zur programmierung bereitstellen. Das Einbinden in bestehende Entwicklungsumgebungen ist neben der benutzerfreundlichen Schnittstellengestaltung Voraussetzung für die Nutzung der Lösung. Die Entwicklungsumgebung Eclipse ist im Labor-Kontext weit verbreitet. Da es sich um eine plattformunabhängige [IDE](#) handelt, muss auch die Lösung plattformübergreifend in diese Entwicklungsumgebung integrierbar sein. Nachrichtenabläufe sollen nachvollziehbar sein um fehlerhafte Konfigurationen rasch zu identifizieren und zu korrigieren. Hierfür muss es eine Logging-Funktion geben die den Nachrichtenverlauf aufzeichnet und ausgibt. Programmierer können anhand der Logeinträge den Versand nachvollziehen wodurch Fehler ersichtlich werden.

Neben den Netzwerktechnischen- und Nutzungsbezogenen Anforderungen muss die Lösung auch Ausgabemöglichkeiten über die grafische Benutzeroberfläche einer Smartphone-App bereitstellen. Teil dieser Oberfläche ist eine farbwechselbare Signal-LED. Zur Ausgabe von Texten und Zeichen muss ein Textfeld implementiert werden. Durch zwei Tasten muss der Nutzer außerdem mit der Anwendung interagieren können. Neben den optischen Ausgaben bzw. Bedienelementen muss die Smartphone-App auch haptische Ausgaben wie Vibrationen umsetzen können. Für die Einstellung von Vibrationsmustern muss eine Vibrationszeitdauer konfigurierbar sein.

Kapitel 3

Architektur

Die implementierte Lösung 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

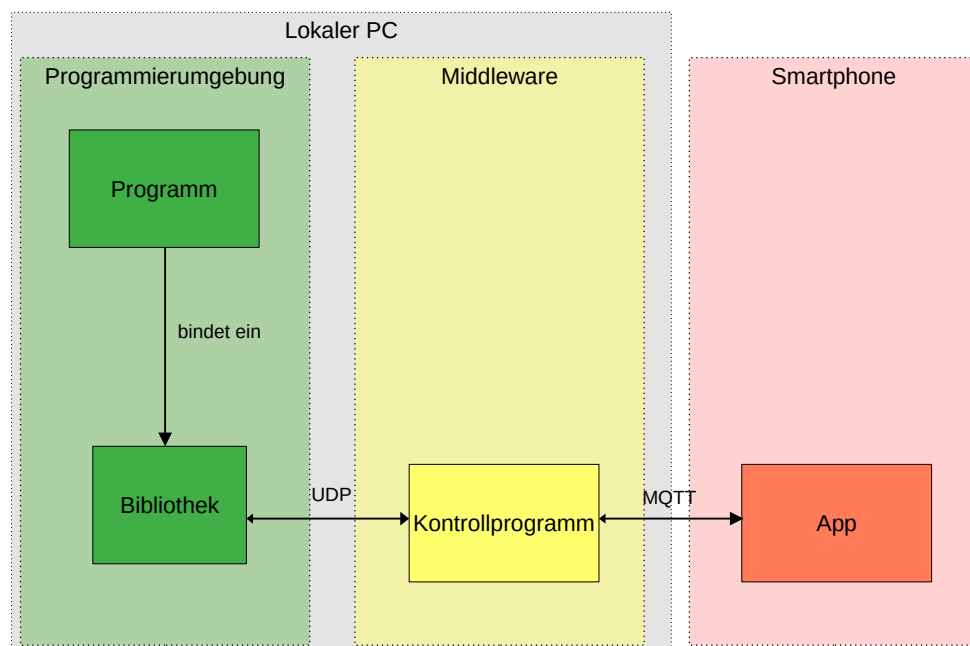


Abbildung 3.1: System-Aufbau

ung eine programmiersprachenunabhängige Bibliothek mit Schnittstellen in Form von Stub-Funktionen an, die die [API](#) der Kontrollanwendung nutzen. Die Bibliothek

kann in bestehenden Programmcode zum Zweck der Nutzung eingebunden werden. Zur Unterstützung der Verständlichkeit für angehende Programmierer wurde bei der Entwicklung der Bibliothek auf die Implementierung von 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 Programmierumgebung. Führt ein Programmierer einen Funktionsaufruf der Bibliothek aus, übermittelt die Bibliothek diesen per [UDP](#) dem Kontrollprogramm welches über die weitere Verfahrensweise entscheidet. Bei einer Ausgabeanfrage wird die Nachricht beispielsweise per [MQTT](#) an das Smartphone übertragen. Die Kontrollanwendung speichert zudem Sensormesswerte unpersistently in einem Key-Value-Store um sie zu cachieren. Zum Cachen werden Messwerte in regelmäßigen Abständen von der Smartphone-App an die Kontrollanwendung übermittelt. Bei Eingang einer Sensorwert-Anfrage der Bibliothek sendet die Kontrollanwendung den zuletzt von der Smartphone-App übermittelten Wert an die Bibliothek zurück.

Zur Übermittlung der Nachrichten werden die Protokolle MQTT und UDP eingesetzt. MQTT ist ein auf einem Observer-Pattern basierendes Client-Server-Protokoll. Durch einen 2 Byte großen Header und maximalen Payload-Größe von 260 MB ist es leichtgewichtig und gleichzeitig flexibel. Nachrichten werden zur Publizierung auf einem Topic an einen MQTT Broker gesendet. Dieser leitet sie dann an alle Clients weiter die das Topic abonniert haben. Die UDP-Kommunikation zwischen Bibliothek und Kontrollprogramm auf dem lokalen PC wird über ein Loopback-Interface umgesetzt um die Latenzzeiten zwischen Programmierumgebung und Kontrollprogramm zu minimieren. Als Nachrichtenformat wird [JSON](#) verwendet. Das menschlesbare, kompakte Nachrichtenformat ist weit verbreitet und wird von vielen Softwarebibliotheken unterstützt.

Kapitel 4

Nachrichtenformate

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 außerdem 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, wo 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 Addressierung der Messwerte im Key-Value-Speicher. Die Kürzel sind in Tabelle 4.2

aufgelistet. Ist für den Sensortyp ein Sensormesswert im Key-Value-Store vorhanden,

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

Tabelle 4.2: Sensor-Kürzel mit Beschreibung

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

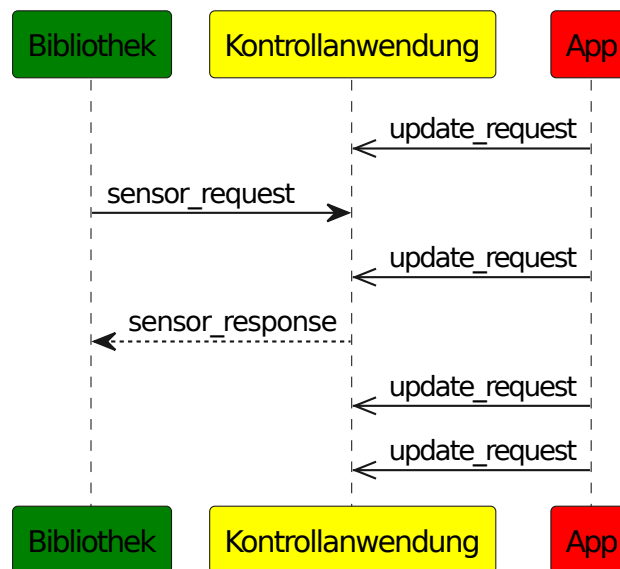


Abbildung 4.1: Nachrichtenablauf der Sensordatenübermittlung

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 vom Smartphone 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

Appseitigen Ausgaben sind unterscheidbar in Ausgaben mit und ohne Rückgabewert. Für erstere gibt es den Nachrichtentyp *rpc_request*.

RPC (Remote Procedure Call) und 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 Remote Method Invocation (**RMI**).

Der Nachrichtentyp *rpc_request* enthält die Felder *command* und *value*. Diese spezifizieren die Ausgabe und falls vorgesehen eine mit der Ausgabe verbundene Größe. 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

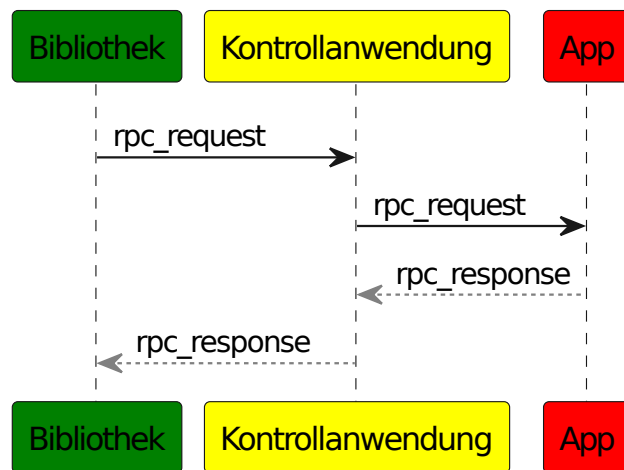


Abbildung 4.2: Nachrichtenablauf der RPC-Anfragen

drei Komponenten Bibliothek, Kontrollprogramm 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 eventuell anfallender 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

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. Für diese bietet Sie 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 SensorEventListenern die Sensormessdaten in periodischen Zeitabständen an die Kontrollanwendung. Verwendete Sensoren sind beispielsweise der 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.

5.1 Startvorgang

Ein Ablaufplan des Startvorgangs ist in Abbildung [5.1](#) zu sehen. In der RootActivity werden anfänglich alle UI-Elemente zur programmatischen Ansteuerung eingebunden. Anschließend werden die zwei Konfigurationsdaten config.json und protocol.json eingelesen. In Ersterer sind grundlegende Konfigurationsparameter wie Hostname und Port des MQTT-Brokers oder der Name des Topics definiert. In der Datei protocol.json werden neben den Schlüsselwörtern und Abkürzungen für Sensortypen und Ausgabekommandos auch Vorlagen für Nachrichtenformate aufgeführt.

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 der weitere Ablauf der Anwendung definiert, der erst ausgeführt werden soll, wenn eine MQTT-Verbindung hergestellt ist. RootActivity und MQTT-Service tauschen jeweils ihre

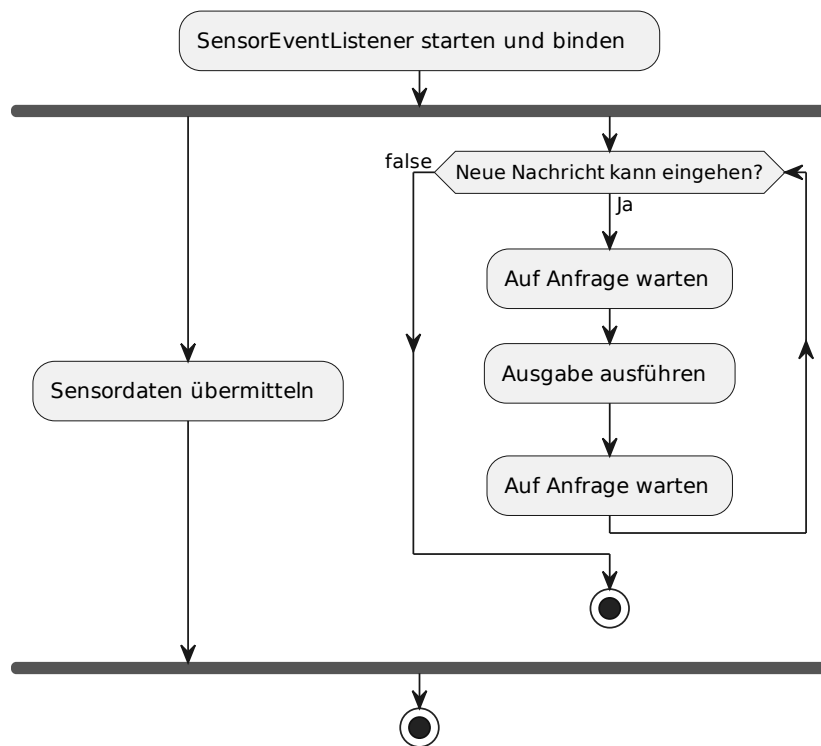


Abbildung 5.1: Ablaufdiagramm Android Anwendung

Objekt-Referenzen aus. Methoden des MQTT-Service rufen zum Zweck der Ausgabe Methoden der RootActivity auf. SensorEventListener benötigen eine Referenz auf den MQTT-Service, da sie bei Sensorwerten über diesen `update_requests` versenden.

Der Nachrichtenempfang im MQTT-Service wird nicht im UI-Thread behandelt. Dies ist jedoch Voraussetzung um UI-Elemente wie die Signal-LED im Service verändern zu können. Android unterbindet Änderungen der UI-Elemente wenn der verändernde Thread nicht der UI-Thread ist. Dieses Problem wird durch die Methode `runOnUiThread` umgangen, welche die Änderung in der Ausführungswarteschlange des UI-Threads einreicht.

Der MQTT-Service baut eine Verbindung zu einem in 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 Sensormeessung wird in Abbildung 5.2 dargestellt. SensorEventListener werden,

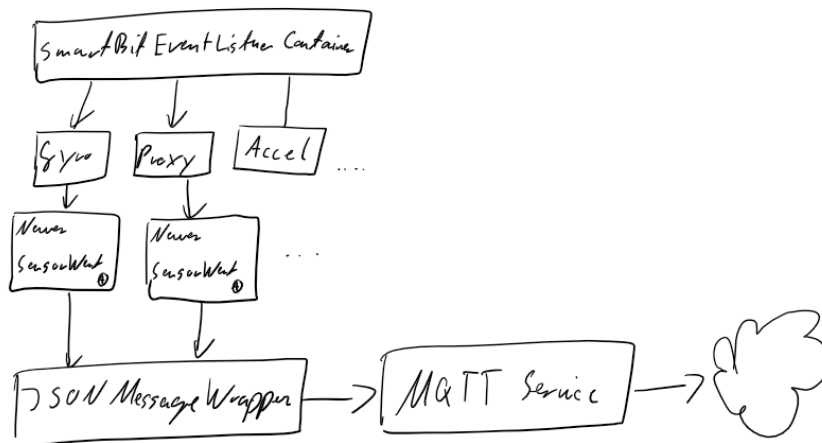


Abbildung 5.2: Ablauf SensorEventListener

nach Prüfung der Verfügbarkeit der jeweiligen Sensoren, gestartet und zentral in einem `SmartBitEventListenerContainer` gespeichert. Diese Klasse beinhaltet `SensorEventListener` für alle unterstützten Sensoren und dient der Datenhaltung der einzelnen `SensorEventListener`. Aufgabe der `SensorEventListener` ist es auf Sensorwert-Änderungen zu reagieren und eine entsprechende Callback-Funktion 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 korrekte Feld ein. Die so generierte Nachricht wird anschließend über den gebundenen MQTT-Service an das vorher definierte Topic versendet.

Anschließend 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-Stufe 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.

5.2 Sensoren

Smartphones beinhalten Sensoren, die Daten über die Umgebungseigenschaften erfassen. Dazu zählen beispielsweise Bewegung, Annäherung, aber auch Temperatur oder Luftdruck. Ihr Zweck besteht darin auf Änderungen der Werte zu reagieren. Für unterschiedliche Aufgaben werden unterschiedliche Sensoren benötigt. Beispielsweise wird für *Diebstahl-Alarm* nur der Näherungssensor verwendet, für *Dreh-Zähler* der Lagesensor. Insgesamt werden in der Android Anwendung werden folgende Sensortypen verwendet: Lineare Beschleunigungssensoren, 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. Die Erdbeschleunigung ist auch in diesen Messwerten enthalten. Diese muss für die bereinigten, realen Werte von den aufgenommenen Werten subtrahiert werden[6]. Messeinheiten unterscheiden sich je nach Sensor. Das Gyroskop misst keine Beschleunigung sondern die aktuelle Geschwindigkeit in rad/s der gleichen Achsen. Zur Übersicht sind diese in Abbildung 5.3 zu dargestellt. Die Frequenz mit der Messwerte erfasst werden kann manuell angegeben werden. Hierfür stehen vier Stufen zur Auswahl. In der

Bezeichnung	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 5.1: Sensor-Taktgeschwindigkeiten[7]

Android-App erfolgen alle Messungen mit `SENSOR_DELAY_NORMAL`. Die Stufe

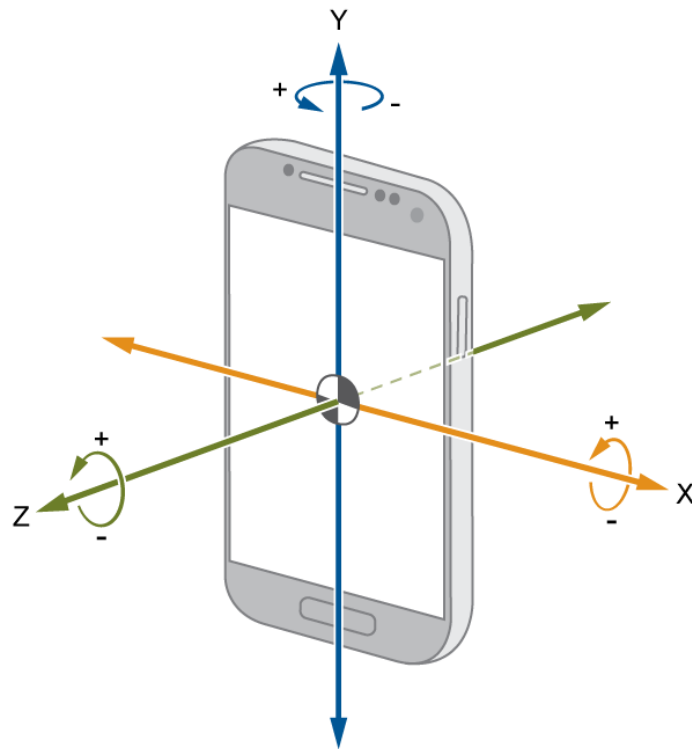


Abbildung 5.3: Android-Koordinatensystem

gilt für Android jedoch nicht als festes Limit, sondern wird eher als Richt-Frequenz behandelt. Android kann die reale Frequenz auch erhöhen. Nicht alle Smartphones besitzen alle Sensoren. Daher wird beim Start überprüft ob der Sensor auch wirklich vorhanden ist. Ist er es nicht, wird auch keine Messung gestartet.

Kapitel 6

Kontrollanwendung

Zur Vermittlung zwischen den Komponenten fungiert als Middleware eine Kontrollanwendung. Sie ist in Python geschrieben und vermittelt zwischen UDP-Anfragen auf der einen Seite vom Client aus und MQTT-Anfragen vom Smartphone auf der anderen Seite. User können über die Library RPC-Anfragen oder Sensor-Anfragen an den Server stellen. Dies geschieht in Form von JSON-Anfragen die per UDP übermittelt werden. Der Server ist unter der localhost-Adresse 127.0.0.1 auf dem Port 5006 erreichbar. Für die MQTT-Verbindung kommt dabei die unter OpenSource-Liznenz stehende MQTT-Library Paho [4] der Eclipse-Foundation zum Einsatz.

Die Kontrollanwendung ist aufgeteilt in einen Datenverwaltungsteil, DataHandler, und eine MQTT Anbindung, MqttHandlerThread. Der zur Sensordatenpufferung dienende Key-Value-Store wird durch die Klasse SensorDB implementiert. Diese Wrapper-Klasse bietet threadsichere Zugriffsfunktionen auf ein intern gespeichertes Python-Dictionary. Eine Übersicht über die Komponenten der Kontrollanwendung ist in Abbildung 6.1 dargestellt. DataHandler wiederrum teilt sich nochmal auf in

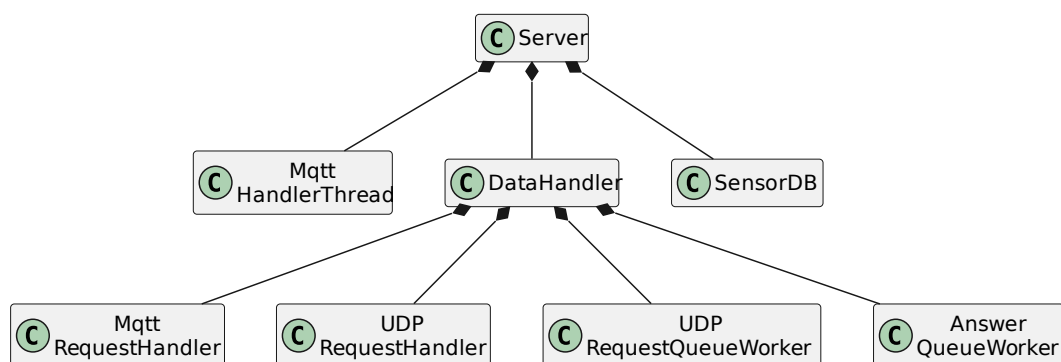


Abbildung 6.1: UML Digaramm Server

vier seperate Funktionen, die als Threads nebenläufig laufen: MqttRequestHandler,

UDPRequestHandler, UDPRequestQueueWorker und AnswerQueueWorker.

Die Funktionsweise und Zwecke dieser Threads wird im Folgenden an zwei Beispielen erläutert. Für das erste Beispiel wird Abbildung 6.2 betrachtet. Dargestellt

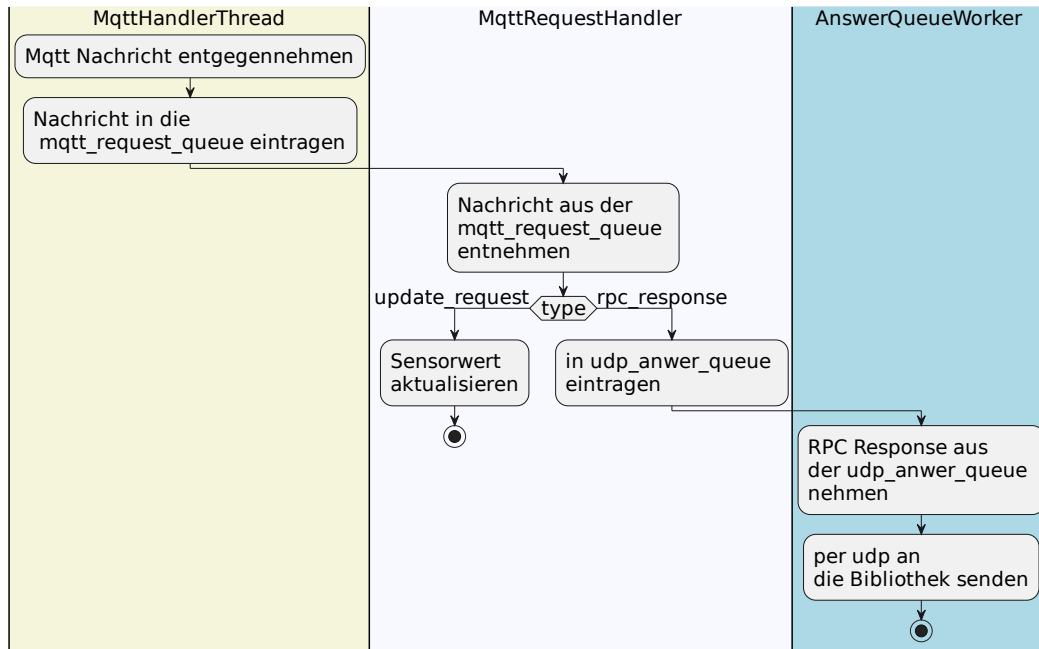


Abbildung 6.2: Ablaufdiagramm MQTT Request

ist ein MQTT-Request, also eine Anfrage des Smartphones, dass über MQTT an den Server gesendet wird. Erreicht ein MQTT Request den Server wird es im MQTT-HandlerThread entgegengenommen. Dieser setzt die Nachricht in eine MQTTRequestQueue ein. Der MQTTRequestHandler des DataHandlers wartet bis ein Eintrag in der Queue vorhanden ist und nimmt gegebenenfalls eine Nachricht. Daraufhin wird der Typ des Requests bestimmt. Handelt sich um ein Sensorupdate muss nur der Sensorwert in der Datenbank aktualisiert werden. Handelt es sich um eine rpc_response, also um eine Antwort auf eine vorausgegangenes rpc_request, dass einen Rückgabewert fordert, wird das request in eine udp_anwer_queue eingefügt. Der AnswerQueue Worker wartet, ähnlich wie der MQTT Request Handler, bis eine neue Nachricht vorhanden ist die per UDP an den Client gesendet werden soll und sendet diese dann gegebenenfalls ab.

Das zweite Beispiel befasst sich mit dem Ablauf eines UDP-Requests, also einer Anfrage die mithilfe der Bibliothek gesendet wurde. Der Ablauf ist in Abbildung 6.3 dargestellt. Erreicht ein UDP Request den Server wird es vom UDPRequestQueue-Worker in eine UDP Request Queue gelegt. Der UDPRequestHandler-Thread entnimmt die Nachricht und bestimmt den Anfragentyp. Handelt es sich um eine Anfrage des

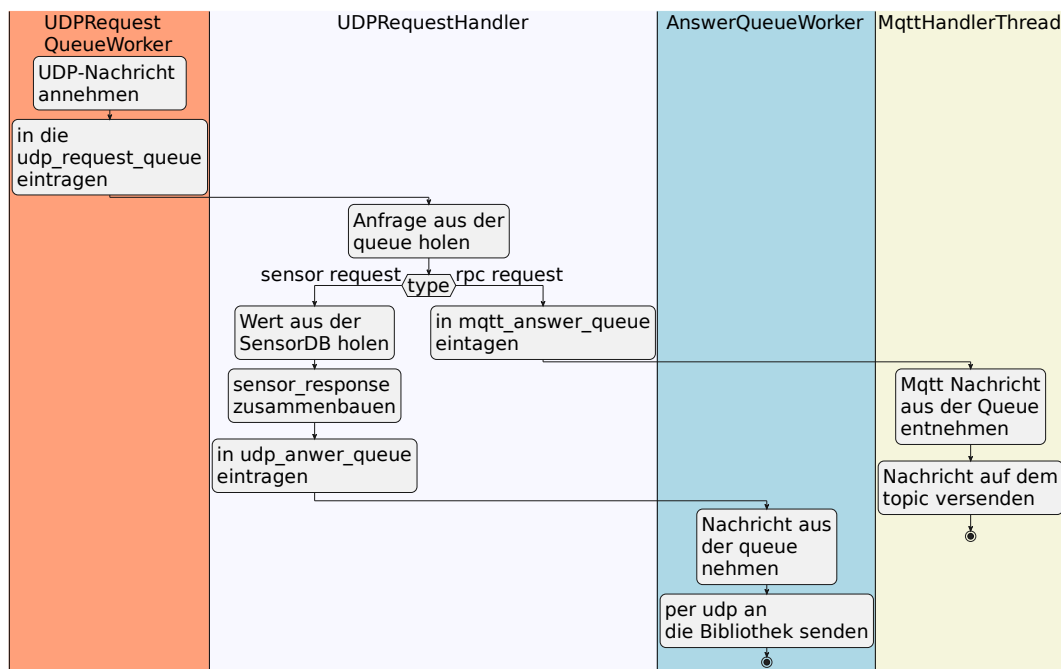


Abbildung 6.3: Ablaufdiagramm UDP Request

Types `RPC_Request` soll die Aktionen auf dem Smartphone auslösen. Sie muss an das Smartphone gesendet werden, was über MQTT möglich ist. Dafür wird Sie in eine `MqttAnswerQueue` eingesetzt. Der `MQTTHandlerThread` entnimmt Sie und sendet Sie per MQTT ab. Ist Request hingegen ein `SensorRequest`, also eine Anfrage auf die ein Sensorwert geantwortet werden soll, wird der nachgefragte Sensorwert über die Klasse `SensorDB` entnommen und in die `Udp_answer_queue` eingetragen. Der `AnswerQueueWorker` entnimmt die Anfrage und sendet Sie per UDP an die Library zurück.

Zusammenfassend erfüllen die Komponenten folgende Aufgaben. Der `MQTTHandlerThread` nimmt Nachrichten direkt per MQTT an und gibt die Anfrage weiter. Außerdem sendet er Nachrichten per MQTT, falls welche anfallen. Der `MQTTRequestHandler` kümmert sich um das Verfahren von MQTT Requests. Der `UDPRequestQueue Worker` nimmt wie der `MQTTHandlerThread` Anfragen die per UDP übermittelt wurden an und gibt Sie zur Behandlung entsprechend weiter. Er sendet jedoch im Gegensatz keine Responses zurück. Hierfür gibt es den `AnswerQueueWorker`, dessen einzige Aufgabe es ist Antworten per UDP zurück zu übermitteln.

Die Kommunikation zwischen den Threads funktioniert über synchronisierte Queues des `queue-Moduls`[9] der `cpython` Implementierung. Es handelt sich um eine threadsichere Monitorklasse, die einen gleichzeitigen Zugriff zweier unterschiedlicher Threads durch Locks verhindert.

Kapitel 7

Programmierungsumgebung

Die Programmierungsumgebung ist die Schnittstelle, die die Programmierer für die Interaktion mit dem Smartphone in ihren Programmen verwenden. Sie besteht aus einer Bibliothek, die Funktionen anbietet mit denen Programmierer Sensorwerte einlesen, oder Ausgaben auf dem Smartphone tätigen können. Sie können die Funktionen in ihren bestehenden Quellcode einbinden und die Funktionen dort verwenden. Die Bibliothek ist in den Programmiersprachen C, Java und Python vorhanden, damit Sie mit verschiedenen Programmiersprachen genutzt werden kann. In Java und Python ist Sie zudem Plattformunabhängig. Für C 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.

Werden die bereitgestellten Funktionen aufgerufen, 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 ist in Abbildung 7.1 dargestellt.

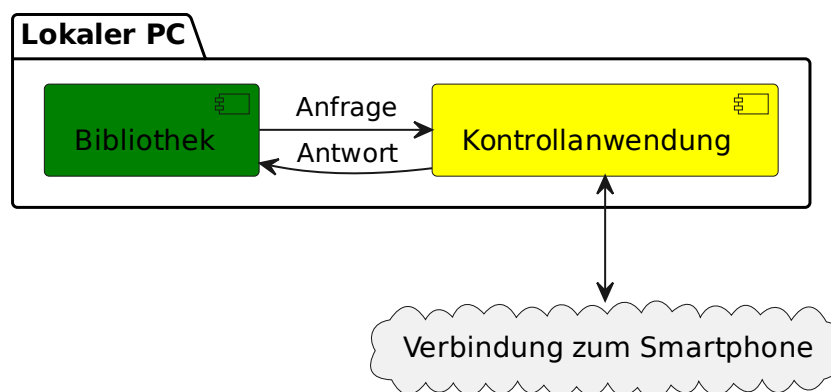


Abbildung 7.1: Schnittstellen der Bibliothek

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. 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. Die Latenzen sind dadurch mit weniger als 1 ms zur Verwendung der Lösung gering genug. 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 Nachrichtenvorlagen 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 Sie vom Programmierer einmal zum Start des Programms als cstring eingelesen werden. Anschließend muss dieser cstring 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 Dateiinhalt einzulesen. Zurückgegeben wird der Inhalt als cstring. Diese Lösung verringert die Anzahl der Lesevorgänge und die damit verbundenen durch IO verursachten Latenzen. Der Inhalt ist ab dem Einlesezeitpunkt auf dem Heap des Arbeitsspeichers gespeichert. Der Programmierer muss diesen am Ende seines Programms durch den Aufruf der `free`-Funktion wieder freigeben. Die Ausgabe erscheint als direkter Grund der Annäherung.

Kapitel 8

Evaluation

In diesem Kapitel wird überprüft, ob die implementierte Lösung die Anforderungen erfüllt. Überprüft wird die qualitative Nutzbarkeit der Lösung anhand der spezifizierten Anforderungen. Für verschiedene Nutzungsszenarien wie der Labor- oder Heimarbeit werden Latenzverhalten gemessen.

In diesem Kapitel wird die Verwendung des Frameworks anhand dem Beispiel *Alarmanlage* vorgestellt.

8.1 Verwendungsbeispiel

Gezeigt wird, wie die Aufgabe in der Programmierumgebung gelöst wurde und wie sich die Aufgabe auf dem Smartphone äußert. In der Programmierumgebung wird dabei die Python-Bibliothek verwendet.

Damit Nachrichten ausgetauscht werden können muss zuerst das Kontrollprogramm gestartet werden. Es wird mit dem Befehl `python ./server.py` in einer lokalen Shell gestartet. Der Vorgang wird in Abbildung 8.1 dargestellt. Es handelt sich um das Python-Skript *server.py*. Die Anwendung meldet, dass Sie sich erfolgreich

```
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

mit dem MQTT-Broker verbunden und das korrekte Topic abonniert hat. Die Kontrollanwendung loggt neben diesen Informationen auch rpc-requests und responses. Sensor-requests und responses werden wegen ihrer hohen Anzahl nicht geloggt.

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

```
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 importiert. Dafür muss die Datei `smartbit.py` im gleichen Ordner wie das Programm gespeichert sein. In Zeile 4 wird ein `Phone`-Objekt erstellt, über das Sensor-Auslesemethoden wie `get_x_accel()` oder Smartphone-Ausgaben wie `vibrate()` aufgerufen werden können. Da das Programm nie abbrechen soll, außer wenn es in der shell gestoppt wird, wird in einer `While-True` Schleife der Näherungssensor immer wieder abgefragt. Anschließend wird eine halbe Sekunde gewartet um den Server nicht mit Anfragen zu überlasten. Ist der abgefragte Näherungssensorwert wie in der achten Zeile überprüft wird 0.0, dann liegt eine Näherung vor. Auf dem Smartphone wird dann mit der Methode `write_text()` der Text `ALARM` ausgegeben. Das Smartphone vibriert 5 mal hintereinander im Abstand von 200 ms für eine Sekunde und lässt die Signal-LED blinken. Insgesamt werden die RPC-Funktionen `vibrate`, `toggle_button` und `write_text` aufgerufen. Keine dieser Funktionen liefert einen Rückgabewert.

Die Bibliothek sendet die Nachrichten an das Smartphone. Nähert sich eine Person dem Smartphone wird der Code in der If-Bedingung ausgeführt. Die jeweiligen Nachrichten werden als `rpc_request` per UDP an die Kontrollanwendung gesendet. Dort werden Sie angenommen, geloggt und per MQTT an das Smartphone weitergereicht. Eine Übersicht der gesendeten Nachrichten ist in Abbildung 8.2 zu sehen. Die Nachrichten werden im JSON Format übertragen. `rpc_requests` haben ein Typ-Feld, ein Kommando und einen Parameterwert für das Kommando. `Write_text` ist für Textausgaben auf dem Smartphone, `vibrate` um das Smartphone vibrieren zu lassen und `led_toggle` um die LED anzusprechen. Für `write_text` wird der gibt der

```
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

Parameter-Wert den anzuzeigenden Text an. Vibrate erwartet für value die Zeitdauer der vibrierung in ms. led_toggle erwartet keinen Parameterwert, da der gesamte Aufruf lediglich die Farbe der LED wechselt. Es ist nicht vorgesehen, dass die Farbe manuell gesetzt werden kann.

Auf dem Smartphone ist die App zu Anfang im Initialmodus. Dieser ist in [Abbildung 8.3](#) dargestellt. Zu sehen sind zwei Buttons, beschriftet mit A und B. Das textfeld

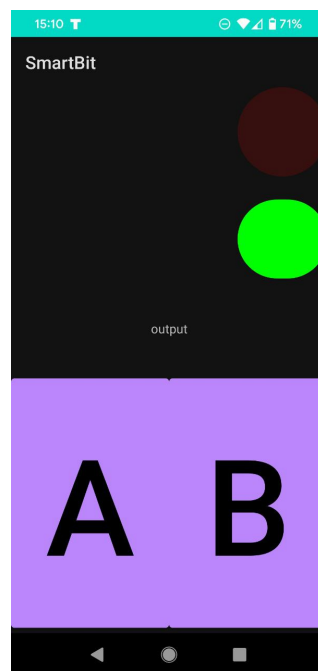


Abbildung 8.3: Initialzustand der Anwendung

liegt in der Mitte und zeigt zum Start den Text *output* an. Daneben gibt es noch eine Vorgangs-LED die ausgegraut oben rechts über der Signal-LED liegt. Sie leuchtet, wenn gerade ein Vorgang wie das Ausführen einer Vibration auf dem Smartphone

ausgeführt wird.

Nähert man sich dem Smartphone geht es über in den Alarmzustand der in Abbildung 8.4 dargestellt ist. Erkennbar ist, dass das Textfeld den Wert *ALARM* darstellt.

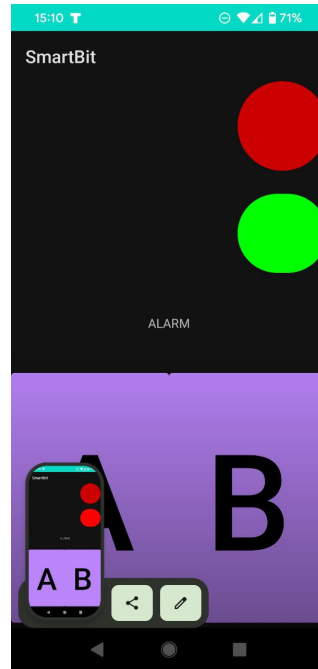


Abbildung 8.4: Initialzustand der Anwendung

Die Farbe der LED hat sich von grün auf rot und wieder auf grün geändert. Das ist erkennbar an der kleinen Screenshot-Vorschau links unten die unmittelbar vor dem in der Abbildung zu sehenden Screenshot zu sehen ist. Die Vorgangs-LED leuchtet rot um zu signalisieren dass gerade eine Ausgabe ausgeführt wird. Nicht sichtbar ist das haptische vibrationsfeedback. Hierfür dient jedoch die Vorgangs-LED. Sie ist auch hilfreich um bei Smartphones mit sanften Vibrationsausgaben eine Ausgabe zu erkennen. Die Reaktionszeit liegt unter einer Sekunde.

8.2 Latenzmessung

Transportwege müssen berücksichtigt werden. Die Zeiten werden in drei verschiedenen Umgebungen überprüft: In nächster Nähe zum MQTT-Server, in mittlerer Nähe und in einer virtuellen Umgebung über ein simuliertes NAT-Netzwerk.

Damit die Benutzbarkeit sichergestellt ist, müssen Latenzzeiten zwischen Smartphone und Programmierumgebung in einem akzeptablen Rahmen liegen. Um dies herauszufinden werden die Latenzzeiten gemessen. In diesem Kapitel wird die Latenzzeit

von drei Standorten aus gemessen.

Der erste Einsatzort ist im gleichen Netzbereich wie der MQTT-Broker wodurch die Transferstrecke reduziert wird. Neben der Latenzzeit zum Broker ist jedoch auch die Anzahl der angemeldeten Geräte am gleichen AccessPoint interessant. Da die Geräte in direktem Wettbewerb um Sendezeit stehen bedeuten mehr Geräte an einem AccessPoint größere Latenzen. In den Messungen werden nur WLAN-Verbindungen nach dem Standard WLAN 802.11n berücksichtigt. Es sind dabei stets beide Geräte, Lokaler PC und Smartphone per WLAN mit dem Internet verbunden. Untersucht wird lediglich die Zeitspanne zwischen `rpc_request` und `rpc_response`. Sensor_requests kommunizieren stets per loopback-interface mit der Kontrollanwendung. Daher die Latenzzeiten gering. Im Durchschnitt belaufen Sie sich auf unter 1ms. Die Messungen sind für den Fall interessant, für den Studierende an der Hochschule die Lösung nutzen möchten. Zum Beispiel in Laboren.

Der zweite Einsatzort ist ein Heimnetz, das etwa 8 Hops entfernt vom MQTT-Broker liegt. Etwa 8 Geräte sind dabei auf dem Access Point angemeldet. Die Messergebnisse sind relevant für die Heimarbeit oder das Erledigen von Hausaufgaben. Die Ergebnisse sind in Abbildung 8.5 dargestellt. Insgesamt wurden 100 Messungen

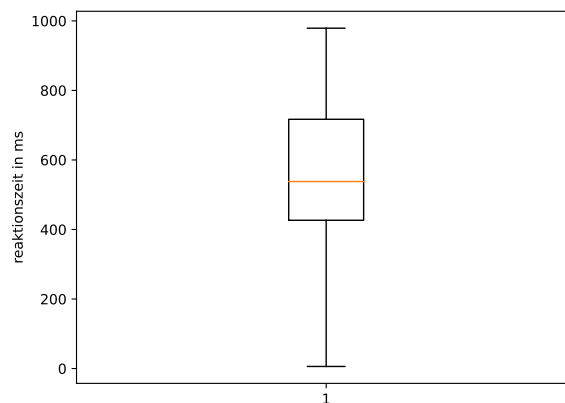


Abbildung 8.5: Zeitmessungen im Heimnetz

durchgeführt. Im Mittel beträgt die Reaktionszeit 537 ms. Die Streuung ist bei einer Standardabweichung von 238ms allerdings auch sehr breit gestreut. Alle Messungen blieben jedoch unter 1s Reaktionszeit.

Der letzte Einsatzort ist in einer virtuellen Maschine auf dem lokalen PC das über den virtuellen Netzwerkkontroller aus einem NAT-Netzbereich auf das Internet zugreift. Das Smartphone befindet sich direkt im Netzbereich des lokalen PCs. Die gemessenen Daten sind in Abbildung 8.6 dargestellt. Wieder wurden 100 Messungen

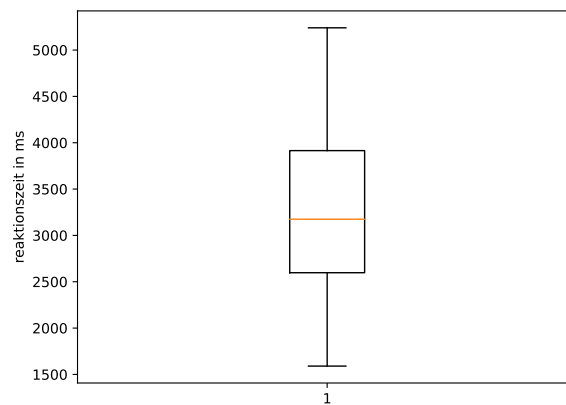


Abbildung 8.6: Zeitmessungen in der virtuellen Maschine

vorgenommen. Im Mittel beträgt die Reaktionszeit nun 3255 ms. Die Streuung ist mit einer Standardabweichung von 845 ms relativ gleichgroß. Die Reaktionszeit ist also im Durchschnitt sehr hoch, etwa sechs mal langsamer als bei einer Verwendung der Programmierumgebung auf dem Hostsystem. Eine direkte Assoziation zwischen Eingabe und Ausgabe kann nicht garantiert werden.

Latenzprobleme per MQTT treten nicht auf. Trotz einem QOS-Level 0 werden auch Ausgabeanfragen sicher übertragen und ausgeführt. Da die Nachrichten über TLS übertragen werden ist der Austausch sicher. Logging-Möglichkeiten in der Kontrollanwendung und der Android-App erleichterten die Entwicklung sehr und halfen bei der Fehlersuche.

Kapitel 9

Fazit

Die implementierte Lösung erfüllt die Anforderungen zufriedenstellend. Die im Vorfeld durchgeführte Konzeption der Lösung ermöglichte eine problemlose Implementierung. Während der Entwicklung wurden jedoch einige Designentscheidungen getroffen, welche die 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-Schnittstelle könnte dies unintuitiv erscheinen. 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 gemacht werden. 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 Datenspeicher 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 unintuitiven 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 Sensmessdatenübertragung erwägenswert um den Energieverbrauch zu reduzieren.

Abbildungsverzeichnis

3.1	System-Aufbau	9
4.1	Nachrichtenablauf der Sensordatenübermittlung	12
4.2	Nachrichtenablauf der RPC-Anfragen	13
5.1	Ablaufdiagramm Android Anwendung	16
5.2	Ablauf SensorEventListener	17
5.3	Android-Koordinatensystem	19
6.1	UML Digaramm Server	20
6.2	Ablaufdiagramm MQTT Request	21
6.3	Ablaufdiagramm UDP Request	22
7.1	Schnittstellen der Bibliothek	23
8.1	Start der Kontrollanwendung	25
8.2	Nachrichtenversand der Kontrollanwendung	27
8.3	Initialzustand der Anwendung	27
8.4	Initialzustand der Anwendung	28
8.5	Zeitmessungen im Heimnetz	29
8.6	Zeitmessungen in der virtuellen Maschine	30

Tabellenverzeichnis

2.1	Beispielprogrammieraufgaben	6
4.1	Nachrichten-Typen	11
4.2	Sensor-Kürzel mit Beschreibung	12
5.1	Sensor-Taktgeschwindigkeiten[7]	18

Abkürzungsverzeichnis

API Application Programming Interface. [9](#)

CSMA/CA Carrier Sense Multiple Access/Collision Avoidance. [7](#)

IDE Integrated Development Environment. [8](#)

JSON JavaScript Object Notation. [10](#)

MQTT Message Queuing Telemetry Transport. [10](#)

QoS Quality of Service. [18](#)

RMI Remote Method Invocation. [13](#)

RPC Remote Procedure Call. [13](#)

RTT Round Trip Time. [7](#)

UDP User Datagram Protocol. [10](#)

UMTS Universal Mobile Telecommunications System. [7](#)

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-arduini official store. <https://store.arduino.cc/collections/kits>. [letzter Zugriff: 08. Juni. 2022].
- [4] Eclipse. Paho. <https://www.eclipse.org/paho/>. [letzter Zugriff: 03. Juni. 2022].
- [5] Financesonline. Number of smartphone users worldwide 2022. <https://financesonline.com/number-of-smartphone-users-worldwide/>. [letzter Zugriff: 08. Juni. 2022].
- [6] Google. Android developers - accelerometer example. https://developer.android.com/guide/topics/sensors/sensors_motion#sensors-motion-accel. [letzter Zugriff: 04. Juni. 2022].
- [7] Google. Sensors - overview. https://developer.android.com/guide/topics/sensors/sensors_overview#sensors-monitor. [letzter Zugriff: 22. Juni. 2022].
- [8] bitkom. <https://www.bitkom.org/Presse/Presseinformation/Mit-10-Jahren-haben-die-meisten-Kinder-ein-eigenes-Smartphone>. [letzter Zugriff: 07. April. 2022].
- [9] python. queue. <https://docs.python.org/3/library/queue.html>. [letzter Zugriff: 03. Juni. 2022].