Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 3

Integration modellbasierter Erfassungsmethoden in ein Public-Sensing-Testbett

Patrick Alt

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer:	Dipl.-Inf. Damian Philipp
begonnen am:	15. März 2012
beendet am:	14. September 2012
CR-Klassifikation:	C.1.4, C.2.1, C.2.4

Kurzfassung

Mit Public Sensing können große Gebiete unserer Umgebung durch mobile Sensoren in Smartphones, die sich mit ihren Besitzern bewegen, abgedeckt werden. Dadurch wird es möglich, Daten über unser Umfeld, wie zum Beispiel die Heimatstadt, zu sammeln, ohne dass aufwendige Sensornetze installiert werden müssen. Die Daten können durch Lokalisierungsdienste wie GPS dabei direkt mit einer Position verknüpft werden.

Ein großer Nachteil ist, dass durch die ständige Verwendung von Sensoren und GPS die Akkus der Smartphones sehr schnell zur Neige gehen. Ein weiteres Problem ist, dass wegen der unkontrollierbaren Mobilität der Smartphones durch ihre Besitzer nie sichergestellt werden kann, dass bestimmte Bereiche durch Sensoraufnahmen abgedeckt sind.

Modellbasierte Erfassungsmethoden versuchen diesen beiden Punkten entgegenzuwirken, indem fehlende Daten berechnet werden und die Anzahl benötigter Messwerte sogar aktiv eingeschränkt wird. Ziel dieser Arbeit ist es, diese modellbasierten Erfassungsmethoden in ein Realwelt-Testbett für Public-Sensing-Systeme zu integrieren, damit sie in einer realen Umgebung validiert werden können.

Um die modellbasierten Erfassungsmethoden im kleinen Rahmen vorführen zu können, wird außerdem das Testbett so erweitert, dass es optional verkleinert auf einem Tisch ausgeführt werden kann.

Abstract

With Public Sensing it is possible to cover large areas with many mobile sensors in smartphones that are carried by their owners. This way it is possible to gather much data about our environment, e.g. our home town, without having to install complex sensor networks. With the help of localization services like GPS, the data can be linked to certain positions immediately, too.

One big disadvantage of this is the fact that the devices run out of power very fast while using sensors and GPS receiver continuously. Another problem is that you can never be sure that certain points of interest are covered by devices due to the uncontrollable mobility of the devices with their owners.

Model driven data acquisition tries to resolve both of these problems by calculating missing values and actively reducing the amount of data needed. Goal of this work is to integrate this model driven approach into a Public Sensing testbed for being able to validate it in a real-world environment.

For presenting the model driven data acquisition on a small scale, the testbed will be extended so that in can be used in a very small area like a desk, for example.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In den letzten Jahren wurden Smartphones immer beliebter und sind heute aus dem Alltag vieler Menschen nicht mehr wegzudenken. Da diese Geräte im Vergleich zu gewöhnlichen Mobiltelefonen oft noch ein breites Spektrum an Sensoren besitzen (z. B. Bewegungs-, Lage-, Magnetfeld-, Licht- und Annäherungssensoren sowie Möglichkeiten zur Positionierung), entstand die Idee des sogenannten *Public Sensing*: Die große Verbreitung und Beweglichkeit der Smartphones und damit auch der Sensoren soll ausgenutzt werden, um teure statisch angebrachte Sensornetzwerke zu ersetzen oder zu ergänzen. So ist es zum Beispiel möglich, in verschiedenen Stadtteilen den Lärmpegel zu messen, ohne Mikrophone verteilen zu müssen. Auch im persönlichen Bereich ist Public Sensing von großem Nutzen: Beispielsweise können Helligkeitsmessungen dazu beitragen, den Benutzer zu warnen, wenn er sich zu lange den UV-Strahlen der Sonne aussetzt. Durch die Anwendung BikeNet [EML+07] können Radfahrer automatisch ihre Route und Geschwindigkeit aufzeichnen lassen.

Da für Public Sensing die Sensoren und auch WLAN oder das mobile Internet (für den Datenaustausch) der Smartphones ständig aktiviert sein müssen und die in den Akkus zur Verfügung stehende Energie dadurch für keinen vollen Tag mehr ausreicht, ist dieser Ansatz momentan nur eingeschränkt einsetzbar. Das liegt daran, dass die Menschen zumeist nur dann die Möglichkeit haben, den Akku zu laden, während sie schlafen. Ein weiteres Problem ist, dass durch die nicht kontrollierbare Mobilität der Smartphones nie sichergestellt werden kann, dass gewünschte Bereiche, von denen man sich Messungen erhofft, auch von den Geräten ausreichend abgedeckt werden. Derzeitige Ansätze in der Forschung versuchen durch modellbasierte Erfassungsmethoden diesen beiden Problemen entgegenzuwirken.

Durch ein Modell der Daten können fehlende Messwerte mit Hilfe vorhandener Werte nachträglich berechnet werden. Es geht dabei davon aus, dass vorher bestimmte Messpunkte spezifiziert worden sind, an denen Messungen durchgeführt werden sollen. Durch den gleichen Ansatz kann man auch aktiv die Anzahl ausgewählter Messpunkte reduzieren, da die Werte der anderen Punkte dann berechnet werden können. Damit müssen Sensoren seltener eingeschaltet werden und es sind weniger Datenübertragungen nötig, wodurch der Energieverbrauch eingeschränkt wird. Ein System zur Überprüfung der Validität des Modells stellt dabei kontinuierlich sicher, dass spezifizierte Qualitätsmerkmale eingehalten werden.

Bisher wurden die Methoden nur in Simulationen getestet und weiterentwickelt. Ziel dieser Arbeit ist es deshalb, die modellbasierten Erfassungsmethoden und die Überprüfungsmechanismen in ein bestehendes Realwelt-Testbett für Public-Sensing-Systeme zu integrieren, um sie damit in einer realen Umgebung evaluieren zu können. Außerdem soll ein Demonstrator

entwickelt werden, der das System so weit verkleinert, dass es auf einem Tisch ausgeführt werden kann, um es im kleinen Rahmen vorführen zu können.

Dazu werden zunächst in Kapitel 2 Public Sensing und die modellbasierten Erfassungsmethoden sowie das existierende Realwelt-Testbett genauer erklärt, bevor in Kapitel 3 das zugrundeliegende Systemmodell erläutert wird. Kapitel 4 und 5 widmen sich dem Entwurf und der Implementierung der modellbasierten Erfassungsmethoden in das bestehende Realwelt-Testbett. Nach der Evaluierung in Kapitel 6 schließt die Arbeit mit einer Zusammenfassung und einem Ausblick in Kapitel 7.

2. Grundlagen & verwandte Arbeiten

In diesem Kapitel werden Public Sensing und damit zusammenhängende Arbeiten ausführlicher beschrieben. Außerdem werden die dieser Arbeit zugrunde liegenden modellgetriebenen Erfassungsmethoden motiviert und vorgestellt sowie das verwendete Realwelt-Testbett erklärt.

2.1. Public Sensing

Public Sensing ist ein noch sehr junges Paradigma in der Forschung und es gibt dementsprechend auch noch viele Namen dafür: Public Sensing, Urban Sensing [CHK08], People-Centric Sensing [CEL+08], Citizen-Sensing [CHK08], Mobile Phone Sensing [LML+10] oder auch Global Sensing [LML+10]. Die Idee ist, mit Hilfe von Sensoren in Smartphones und ähnlicher Geräte wie z. B. Tablets, in unserer direkten Umgebung Daten zu sammeln. Möglich wird das, da ein sehr großer Anteil der Bevölkerung bereits heute jeden Tag ein solches Gerät bei sich trägt. Mobile Kommunikationsgeräte wie Handys und Smartphones sind so entwickelt worden, dass sie unsere Worte aufzeichnen und weiterleiten können. Moderne Smartphones und Tablets besitzen aber bereits weit mehr als das: Helligkeits-, Rotations- und Gravitationssensoren gehören zur Standardausstattung genauso wie Sensoren zur Messung von Annäherung und Beschleunigung. Manche Geräte besitzen außerdem Sensoren, um Luftdruck und relative Luftfeuchtigkeit zu messen. Zusammen mit einem GPS-Sensor, ohne den kaum ein Smartphone mehr auskommt, können die aufgezeichneten Daten mit einer Position verknüpft werden.

Damit muss für Public Sensing kein teures Sensornetzwerk mehr aufgebaut werden, das zudem statisch ist. Die Mobilität von Smartphones zusammen mit ihren Besitzern sorgt sogar dafür, dass ein bedeutend größeres Gebiet erfasst werden kann [LBD+05]. Durch WLAN und Internetverbindung ist außerdem dafür gesorgt, dass Daten auf einfache Weise mit anderen geteilt, gesammelt und weitergeleitet werden können.

Diese Daten bringen in vielen Bereichen unseres Lebens Vorteile: Wird in einer Stadt zum Beispiel regelmäßig und überall die Luftverschmutzung und der Lärm gemessen, können Familien einen möglichst ansprechenden Ort zum Leben auswählen.

Im Gegensatz zu Public Sensing werden Sensoren schon seit langem außerhalb von Städten in der Natur eingesetzt, um beispielsweise den Zustand der Lebensräume von Pflanzen und Tieren zu untersuchen und zu beobachten. Durch Thermometer in Seen und Windstärke-, CO_2- und anderen Messungen, die über einen längeren Zeitraum aufgenommen werden, können viele Rückschlüsse auf die Umgebung gezogen werden.

Genauso können jetzt bei Public Sensing Sensoren in der Bevölkerung dazu verwendet werden, unsere direkte Umgebung aufzuzeichnen. Werden die so gewonnenen Daten visualisiert, können sie der gesamten Gemeinschaft zu Gute kommen, nicht nur den Forschern und Biologen.

Auch in unserem sozialen Umfeld wird dies eine Rolle spielen. Schon heute kann man auf Plattformen wie Facebook und Twitter seine Posts mit Bildern und einem Standort versehen, um seine Freunde auf dem Laufenden zu halten. Dies ist bereits der Beginn von Public Sensing und kann erweitert werden, um sozialen Kontakten automatisiert mehr Informationen über den Aufenthaltsort, über sich selbst und seine Lebensweise zukommen zu lassen. Dies wird auch als *Social Sensing* bezeichnet [CEL+08].

Auch im privaten Bereich (*Personal Sensing*) entstehen viele Vorteile. Wenn man zum Beispiel beim Jogging abgelaufene Routen und die Geschwindigkeit aufzeichnet, kann man über längere Zeiträume den Verbrauch der Kalorien berechnen und beobachten, wie die eigene Ausdauer zunimmt [CMT+08].

Die derzeitige Forschung beschäftigt sich mit vielen Facetten dieses neuen Paradigmas; zum Beispiel muss bei der Entwicklung der passenden Software auch die Privatsphäre der Menschen beachtet werden [CKK+08]. Dabei entstanden zwei grundlegende Designs für Public-Sensing-Systeme, die je nach Anforderung aber auch miteinander vermischt werden können [CEL+08]:

Beim *mitwirkenden* (engl. *participatory*) Public Sensing entscheidet der Smartphone-Besitzer fortwährend, welche Daten aufgenommen und versendet werden sollen. Da der Besitzer dafür ständig zum Beispiel über eine GUI Eingaben tätigen muss und so bei seiner eigentlichen Arbeit unterbrochen wird, ist die Anzahl an bereitwilligen Teilnehmern am System stark eingeschränkt. Andererseits ist die Privatsphäre des Benutzers so besser geschützt, denn er kann selbst entscheiden, wann und welche Daten versendet werden sollen. Ein weiterer Vorteil ist, dass die Qualität der Daten verbessert werden kann, wenn der Besitzer auf eine Bitte hin sein Smartphone für einen Moment in eine bestimmte Position bringt, um zum Beispiel ein Bild aufnehmen zu können.

Beim *opportunistischen* Public Sensing ist dem Smartphone-Besitzer in der Regel nicht bewusst, wann Daten aufgezeichnet werden, und seine Mitwirkung ist nicht erforderlich. Der Zustand des Gerätes wird automatisch erkannt und das Aufnehmen von Daten komplett vom System gesteuert. Auf diese Weise wird der Besitzer nicht gestört und Public Sensing kann im Hintergrund ablaufen. Die Herausforderung bei diesem Ansatz ist die korrekte Erkennung der Umgebung, damit zum Beispiel nicht Helligkeitswerte aufgezeichnet werden, wenn sich das Smartphone in der Tasche befindet, und dass die Privatsphäre der Smartphone-Besitzer dennoch ausreichend geschützt wird [CEL+08].

Diese Arbeit und deren Grundlagen wenden sich dem opportunistischen Public Sensing zu, das laut großflächige Messbereiche, wie beispielsweise eine ganze Stadt, besser unterstützt [LEM+08].

Hier wird auch davon ausgegangen, dass Sensordaten nur an bestimmten vorher festgelegten Punkten aufgenommen werden, wenn Smartphones daran vorbeikommen [PDR11]. Eine

andere Realisierungsmöglichkeit ist, Smartphones zunächst unabhängig von ihrer Position Daten aufzeichnen zu lassen und sie später in Zusammenhang mit dem Aufnahmestandort zu analysieren (siehe z. B. [MSN+09]).

2.1.1. Testumgebungen und reale Systeme

Es gibt bereits viele Anwendungen, die mit Hilfe von mobilen Geräten wie Smartphones bestimmte Zwecke erfüllen. Bei *OpenStreetMap* können Geräte mit GPS-Empfänger zum Beispiel dazu beitragen, im Internet frei zugängliche Karten der ganzen Welt zu vervollständigen und zu verbessern [HW08]. Dies ist auf einfache Weise möglich, indem man sein Smartphone kontinuierlich Positionsdaten sammeln lässt, während man unterwegs ist, und diese später hochlädt. OpenStreetMap fällt dabei in die Sparte des mitwirkenden Public Sensing, da das Aufzeichnen von Positionsdaten explizit aktiviert werden muss und die Karten hauptsächlich durch sogenannte *Mapping Partys* erstellt werden, bei denen sich Gruppen von Menschen für z. B. ein Wochenende ein Gebiet aufteilen und abfahren oder ablaufen, um es zu kartographieren. Da sich nur sehr wenige Benutzer aktiv an der Erweiterung der Karten beteiligen, existieren Überlegungen, wie das Erstellen der Karten über einen opportunistischen Ansatz automatisiert werden kann [BWD11] oder wie existierende Karten auf ähnlichem Wege korrigiert und validiert werden können [BWDR11].

Das *MobGeoSen*-System wurde bereits eingesetzt, um Lärm und Kohlenstoffmonooxid-Konzentration auf den Schulwegen von Kindern zu messen [KBP+08]. Da das System dabei so ausgelegt ist, dass die Daten während der normalen Aktivitäten der Schüler gemessen werden können, kann es dem opportunistischen Public Sensing zugeschrieben werden. Ein großes Problem dieses Systems ist, dass die mobilen Geräte nur maximal 8 Stunden funktionsfähig sind, bevor ihr Akku wieder geladen werden muss.

Ähnliche Systeme, die auf die Messung von Lärmbelästigung und Luftverschmutzung spezialisiert sind, sind *NoiseTube* [MSN+09] und *PollutionSpy* [KBRL09]. Beim *Bubble-Sensing*-System [LLEC10] können zwar beliebige Sensoren verwendet werden, die Erfassung von Daten ist konzeptionell aber auf nur einen Messpunkt mit gewissem Radius, der sogenannten Bubble, festgelegt.

CenceMe ist dagegen nicht auf Messpunkte festgelegt und kann beliebige Sensordaten sammeln [MLEC07]. Das System wurde entwickelt, damit Benutzer Daten über ihre Stimmungslage, ihre Aktivitäten und ihre Umgebung mit ihren Freunden teilen können. So ist es zum Beispiel möglich, Geschwindigkeiten und Beschleunigungen aufzuzeichnen, Bilder und Audiodaten aufzunehmen, den Standort weiterzugeben oder über Plugins für Facebook und Pidgin soziale Kontakte zu importieren. Dadurch ist es mit CenceMe auch möglich, die Privatsphäre zu schützen, da der Benutzer entscheiden kann, welcher seiner Freunde welche Daten erhalten darf.

MetroSense ist ein System für opportunistisches Public Sensing in sehr großem Rahmen [CEL+06]. Der Fokus liegt dort auf der Interaktion zwischen Smartphones und anderen Geräten mit Sensoren, wodurch die Leistung und Effizienz des Systems verbessert wird.

2.2. Modellgetriebene Datenerfassungsmethoden

Ein großes Problem von Public Sensing konnte bisher noch nicht ausreichend gelöst werden: Smartphones, die kontinuierlich Sensordaten sammeln und ihre Position bestimmen müssen, benötigen deutlich mehr Energie als beim durchschnittlichen Gebrauch eines solchen Gerätes. Auch für die nötigen Datenübertragungen wird viel Energie verbraucht. Modellgetriebene Datenerfassungsmethoden versuchen nun, diesem Problem entgegenzuwirken, indem die Anzahl benötigter Messpunkte reduziert wird und so die Geräte entlastet werden.

2.2.1. Allgemeines

Modellgetriebene Erfassungsmethoden werden bereits erfolgreich für statische Sensornetzwerke eingesetzt [GKS05]. Die Idee hinter ihnen ist, dass der gewünschte Teil der Realität mathematisch formalisiert wird. Grundsätzlich ist die Realität nicht berechenbar, es können aber Annahmen über Wahrscheinlichkeiten getroffen werden. Deshalb werden Sensormesswerte als Zufallsvariablen aufgefasst, für die Daten wie Varianz und Kovarianz berechnet werden können. Vorhersagen können dann getroffen werden, wenn Korrelationen zwischen den Zufallswerten bestehen.

Für Umweltwerte wie zum Beispiel die Temperatur wird oft angenommen, dass sie mit mehrdimensionalen Gaußverteilungen modelliert werden können [GKS05]. Um ein solches Modell erstellen zu können, werden zunächst empirische Daten von allen beteiligten Sensoren benötigt, mit denen der Mittelwertsvektor und die Kovarianzmatrix der mehrdimensionalen Gaußverteilung berechnet werden [STY05]. Mit Hilfe dieses Vektors und der Matrix können danach fehlende Messwerte in Abhängigkeit von erhaltenen Werten berechnet werden.

2.2.2. Realisierung

Um ein Modell für ein bestimmtes Phänomen erstellen zu können, werden zunächst empirische Daten von allen Messpunkten benötigt. Dafür müssen die gewünschten Stellen von realen Sensoren abgedeckt werden oder man verwendet das Modell von einem ähnlichen Phänomen an anderer Stelle. Das Sammeln von empirischen Daten wird als *Lernphase* bezeichnet. In dieser Phase werden alle Messpunkte nach Daten gefragt, damit eine mehrdimensionale Gaußverteilung erstellt werden kann.

Wenn das Modell erstellt wurde, ist es möglich nur noch einen Teil der Messpunkte nach Daten zu fragen. Für die anderen werden dann die Sensorwerte nachträglich berechnet, es wird eine sogenannte *Inferenz* durchgeführt. Diese Phase wird *Optimierungsphase* genannt.

Das Vorgehen wurde nun für Public Sensing adaptiert [PSDR12]: Um ein Modell für ausgewählte Messpunkte erstellen zu können, muss das System zunächst Daten von allen Messpunkten sammeln und ist in dieser Phase darauf angewiesen, dass jeder Messpunkt von Smartphones abgedeckt wird. Werden für bestimmte Messpunkte in der Lernphase

keine Daten geliefert, sind diese Punkte später nicht Teil des Modells und können bei der Optimierung nicht berücksichtigt werden.

In der Optimierungsphase werden Sensorwerte für Messpunkte berechnet, von denen keine Daten erhalten wurden. Die Berechnung geschieht dabei auf Basis des Modells und der erhaltenen Messwerte. Es ist also möglich, die Anzahl Messpunkte, an denen Smartphones ihre Sensoren einschalten müssen, aktiv zu reduzieren und trotzdem für alle Punkte Daten zu erhalten. Auf diese Weise kann der Energieverbrauch der Smartphones stark eingeschränkt werden.

2.2.3. Sensorauswahl

Um möglichst fehlerfreie Werte zu erhalten, müssen die Messpunkte ausgewählt werden, bei denen die entstehende Varianz bei der Inferenz möglichst gering ist. Je mehr Messpunkte ausgewählt werden, desto niedriger ist die vorhergesagte Varianz. Das Problem ist also, eine Menge an Messpunkten zu wählen, für die die vorhergesagte Varianz möglichst gering ist, die Kosten zum Abfragen der Messpunkte aber ebenfalls möglichst gering ausfallen. Das Lösen solcher Probleme ist NP-schwer [GKS05], aber es wurde ein gieriger Approximationsalgorithmus dafür entwickelt [Krao8], der in angepasster Form auch für die modellbasierten Erfassungsmethoden für Public Sensing verwendet wird [PSDR12].

Die Auswahl der Sensoren wird dabei zunächst unter der Annahme getroffen, dass jeder Messpunkt von einem Smartphone abgedeckt wird, also jeder Messpunkt Daten liefern würde. Durch die unkontrollierbare Mobilität der Smartphones kann dies aber nicht sichergestellt werden. Durch die modellbasierten Erfassungsmethoden wird aber auch dieses Problem gelöst, da Werte von Messpunkten, die keine Daten liefern, trotzdem berechnet werden können. Voraussetzung dafür ist nur, dass die Messpunkte zumindest in der Lernphase von Smartphones abgedeckt wurden, da sie sonst nicht Teil des Modells sind.

2.2.4. Validitätsprüfung

Wenn die Umgebung sich ändert, zum Beispiel durch Temperaturschwankungen im Tagesverlauf, werden die berechneten Werte des Modells schlechter. Um zu verhindern, dass sie nicht einen definierten Qualitätsstandard unterschreiten, gibt es den Validitätsprüfer namens `MOCHA` (Model Validity Check Algorithm), der die Korrektheit des Modells kontinuierlich überprüft. Er wählt dafür zusätzlich zu den Sensoren, die für die Inferenz ausgewählt wurden, noch weitere Sensoren zufällig aus, deren Werte mit den berechneten Daten des Modells verglichen werden. Übersteigt der Fehler dabei einen vorgegebenen Schwellwert, wird das Modell invalidiert, sodass das System in eine neue Lernphase geht, damit das Modell angepasst werden kann.

Um nicht durch etwaige Messfehler oder Ausreißer das Modell unnötig zu invalidieren, ist ein Sliding-Window-Ansatz integriert, durch den ein Modell erst dann verworfen wird, wenn eine bestimmte Anzahl an Schwellwertüberschreitungen während der letzten Fenstergröße registriert wurden.

2.3. Realwelt-Testbett

Um all die entwickelten Algorithmen und Forschungsergebnisse bezüglich Public Sensing in einer realen Umgebung testen zu können, wurde in vorangegangenen Arbeiten an der Universität Stuttgart ein Testbett entwickelt, das auch Grundlage dieser Arbeit ist [AMT12]. Das vorhandene Realwelt-Testbett wurde so entworfen, dass es sowohl in vollem als auch in kleinerem Maßstab (z.B. innerhalb von Gebäuden) verwendet werden kann. In Abbildung 2.1 wird seine Architektur, bestehend aus mehreren Server-Applikationen und einer Android-App als Client für die beteiligten Smartphones, dargestellt. Diese Komponenten und ihre Funktionen werden im Folgenden vorgestellt.

2.3.1. Public-Sensing-Server

Der für das Realwelt-Testbett entwickelte Server hat unter anderem folgende Aufgaben:

- Sammeln und Speichern aller gemessenen Sensordaten, damit der Benutzer sie auch später noch an einer zentralen Stelle einsehen und auswerten kann. Die Messwerte werden dafür in einer Datenbank abgelegt.
- Unterstützung der Bluetooth-Lokalisierung, die im Testbett für die Positionierung innerhalb von Gebäuden verwendet wird. Der Server kennt den Standort von Bluetooth-Beacons und kann damit die Position eines Smartphones berechnen, je nachdem welche Beacons es in seiner Umgebung entdeckt hat.
- Bereitstellen eines Interfaces für den Benutzer, in dem er Erfassungsaufgaben anlegen, Beacons eintragen und Messergebnisse abrufen kann. So ist es zum Beispiel durch Klicken in eine Karte ganz einfach möglich, Messpunkte festzulegen und später für diese Punkte die Sensordaten einzusehen.
- Kontaktieren der Smartphones, damit diese Daten aufnehmen. Dafür sendet der Server zu den konfigurierten Zeitpunkten Nachrichten an die Smartphones, die die Position der Messpunkte und die benötigten Sensortypen einer Erfassungsaufgabe enthalten. Erkennt ein Smartphone durch eine vorherige Positionierung, dass es sich bei einem der Messpunkte befindet, kann es seine Sensoren aktivieren und Daten aufzeichnen.

Die gemessenen Sensordaten werden dabei in einer *MySQL-Datenbank* gespeichert, damit sie effizient und je nach Anforderung in verschiedenen Formen auch wieder ausgelesen werden können. Außerdem werden in der Datenbank auch alle nötigen Verwaltungsdaten und Erfassungsaufgaben gespeichert, die der Benutzer mit Hilfe eines Web-Interfaces bearbeiten und erstellen kann.

Das Web-Interface, der *PHP-Server*, dient sowohl für den Benutzer des Public-Sensing-Systems als auch für die Smartphones als Zugriffsstelle, um Daten aus der Datenbank zu lesen, zu verändern und neu hinzuzufügen. Damit die Erfassungsaufgaben zeitgesteuert getriggert und Smartphones automatisch kontaktiert werden können, wurde eine Java-Applikation, der sogenannte *Java-Server*, entwickelt.

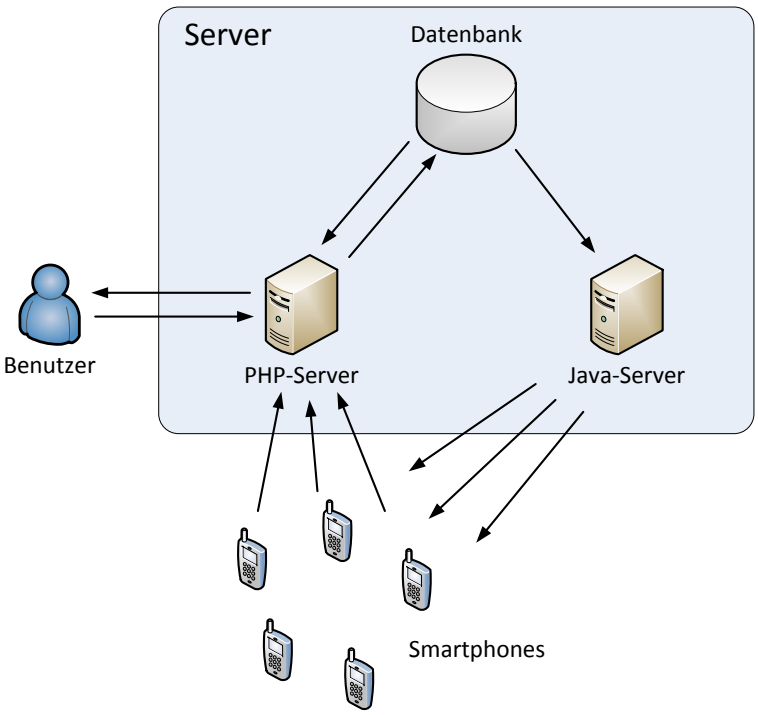


Abbildung 2.1.: Architektur des Realwelt-Testbetts

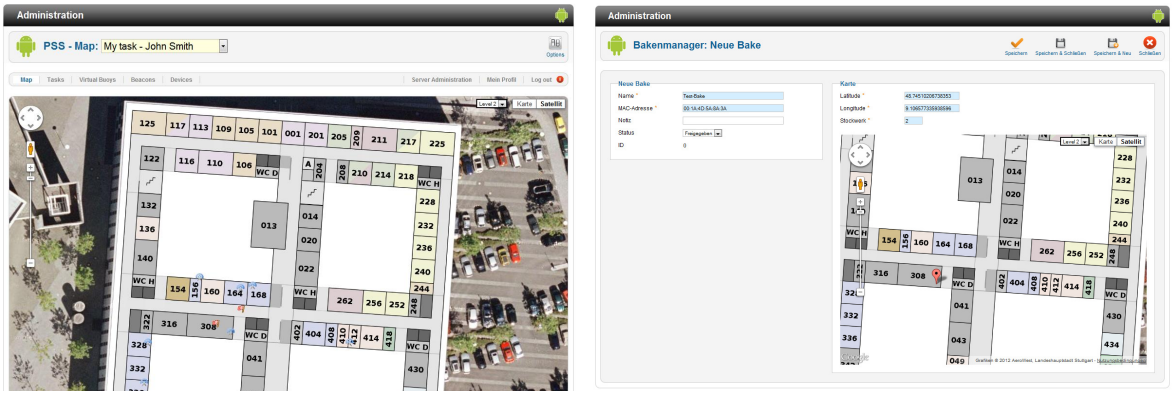
PHP-Server

Das Web-Interface ist in PHP geschrieben, da damit die Anbindung der Datenbank und Google-Maps sehr einfach umzusetzen war. Neben diesem graphischen Interface für den Benutzer (GUI), das passwortgeschützt ist, wird der PHP-Server auch von den Smartphones für die Lokalisierung und die Abgabe von Sensordaten verwendet. Beide Applikationen sind nach dem Model-View-Controller-Prinzip (MVC) entworfen.

Damit ein Smartphone vom Public-Sensing-System verwendet werden kann, müssen drei Bedingungen erfüllt sein:

- Das Smartphone muss vom Server erreichbar sein und umgekehrt (zumeist über eine Internetverbindung). Dabei ist es egal, ob für die Anbindung WLAN, Bluetooth, das mobile Internet oder auch eine Kombination daraus verwendet wird.
- Auf dem Smartphone muss die Public-Sensing-App für Android aktiv sein.
- Das Smartphone muss am Server angemeldet und dort als aktiv gekennzeichnet sein.

Die Aufgaben der Anmeldung, Lokalisierung und Sensordatenübermittlung werden durch HTTP-Requests der Smartphones an den Server realisiert.



(a) Mashup in Google-Maps **(b)** Ansicht zum Registrieren eines Beacons

Abbildung 2.2.: Screenshots vom Web-Interface des PHP-Servers

Loggt sich ein Benutzer mit seinen persönlichen Daten in der GUI des Servers ein, bekommt er eine Google-Map mit den Daten seiner zuletzt angelegten Erfassungsaufgaben angezeigt (siehe Abbildung 2.2a). Messpunkte dieser Aufgaben, deren letzten Sensordaten nach einem Klick per kleinem PopUp angezeigt werden, sind mit roten Fahnen gekennzeichnet. Durch eine Auswahlbox kann zwischen den verschiedenen angelegten Aufgaben gewechselt werden. Lokalisierte Geräte werden mit einem Handy-Symbol und Beacons mit einem blauen Funkturm-Symbol auf der Karte angezeigt.

Im Aufgabenmanager kann der Benutzer neue Erfassungsaufgaben erstellen und vorhandene verwalten. Zum Erstellen einer Aufgabe muss im Formular ein Name, Start- und Endzeitpunkt, die Sensortypen (Licht, Lautstärke, ...), Messpunkte, an denen Daten aufgenommen werden sollen, sowie der *Aufgabentyp* spezifiziert werden. Aufgabentypen wurden eingeführt, um Entwicklern die Möglichkeit zu geben, unterschiedliche Verhaltensweisen bei der Ausführung von Erfassungsaufgaben zu implementieren. Bisher wurde allerdings nur der Standard-Aufgabentyp verwendet, bei dem rundenbasiert in jeder Periode alle Messpunkte an die Smartphones geschickt werden, damit an diesen Stellen Messungen durchgeführt werden.

Des Weiteren kann der Benutzer im Bojenmanager seine Messpunkte detaillierter verwalten (zum Beispiel Radien mit angeben) und Standard-Messpunkte zur komfortablen Wiederverwendung definieren. Im Beaconmanager werden dagegen die Bluetooth-Beacons mit ihren MAC-Adressen und Positionen eingegeben (siehe Abbildung 2.2b), die dann zur Lokalisierung herangezogen werden. Zuletzt gibt es noch einen Gerätemanager, in dem alle registrierten Smartphones verwaltet werden.

Java-Server

Der Java-Server greift auf die gleiche Datenbank zu wie der PHP-Server, muss allerdings nur aus ihr lesen und nichts hineinschreiben. Er prüft in kurzen Abständen, ob neue Aufgaben

angelegt wurden und nimmt diese in seinen Scheduler auf, damit sie zum richtigen Zeitpunkt ausgeführt werden können. Je nach Aufgabentyp kann dies einmalig, periodisch oder in irgendeiner anderen Form geschehen.

Der Standard-Aufgabentyp wird beispielsweise periodisch getriggert, sodass je nach eingestellter Periodendauer Nachrichten an die Smartphones geschickt werden. Dies geschieht über TCP-Socket-Verbindungen zu den Geräten.

Die Nachrichten enthalten dabei jeweils alle vorhandenen Messpunkte einer Aufgabe sowie die Sensortypen, von denen Messwerte erwartet werden. Die Smartphones gleichen diese Messpunkte mit ihrer aktuellen Position ab und überprüfen, ob sie für die angeforderten Sensortypen entsprechende Sensoren besitzen. Befinden sie sich bei einem Messpunkt und verfügen über mindestens einen nötigen Sensor, zeichnen sie die Daten auf und schicken sie per HTTP-Request an den PHP-Server.

Weitere Aufgabentypen können ganz einfach implementiert werden, indem jeweils eine neue Klasse zum Code hinzugefügt wird.

2.3.2. Public-Sensing-Client

Der Public-Sensing-Client wurde für opportunistisches Public Sensing entwickelt. Es ist eine Android-App, die auf Smartphones im Hintergrund läuft und durch Interaktion mit dem Server oder anderen Geräten arbeitet. So kann sie zum Beispiel auf Anforderung des Servers automatisiert Sensoren aktivieren und Messwerte zurückschicken. Der Benutzer des Smartphones hat keine Möglichkeit die Funktionalität der App zu beeinflussen. Er kann nur zu Beginn benötigte Einstellungen tätigen (zum Beispiel muss die URL zum PHP-Server eingegeben werden), die Dienste der App starten und das Gerät auf dem Server registrieren.

Die Architektur der App ist sehr modular und basiert auf verschiedenen Services. Ein für diese Arbeit wichtiger Service ist der *LocatingService*. Dieser verwaltet das Positionierungssystem in der App und wurde so gestaltet, dass verschiedene Ansätze zur Positionierung auf einfache Weise hinzugefügt und über die Einstellungen ausgewählt werden können: sogenannte *Location-Provider*. So ist neben einer Lokalisierung über GPS auch die Bluetooth-Positionierung integriert, bei der die Umgebung nach sichtbaren Bluetooth-Beacons gescannt wird. Die so gefundenen MAC-Adressen werden per HTTP-Request an den Server geschickt, der die ungefähre Position des Gerätes mit Hilfe der eingetragenen Beacons berechnen kann. Dies passiert sofort, sodass die Antwort des HTTP-Requests bereits die Position des Gerätes enthält (sofern erfolgreich positioniert).

Bestimmt das Gerät seine Position per GPS oder auf eine andere Weise selbst, setzt der Service den HTTP-Request trotzdem ab, damit auch der Server die Position des Gerätes kennt. Dabei kann genau der gleiche Request verwendet werden, da der Server einen Fallback auf die eventuell mitgelieferten Positionsdaten hat.

Im Testbett besteht eine Position aus dem Längengrad, dem Breitengrad und einer Ebenennummer. Dies ist eine sogenannte 2,5-D-Position, wobei bei der GPS-Lokalisierung immer die 0 als Nummer der Ebene verwendet wird.

Der *ReadSensorsService* aktiviert je nach Anfrage die jeweiligen Sensoren und ist auch dafür zuständig, die gemessenen Werte durch HTTP-Requests an den PHP-Server zu übermitteln. Dagegen stellt der *SocketServerService* einen Socket-Server zur Verfügung, den der Java-Server kontaktieren kann, um Sensoranfragen zu übermitteln. Geht eine Nachricht ein, übergibt der Service die Aufgabe an den ReadSensorsService.

2.3.3. Bluetooth-Positionierung

Für den verkleinerten Maßstab wurde ein Positionierungssystem auf Basis von Bluetooth entwickelt, da GPS im Gebäudeinneren nicht zuverlässig und exakt genug funktioniert. Dabei suchen die Smartphones in regelmäßigen Abständen (ca. 30 Sekunden) ihre Umgebung nach sichtbaren Bluetooth-Geräten ab und senden alle gefundenen MAC-Adressen zusammen mit der jeweiligen Signalstärke an den Public-Sensing-Server. Der Server gleicht die MAC-Adressen dann mit bekannten und für die Positionierung verteilten Bluetooth-Beacons ab und bestimmt zusammen mit der Signalstärke die ungefähre Position des Smartphones. Nach erfolgreicher Lokalisierung sendet der Server die Position an das Smartphone zurück. Diese Positionierung basiert auf Ideen von [FKZL03] und [CIL06].

2.3.4. Verwendung und Ablauf

Für die Verwendung muss sichergestellt werden, dass einige Geräte mit der App im Testgebiet verfügbar sind. Ist dies der Fall, kann ein Benutzer des Public-Sensing-Systems Daten sammeln lassen. Dazu loggt er sich im Web-Interface des PHP-Servers ein und erstellt über das entsprechende Formular eine neue Erfassungsaufgabe (siehe Abbildung 2.3). Dabei muss er einen Namen, den Startzeitpunkt, den Endzeitpunkt, die Periodendauer und die gewünschten Sensortypen (z. B. Licht, Lautstärke) spezifizieren. In der Karte beim Formular klickt er an die Stellen, von denen er Daten erhalten möchte. Es werden damit automatisch die entsprechenden Messpunkte erstellt.

Wird der Startzeitpunkt der Aufgabe erreicht, übernimmt sie der Java-Server in seinen Scheduler und löst sie aus. Dabei werden die Positionsdaten der ausgewählten Messpunkte und die benötigten Sensortypen an alle registrierten Smartphones geschickt, die sich frei im Testgebiet bewegen können. Diejenigen, die durch vorherige erfolgreiche Lokalisierung ihre eigene Position kennen, überprüfen, ob sie sich in der Nähe eines Messpunktes befinden. Falls ja, aktivieren sie die angeforderten Sensoren (wenn vorhanden) und schicken die gemessenen Daten zurück an den Server. Dieser Ablauf wiederholt sich so lange periodisch, bis der Endzeitpunkt der Erfassungsaufgabe erreicht wird.

Während der gesamten Ausführungszeit und danach kann der Benutzer die Ergebnisse im Web-Interface einsehen und analysieren.

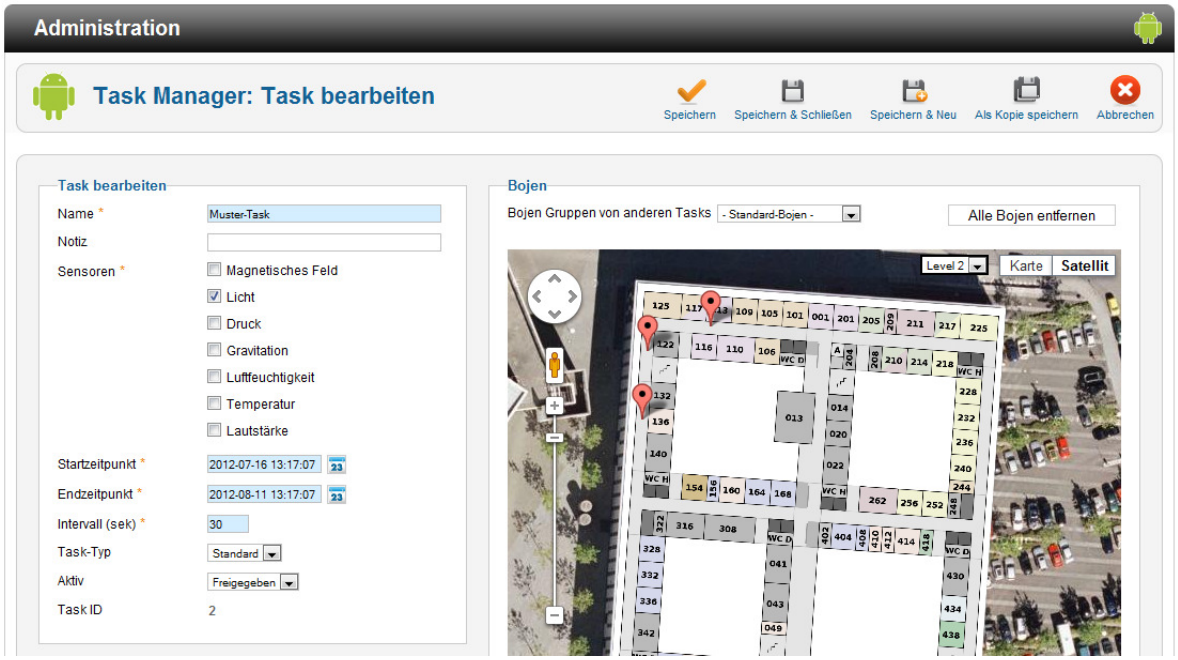


Abbildung 2.3.: Das Formular zum Erstellen einer Erfassungsaufgabe; dabei können links die Einstellungen vorgenommen und rechts in der Karte die virtuellen Sensoren gesetzt werden.

3. Systemmodell & Anforderungen

In dieser Arbeit kommt das Systemmodell von [AMT12] zum Einsatz, das im Folgenden nochmals dargestellt wird. Des Weiteren werden verwendete Begriffe erläutert und die Ziele der Arbeit formuliert.

3.1. Komponenten

Smartphones und andere mobile Geräte im Public-Sensing-System werden als *mobile Knoten* bezeichnet. Sie besitzen verschiedene Sensoren, wie zum Beispiel Helligkeits- und Temperatursensoren und verfügen über ein Positionierungssystem, mit dem sie sich lokalisieren können. Es wird angenommen, dass die Bewegung der mobilen Knoten vom System nicht beeinflusst werden kann.

Neben den mobilen Knoten gibt es im Public-Sensing-System einen Server, der sich im Internet befindet und über WLAN oder das mobile Internet für diese Geräte immer erreichbar ist. Der Server dient als Interface für den Benutzer, um Aufgaben für das System einreichen zu können und deren Ergebnisse anzuzeigen. Außerdem koordiniert der Server die Ausführung dieser Aufgaben, indem er je nach Aufgabe periodisch Anfragen für Sensorwerte an die mobilen Knoten verschickt.

Für die Bluetooth-Positionierung befinden sich darüber hinaus Bluetooth-Beacons im Gebiet, die über ihre MAC-Adresse identifiziert werden können. Der Server kennt dabei sowohl die MAC-Adressen der Beacons als auch ihre genaue Position.

Abbildung 3.1 zeigt alle Komponenten des Systemmodells und ihre Beziehungen zueinander im Überblick.

3.2. Anfragemodell

Möchte man über ein bestimmtes Gebiet Informationen sammeln, muss zunächst festgelegt werden, an welchen Stellen genau Messungen vorgenommen werden sollen. Zum Beispiel könnte es für die Leute in einer Stadt sehr interessant sein, an welchen Straßen es eher laut zugeht oder eher ruhiger. In einem statischen Sensornetzwerk würde man in diesem Fall einfach an allen interessanten Stellen einen Sensor anbringen. Wenn die Sensoren aber zusammen mit den Menschen umhergetragen werden, ist dies nicht möglich. Deshalb wurde die Idee der *virtuellen Sensoren* entwickelt [PDR11]. Ein virtueller Sensor ist ein Punkt im

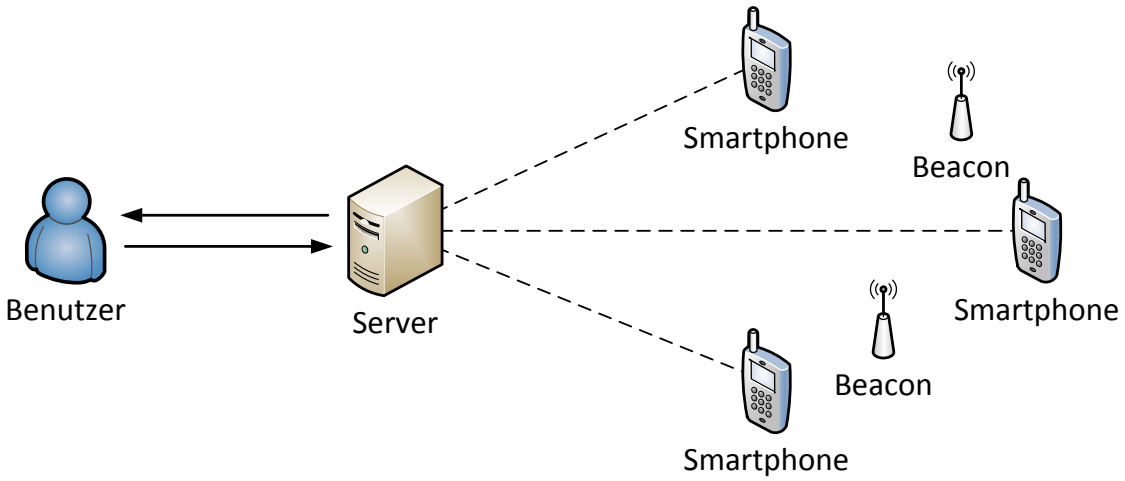


Abbildung 3.1.: Übersicht über das Systemmodell

abzudeckenden Gebiet, an dem Sensormesswerte aufgenommen werden sollen. Jeder dieser virtuellen Sensoren s hat ein kleines Erfassungsgebiet $g(s)$, in dem Messungen für den Sensor vorgenommen werden können. Befindet sich ein mobiler Knoten in $g(s)$, wird ein von diesem Knoten aufgenommener Messwert dem virtuellen Sensor s zugesprochen. Befindet sich mindestens ein mobiler Knoten in $g(s)$, wird s als *verfügbar* bezeichnet, andernfalls als *nicht verfügbar*.

Benutzer im Public-Sensing-System können zwei Rollen einnehmen. Zum einen sind sie Träger der mobilen Knoten und werden deshalb hier als *Besitzer* bezeichnet. Zum anderen können sie Aufgaben für das System spezifizieren und sind somit *Benutzer* des Public-Sensing-Systems.

Möchte ein Benutzer eine neue Aufgabe für das System formulieren, erstellt er eine *Erfassungsaufgabe*. Eine Erfassungsaufgabe A ist ein 6-Tupel $A = (S, a, t_s, t_e, p, E)$, wobei S ein Set von virtuellen Sensoren ist, a die Art oder der Typ des Sensors (zum Beispiel Licht oder Lautstärke), für den alle virtuellen Sensoren Messwerte liefern sollen, t_s der Zeitpunkt, zu dem die Erfassungsaufgabe gestartet werden soll, t_e der Endzeitpunkt, p die Intervalldauer, in der virtuelle Sensoren periodisch Messwerte liefern sollen, und E ein Set von Einstellungen für die modellgetriebenen Erfassungsmethoden.

Wie in 2.3 dargestellt unterstützt das verwendete Realwelt-Testbett auch mehrere verschiedene Sensortypen pro Erfassungsaufgabe. Da die modellgetriebenen Erfassungsmethoden aber nur mit einem Wert pro virtuellem Sensor und Periode funktionieren, ist a hier keine Menge, sondern nur ein Wert. Zur Vereinfachung arbeitet eine Aufgabe mit Modell-Anbindung also nur mit einem Sensortyp.

Bei der Ausführung einer Erfassungsaufgabe mit den modellgetriebenen Erfassungsmethoden gibt es verschiedene Arten von Messwerten. Die Sensormesswerte, die durch einen mobilen Knoten aufgezeichnet werden, werden hier als *effektive Messwerte* bezeichnet, die

vom Modell berechneten Daten entsprechend als *berechnete Werte*. Ein Teil der effektiven Messwerte sind dabei *Kontrollwerte*, die die Erfassungsmethoden nicht zur Berechnung der Inferenz verwenden, sondern zur Überprüfung, ob ein Modell valide ist.

3.3. Anforderungen

Bisher wurden die entwickelten modellbasierten Erfassungsmethoden nur in simulierten Umgebungen untersucht [PSDR12]. Aufgabe dieser Arbeit ist es, die modellbasierten Erfassungsmethoden in das Realwelt-Testbett der Universität Stuttgart zu integrieren, um ihre Funktion auch in realer Umgebung validieren zu können. Der Server soll dabei sowohl unter Linux als auch unter Windows lauffähig sein und mehrere Erfassungsaufgaben mit Modell-Anbindung parallel unterstützen.

Damit die Validität eines Modells gezeigt werden kann, ist es nötig, das System so zu erweitern, dass in jeder Periode auch *Referenzwerte* aufgenommen werden können. Referenzwerte sind dabei tatsächlich aufgenommene Sensordaten, die vom Modell aber in keiner Weise verwendet werden. Sie dürfen also nicht mit den Kontrollwerten verwechselt werden. Der Referenzwert eines virtuellen Sensors kann damit neben dem entsprechenden berechneten Wert des Modells in der GUI angezeigt werden, um sofort erkennen zu können, ob der berechnete Wert zur Realität passt.

Um den Code zur Integration der modellbasierten Erfassungsmethoden selbst testen zu können, ist vorgesehen, eine Testumgebung zu schaffen, in der Ausführungen von Erfassungsaufgaben mit Modell-Anbindung simuliert werden können. Der Testserver soll dabei fest integriert auch die Speicherung von Referenzwerten vornehmen.

Außerdem soll auch ein Demonstrator entwickelt werden, der das System soweit verkleinert, dass es auf der Größe einer Schreibtischplatte ausgeführt werden kann. Damit soll es möglich sein, die Funktionen der modellbasierten Erfassungsmethoden zum Beispiel mit Messungen und Berechnungen von Helligkeitswerten zu demonstrieren.

Dafür sind auch noch zusätzliche Ergänzungen am Server zu Analyse- und Vorführzwecken vorgesehen: Die Ausgabe der Sensormesswerte in der GUI muss übersichtlicher gestaltet werden. Zudem soll es möglich sein, dort auch ältere Daten und nicht nur die aktuellen Messwerte wieder anzeigen zu lassen, um nach der Ausführung die Ergebnisse in jedem Schritt noch einmal in Ruhe begutachten zu können.

Da die modellbasierten Erfassungsmethoden mit ermittelten Abhängigkeiten zwischen virtuellen Sensoren arbeiten, berechnen sie Kovarianzen für die Sensoren. Diese Kovarianzen sind nur aussagekräftig, wenn die zugrunde liegenden Messwerte zum gleichen Zeitpunkt aufgenommen wurden. In den Lernphasen müssen also immer möglichst alle virtuellen Sensoren in jeder Periode verfügbar sein. Da für den Demonstrator aber nur sehr wenige Smartphones vorliegen, muss eine Möglichkeit geschaffen werden, mit wenigen Geräten trotzdem möglichst viele Punkte pro Periode abzudecken.

Um das System im Informatikgebäude der Universität Stuttgart verwenden zu können, muss zudem die Bluetooth-Lokalisierung speziell für diesen Ort verbessert werden, da für das bestehende Testbett bisher nur die softwaretechnische Grundlage für die Bluetooth-Lokalisierung mit nur sehr wenigen Beacons in einem Abschnitt des Gebäudes entwickelt wurde.

4. Entwurf

Dieses Kapitel soll die Überlegungen vorstellen, die angestellt wurden, um die in Abschnitt 3.3 formulierten Ziele zu erreichen. Dazu wird zunächst auf die zu entwickelnde Komponente eingegangen, die die modellbasierten Erfassungsmethoden verwaltet und deren Anbindung an den Public-Sensing-Server erlaubt. Anschließend wird für diese Komponente die Testumgebung entworfen, in der sie evaluiert werden kann. Nach den Verbesserungen der Bluetooth-Positionierung und dem Entwurf des Demonstrators werden schließlich die nötigen Änderungen am bestehenden Realwelt-Testbett vorgestellt.

4.1. Architektur der Modell-Komponente

Der Modell-Code von [PSDR12] wurde in C++ geschrieben. Deshalb ist es vorgesehen, auch die Integration in das Testbett in C++ zu schreiben. Ein Problem stellt hierbei die Anbindung an den Server des Testbetts dar. Da das Modell Einfluss darauf nehmen muss, welche virtuellen Sensoren in jeder Runde angefragt werden, muss der Java-Server beim Verschicken seiner Nachrichten die Auswahl des Modells berücksichtigen. Da er, wie in 2.3.1 erläutert, auf einfache Weise um weitere Aufgabentypen erweitert werden kann, wird dafür ein neuer Aufgabentyp verwendet, der im Gegensatz zum Standard-Aufgabentyp nicht alle virtuellen Sensoren zur Anfrage auswählt, sondern nur die, die auch vom Modell ausgewählt wurden.

4.1.1. Kommunikation zwischen Modell und Java-Server

Damit der Java-Server erfährt, welche virtuellen Sensoren vom Modell aktiviert (ausgewählt) wurden, muss eine Kommunikationsleitung zwischen den beiden Applikationen entworfen werden. Es wäre zum Beispiel möglich, den Java-Server komplett durch einen in C++ implementierten Server mit Modell-Integration zu ersetzen. Eine weitere Möglichkeit ist das Verwenden von einer Socket-Verbindung zwischen den beiden Applikationen. Auch kann ein Datenaustausch über die bestehende Datenbank durchgeführt werden.

Da das Ersetzen des gesamten Java-Servers durch C++-Code extrem aufwendig ist und dabei die Kompatibilität der ebenfalls in Java geschriebenen Android-App verloren gehen würde, wird von dieser Lösung abgesehen. Bedenkt man, dass die virtuellen Sensoren sowieso bereits in der Datenbank gespeichert sind und die aktivierten Sensoren dort einfach über eine zusätzliche Spalte markiert werden können, fällt die Wahl auf die Informationsübertragung über die Datenbank. Die Socket-Verbindung wird dennoch benötigt, damit Java-Server und

Modell-Applikation sich gegenseitig informieren können, wenn Daten für die jeweils andere Seite bereitstehen. Dies wird über ein einfaches Handshake-Protokoll realisiert.

Hat die Modell-Applikation Sensoren ausgewählt, informiert sie den Java-Server mit der Nachricht `CALCULATION_FINISHED` darüber, sodass er Anfragen an die mobilen Knoten schicken kann. Nach 80% der Zeit einer Periode geht der Java-Server davon aus, dass keine weiteren Messwerte geliefert werden und teilt der Modell-Applikation per Nachricht `ROUND_FINISHED` mit, dass die Runde als beendet gilt. Es wurden 80% der Periodenzeit gewählt, da dies akzeptable Pufferzeiten bei Perioden mit 5 bis 60 Sekunden Dauer ergibt. Das Modell verwendet dann nur die bis zu diesem Zeitpunkt eingegangenen Sensormesswerte für die Inferenz und Validierung. Sollten dennoch so spät noch Messwerte ankommen, werden sie aber trotzdem noch vom Modell für die historischen Daten berücksichtigt.

Im Protokoll muss zusätzlich zu den beiden bereits genannten Nachrichten noch eine gesonderte Nachricht vorgesehen werden, mit der das Modell darüber informiert werden kann, dass der Endzeitpunkt der Aufgabe erreicht wurde und es sich somit beenden kann. Diese Nachricht heißt `TASK_FINISHED`.

Damit im Testbett weiterhin mehrere Erfassungsaufgaben parallel ausgeführt werden können, ist die Modellanbindung im Aufgabentyp so zu implementieren, dass mehrere Modell-Instanzen gestartet werden können. Zu jeder Aufgabe, die mit Modell-Unterstützung gestartet wird, soll also genau eine Modell-Instanz gehören.

Abbildung 4.1 zeigt den Verlauf von Nachrichten zwischen Modell und Java-Server im Überblick und in Abbildung 4.2 wird das System mit der zusätzlichen Modell-Applikation und ihren Beziehungen dargestellt.

Um das hier vorgestellte Kommunikationsprotokoll in den Java-Server durch einen zusätzlichen Aufgabentyp zu integrieren, reicht es aus, eine weitere Aufgabentyp-Klasse in den Java-Code einzufügen.

4.1.2. Modell-Applikation

Um den Modell-Code korrekt an das Testbett anbinden zu können, sind mehrere Punkte zu beachten, die in der Modell-Applikation integriert werden. Die Modell-Applikation ist damit ein Wrapper für den eigentlichen Modell-Code.

Verwaltung der Socket-Verbindung

Das Modell darf seine Berechnungen und die Sensorauswahl oder -markierung in der Datenbank erst durchführen, wenn es eine Nachricht vom Java-Server erhalten hat. Dafür wird eine Klasse entwickelt, die die Verwaltung der Socket-Verbindung übernimmt und die Arbeit der restlichen Modell-Applikation auslöst.

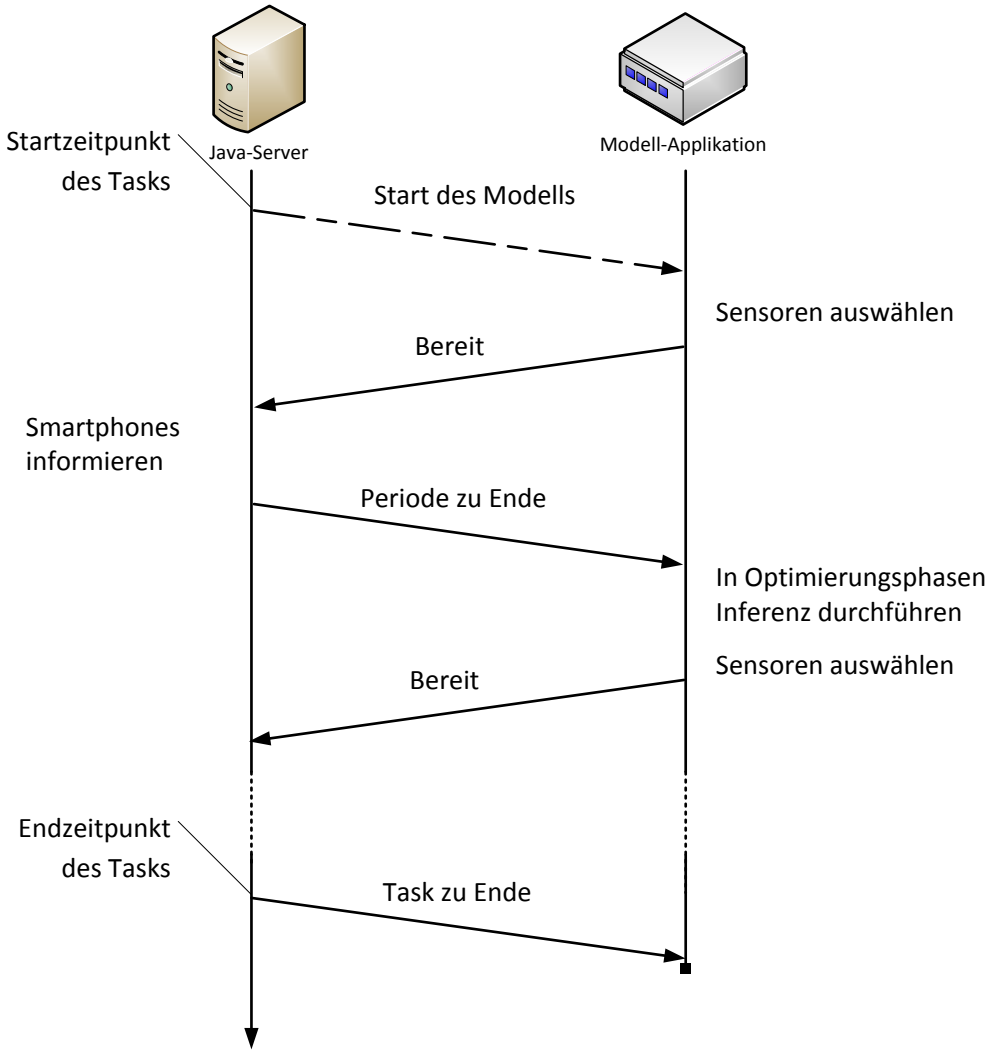


Abbildung 4.1.: Zeitlicher Ablauf der Aufgaben von Modell und Java-Server

Verwaltung des Modells

Für die Aufgaben des Modells müssen verschiedene Zustände gespeichert werden, zum Beispiel die ausgewählten Sensoren und deren Aufgabe, also ob sie für die Inferenz ausgewählt wurden oder für einen Kontrollwert. Die Klasse, die das übernimmt, wird außerdem die API-Funktionen des Modells aufrufen und sich um das Protokollieren der Modell-Statistiken kümmern.

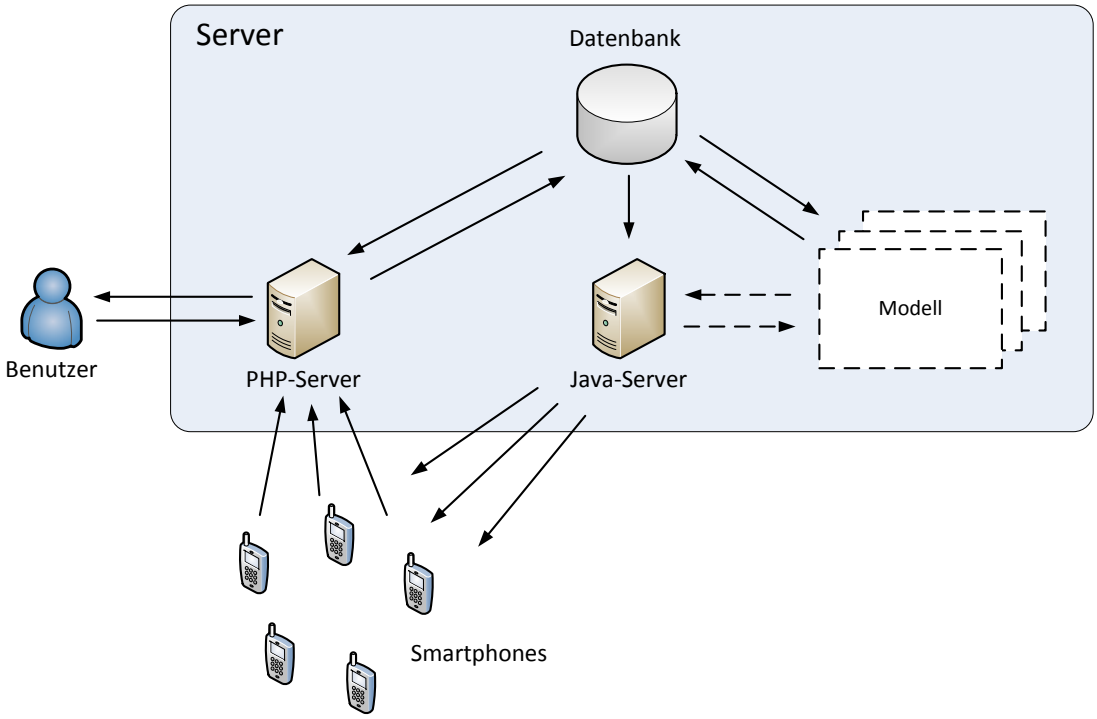


Abbildung 4.2.: Architektur des Realwelt-Testbetts mit Modell-Anbindung

Zugriff auf die Datenbank

Die vom Modell aktivierten Sensoren müssen in der Datenbank markiert werden. Damit am Ende der Periode eine Inferenz ausgeführt werden kann, müssen zunächst die aufgenommenen Messwerte dieser Periode aus der Datenbank gelesen werden. Außerdem müssen auch die Ergebnisse der Inferenz in der Datenbank gespeichert werden. Dies übernimmt eine weitere Klasse.

Für das Erzeugen eines konkreten Modells greift der Modell-Code mit dem sogenannten ObservationStore an verschiedenen Stellen auch direkt auf die gespeicherten Messwerte zu. Um diesen Zugriff auf die Datenbank zu realisieren, bei dem nicht nur die Daten einer Periode, sondern alle aufgenommenen Messwerte abgefragt werden, muss der vorhandene ObservationStore ersetzt werden.

Vereinigung von Perioden

Da in den Lernphasen immer möglichst alle virtuellen Sensoren in jeder Periode verfügbar sein sollten (siehe 3.3) und für den Demonstrator nur sehr wenige Smartphones vorlie-

gen, muss eine Möglichkeit gefunden werden, dem Modell-Code Messwerte mit gleichem Zeitstempel zu übergeben, obwohl sie gar nicht gleichzeitig aufgezeichnet wurden.

Eine Idee ist das Ende einer Lernphase manuell zu bestimmen. Davor können die mobilen Knoten beliebig im Testgebiet umher bewegt werden, während sie kontinuierlich Daten aufzeichnen. Wird die Lernphase beendet, können die Sensorwerte so aufgefasst werden, als wenn sie alle gleichzeitig gemessen wurden, damit das Modell Kovarianzen berechnen kann. Um mehrere Datensätze verwenden zu können, könnten die Daten nachträglich auf einige wenige Perioden aufgeteilt werden, die dann jeweils möglichst viele virtuelle Sensoren beinhalten.

Eine andere Idee ist, die periodische Anforderung von Sensorwerten beizubehalten, am Ende der Lernphase aber mehrere Perioden zusammenzufassen und dem Modell als kleinere Zahl insgesamter Perioden dafür aber mit jeweils wesentlich mehr Daten zu präsentieren.

Bei beiden Optionen ist zu beachten, dass sie nur funktionieren, wenn sich der Zustand des Testgebietes in der Lernphase nicht verändert (es also zum Beispiel nicht dunkler wird). Nur so ist es zulässig, dass Daten nachträglich einem anderen Zeitpunkt zugeordnet werden.

Bei der Idee mit den manuellen Lernphasen könnten diese durch die kontinuierlichen Messungen zwar wesentlich schneller durchgeführt werden, dadurch dass kontinuierlich Messwerte aufgenommen werden, ist es aber sehr wahrscheinlich, dass viele fehlerhafte Werte verwendet werden. Zum Beispiel könnte beim Umlegen der Smartphones der Helligkeitssensor aus Versehen verdeckt werden oder kurzzeitig im Schatten einer Person liegen. Deshalb wird die Idee der zusammengefassten Perioden umgesetzt. Für diese ist nur eine Änderung in der Modell-Applikation nötig, die durch Änderungen an den Zeitstempeln in der Datenbank veranlasst, dass jeweils mehrere Perioden nachträglich zu einer vereinigt werden. Da die rundenbasierte Ausführung der Aufgaben damit beibehalten wird, muss am Rest des Systems nichts verändert werden.

Hilfsfunktionen

Für die Konfiguration des Systems sind Klassen nötig, die Zugriff auf sowohl die globalen als auch die aufgabenpezifischen Einstellungen bieten.

Des Weiteren muss eine Möglichkeit vorgesehen werden, den Benutzer über bestimmte Ereignisse zu informieren. Falls nur wenige Smartphones verfügbar sind und das Vereinigen von Perioden nötig ist, muss er zum Beispiel darauf aufmerksam gemacht werden, wann eine Lernphase beginnt und wann sie endet, da die Smartphones in den Lernphasen bewegt werden müssen, um die Zahl verfügbarer virtueller Sensoren zu erhöhen. Eine Möglichkeit wäre, in der GUI des PHP-Servers eine entsprechende Meldung erscheinen zu lassen. Da aber nur die Modell-Applikation selbst weiß, welche Phase gerade aktiv ist, gibt es keinen einfachen Weg, diese Information an den PHP-Server zu übertragen, zu dem keine direkte Verbindung existiert. Die Modell-Applikation muss den Benutzer also irgendwie selbst informieren. Da der Benutzer diese Informationen auch erhalten muss, wenn er die GUI gerade nicht sehen kann, wird der Benutzer via Audio-Signale über den Wechsel zwischen

Lern- und Optimierungsphasen informiert. Außerdem können die Audio-Signale auch dafür verwendet werden, den Benutzer über unerwartete Fehler zu informieren.

4.2. Testserver für Simulationen

Da die Sensormesswerte und die virtuellen Sensoren von einem Test-Datensatz vorgegeben werden, müssen beim Einsatz des Testservers sowohl die mobilen Knoten als auch der PHP-Server durch die simulierte Umgebung ersetzt werden. Um dies zu erreichen, soll auch der Java-Server durch einen Testserver ersetzt werden, der anstatt dem Versenden von Anfragen für Messwerte nach und nach direkt die Werte aus einem Test-Datensatz in die Datenbank einliest. Auf diese Weise kann der Ablauf einer Periode auch stark beschleunigt werden. Es muss dabei nur beachtet werden, dass eine Periode mindestens eine Sekunde dauert, da die Zeitstempel technisch bedingt im Realwelt-Testbett nur mit einer Auflösung von einer Sekunde in der Datenbank gespeichert werden.

4.3. Bluetooth-Positionierung

Um die modellbasierten Erfassungsmethoden auch in etwas größerem Maßstab im Gebäude testen zu können, soll die Bluetooth-Positionierung insoweit verbessert werden, dass sie zumindest in zwei Gängen des Gebäudes raumgenau funktioniert.

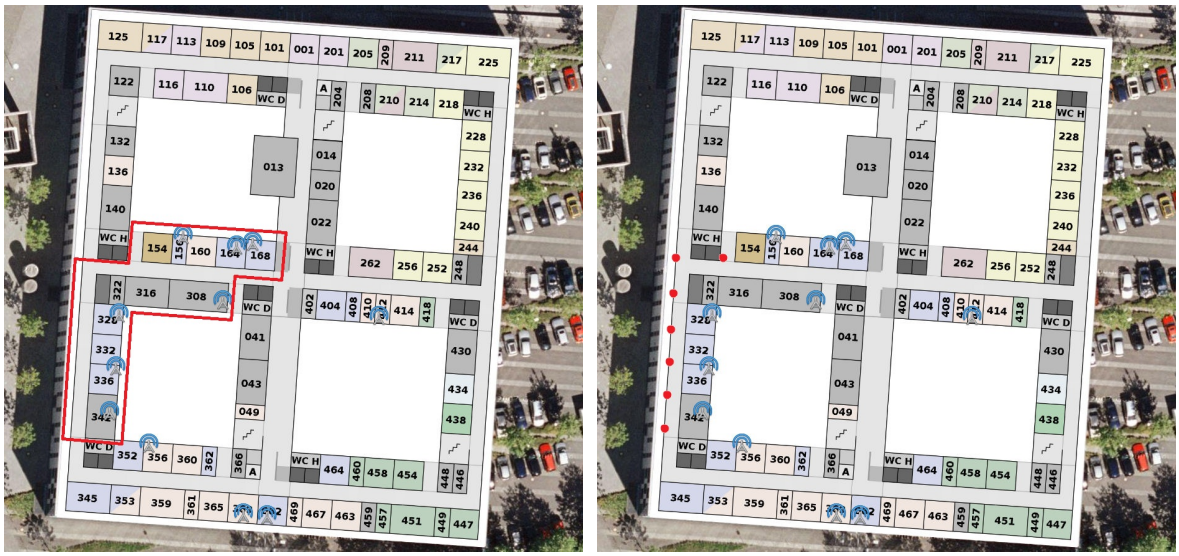
In der vorangegangen Arbeit wurden für den Test des Positionierungssystem im Informatikgebäude der Universität Stuttgart bereits herkömmliche Bluetooth-USB-Sticks an Computern im zweiten Stock angebracht. Momentan sind die Beacons dort so verteilt wie in Abbildung 4.3a zu sehen. Dabei funktioniert die Positionierung bisher nur im mit A gekennzeichneten Bereich raumgenau. Im süd-westlichen Gang ist es aber zum Beispiel nicht möglich, einen mobilen Knoten im Gang selbst korrekt zu lokalisieren: Angenommen ein Knoten befindet sich an der mit P markierten Position (siehe Abbildung 4.3b), dann würde dieser stattdessen ungefähr bei der mit P' markierten Position lokalisiert werden, da westlich des Knotens kein Beacon verfügbar ist und das System nicht ausmachen kann, wo sich der Knoten im Bereich B genau befindet (angenommen der Knoten empfängt Signale von den beiden ihm am nächsten gelegenen Beacons). Allgemein würde jeder Knoten dort irgendwo auf der Geraden g lokalisiert werden, egal ob er sich im Gang oder in einem der Zimmer befindet.

Um den in Abbildung 4.4a markierten Bereich abdecken zu können, müssen im süd-westlichen Gang also noch zusätzliche Beacons an der Wand des Ganges angebracht werden (vgl. 4.4b). Da dort keine Computer verfügbar sind, um Bluetooth-USB-Sticks verwenden zu können, kommen dort an Batteriepackungen angeschlossene Bluetooth-Beacons zum Einsatz.

Tests haben gezeigt, dass diese Bluetooth-Beacons eine so hohe Reichweite haben, dass sie fast im gesamten Gang sichtbar sind. Ein weiteres Problem ist, dass Beacons mit großem Abstand zum Smartphone immer eine ähnlich große Signalstärke aufweisen, auch wenn einige von

(a) Momentane Position der Beacons **(b)** Fehlerhafte Positionierung

Abbildung 4.3.: Position der Beacons und deren Auswirkungen



(a) Abzudeckender Bereich **(b)** Zusätzlich anzubringende Beacons

Abbildung 4.4.: Abzudeckender Bereich mit dafür nötigen zusätzlichen Beacons

ihnen noch mehrere Meter weiter entfernt sind als andere. Nur Beacons mit geringem Abstand zum Smartphone unterscheiden sich in ihrer Signalstärke so weit, dass sie für die Positionierung einen Nutzen haben und sich nicht negativ auswirken. Die Berechnung einer Position wird deshalb so angepasst, dass nur noch höchstens drei Beacons berücksichtigt werden: Wurden mehr als drei Beacons erkannt, verwendet der PHP-Server nur noch die drei Beacons, zu denen die größten Signalstärken gehören.

4.4. Demonstrator

Um die Funktionen des Modells in einem sehr kleinen Bereich vorführen zu können, soll ein Demonstrator entwickelt werden, der auf einem Tisch Platz findet und dort zur Messung von Helligkeitswerten voll funktionsfähig ist. Als Server kann dort ein einfacher Laptop mit Internetanbindung dienen und zur Veränderung von Helligkeitswerten in bestimmten Bereichen werden Schreibtischlampen und Abschattungen aufgestellt. Mehr Probleme bereitet die Positionierung der Smartphones auf dem Schreibtisch.

4.4.1. Positionierung

Um die Bluetooth-Positionierung ebenfalls auf diese Größe zu verkleinern, wurden die Bluetooth-Beacons an den Rändern des Testgebietes auf einem Schreibtisch und etwas entfernt an den Wänden des Raumes verteilt. Dadurch ist das Gebiet gut abgedeckt, jedoch haben Tests gezeigt, dass dies nicht funktioniert, da sich die empfangenen Signalstärken bei diesen kleinen Abständen zu wenig unterscheiden bzw. insgesamt zu stark variieren, um eine ausreichend genaue Lokalisierung durchführen zu können.

Aus diesem Grund ist es nötig, für den Demonstrator eine andere Art der Lokalisierung zu entwerfen, die dann auf mehrere Zentimeter genau zuverlässig funktioniert. Da manche moderne Smartphones bereits heute zusätzlich mit einem NFC-Lesegerät (Near-Field-Communication) ausgestattet sind, können NFC-Tags auf der Tischfläche für die Positionierung verwendet werden. Wenn jeder NFC-Tag eindeutig identifiziert werden kann, ist es möglich ein Smartphone einer bestimmten Position zuzuordnen, wenn es auf einen Tag gelegt wird. Da NFC, wie der Name schon sagt, nur auf sehr kurze Entfernung funktioniert (maximal $4cm$), kann es bei entsprechender Platzierung der Tags nicht passieren, dass aus Versehen ein anderer Tag außer dem direkt unter dem Smartphone liegenden gelesen wird, wodurch eine sehr genaue Positionierung möglich wird. Abbildung 4.5 zeigt eine nach einigen Tests erstellte Anordnung, bei der die Tags möglichst sparsam verteilt werden und die Smartphones trotzdem beliebig in das Testgebiet gelegt werden können und dabei immer einen Tag erkennen.

NFC-Tags enthalten einen Mikrochip, in dem kleine Datenmengen gespeichert werden können. Hier müssen die Tags nur eine eindeutige ID enthalten, um sie voneinander unterscheiden zu können. Um die NFC-Lokalisierung in das Bluetooth-Positionierungssystem auf dem Server einzubetten, generieren die Smartphones aus der ID eine eindeutige MAC-Adresse.

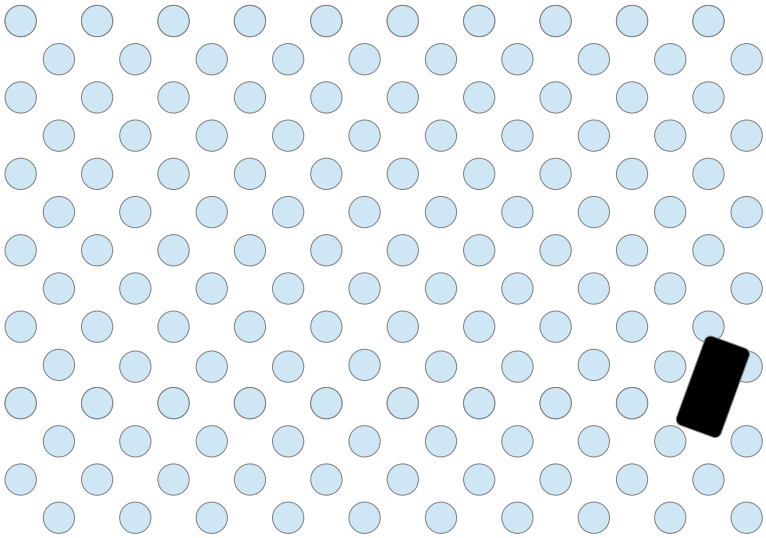


Abbildung 4.5.: Optimale Verteilung von NFC-Tags. Um die Größenverhältnisse nachvollziehen zu können, ist ein Smartphone angedeutet.

Das Smartphone schickt, wie nach einem Bluetooth-Scan, die abgeleitete MAC-Adresse an den Server, der damit die Position des Smartphones bestimmen kann. Erhält der Server nur eine MAC-Adresse vom mobilen Knoten, legt er dessen Position automatisch direkt auf die Position des Beacons bzw. des Tags. Dazu müssen die NFC-Tags nur genau wie die Bluetooth-Beacons auf dem Server eingetragen werden, wodurch sich für den Benutzer eine gemeinsame Oberfläche in der GUI für die Verwaltung beider Positionierungssysteme ergibt. Wichtig ist, dass eventuell vorhandene Bluetooth-Beacons keinen Einfluss auf die durch einen NFC-Tag festgelegte Position nehmen.

Location-Provider für NFC-Lokalisierung

Um die NFC-Lokalisierung auf der Seite der Clients zu realisieren, muss eine Erweiterung für die Android-App entworfen werden. Wie in Abschnitt 2.3.2 erläutert, können in der App sehr einfach neue Location-Provider hinzugefügt werden, ohne die Architektur der App zu verändern. Es muss also nur ein Modul entwickelt werden, das folgende Aufgaben erledigt:

- Auslesen aller erkannten NFC-Tags
- Entscheiden, ob ein Tag zum Public-Sensing-System gehört
- Umwandeln der ID des Tags in eine MAC-Adresse
- Senden der MAC-Adresse an den PHP-Server

Der mobile Knoten erhält damit genau wie bei der Bluetooth-Lokalisierung seine aktuelle Position zurück.

Außerdem soll für Smartphones ohne NFC-Unterstützung eine Möglichkeit geschaffen werden, die ID eines Tags bei den Einstellungen der App angeben zu können, um sie zumindest statisch im Testgebiet positionieren zu können.

4.5. Ergänzungen am Realwelt-Testbett

In diesem Abschnitt wird der Entwurf der am Realwelt-Testbett nötigen Ergänzungen für den Demonstrator und die Aufzeichnung von Referenzwerten vorgestellt.

4.5.1. Ergänzungen an der GUI

Bisher musste man einen virtuellen Sensor in der Karte anklicken, um dessen Werte einsehen zu können, worauf sich ein kleines PopUp-Fenster mit einigen Informationen, wie die letzten Messwerte und die Anzahl aller Messwerte für jeden Sensortyp dieses Sensors, öffnet.

Um Werte besser vergleichen und präsentieren zu können, wird die Karte so verändert, dass die letzten Messwerte und die Anzahl aller Messwerte direkt unterhalb des Sensors in einer kleinen Box angezeigt werden, ohne dass man zuvor auf den Sensor klicken muss. Auf diese Weise kann man die letzten Messwerte von allen Sensoren im Überblick sehen. Damit zwischen tatsächlich aufgenommenen und berechneten Werten unterschieden werden kann, ist es vorgesehen, die berechneten Werte in einer anderen Farbe darzustellen. Dafür erhält die Tabelle der gespeicherten Messwerte eine weitere Spalte, in der vermerkt werden kann, ob Daten berechnet oder tatsächlich aufgenommen wurden.

Außerdem soll es möglich sein, nicht nur die aktuellen, sondern auch ältere Daten bei den Sensoren anzuzeigen. Mit einem Schieberegler kann man einen Zeitpunkt auswählen, von dem dann die Sensordaten angezeigt werden, indem sich mit dem Regler der Inhalt der Boxen unter den Sensoren ändert. Es ist von Vorteil, wenn dieser Regler aktivierbar und deaktivierbar ist, da damit im ausgeschalteten Zustand kontinuierlich aktuell ankommende Daten nachgeladen werden können. Erst wenn der Regler eingeschaltet wird, stoppt die Aktualisierung und man kann einen bestimmten Zeitpunkt in der Vergangenheit aufrufen.

Damit die Aktualisierung der Daten in den Boxen kontinuierlich und unabhängig geschehen kann, ohne dass die ganze Karte neu geladen werden muss, werden die Daten per Ajax-Request (asynchrone HTTP-Anfrage) abgerufen.

Es ist darüber hinaus vorgesehen, den Aufgabenmanager des PHP-Servers zu erweitern, sodass beim Anlegen einer Aufgabe neben den Pflichtangaben, wie zum Beispiel Sensortyp und Startzeitpunkt, auch weitere Einstellungen je nach Aufgabentyp festgelegt werden können. Beim Aufgabentyp für das Modell können dort zum Beispiel die Qualitätsparameter spezifiziert werden.

4.5.2. Referenzwerte

Um das Aufzeichnen und Anzeigen von Referenzwerten zu realisieren, ist es nötig einen (virtuellen) neuen Datentyp einzuführen, der vom Java-Server verwaltet wird. Wenn die Option zum Speichern von Referenzwerten aktiviert ist, soll der Aufgabentyp des Modells beim Versenden der Anfragen für Sensorwerte zunächst die Auswahl des Modells ignorieren und für alle virtuellen Sensoren Messwerte einfordern. Am Ende der Periode, aber vor dem Informieren des Modells, werden dann die Ergebnisse von nicht aktivierten Sensoren in den neuen Datentyp für Referenzwerte umgewandelt. Diese Ergänzung hat den Nebeneffekt, dass der Java-Server jetzt auch in die Datenbank schreiben muss und nicht mehr nur aus ihr lesen. Auf diese Weise greift das Modell auch weiterhin nur auf die Ergebnisse zurück, die es angefordert hat.

Wird bei der Anzeige in der Karte für einen berechneten Wert ein Referenzwert aus der gleichen Periode gefunden, werden beide in der Informationsbox des Sensors eingeblendet.

4.6. Beispielablauf

Um die neuen Komponenten und deren Verwendung noch einmal zusammenzufassen, wird hier der Ablauf einer Erfassungsaufgabe vorgestellt, wenn sie mit modellbasierten Erfassungsmethoden ausgeführt werden soll. Dabei wird aber nur auf die Unterschiede zum in Abschnitt 2.3.4 dargestellten Ablauf eingegangen.

Beim Anlegen einer Erfassungsaufgabe wählt der Benutzer jetzt den Aufgabentyp „Modell“ aus und kann außerdem zusätzliche Einstellungen festlegen (z. B. Qualitätsparameter für das Modell oder ob Referenzwerte aufgezeichnet werden sollen).

Wird die Aufgabe vom Java-Server zum ersten Mal ausgelöst, startet er zunächst die Modell-Applikation mit Aufgaben-ID und einer eindeutigen Port-Nummer und baut direkt eine Socket-Verbindung zu ihr auf.

Das Modell kann dann mit der Lernphase beginnen und die ersten Sensoren aktivieren. Nach dem Versenden einer Socket-Nachricht an den Java-Server kann dieser die Anfragen an die mobilen Knoten weiterleiten, nachdem die Periode zu 80% beendet ist. Durch das Senden einer Socket-Nachricht zurück an die Modell-Applikation wird sie informiert, dass sie jetzt mit ihren Berechnungen beginnen und die nächsten Sensoren auswählen kann. Wenn nach einigen Runden die Lernphase vorüber ist, wechselt das Modell in eine Optimierungsphase. Fehlende Werte werden in dieser Phase durch Inferenz zur Verfügung gestellt.

Sollte der Endzeitpunkt einer Aufgabe erreicht werden, wird das Modell durch eine entsprechende Socket-Nachricht dazu veranlasst, sich zu beenden.

Während der gesamten Ausführungszeit und danach kann der Benutzer die Ergebnisse im Web-Interface mitverfolgen und über den Schieberegler Daten von beliebigen Punkten in der Vergangenheit anzeigen lassen.

5. Implementierung

Dieses Kapitel soll die Implementierung der verschiedenen Komponenten und Ergänzungen näher beschreiben und auf die aufgetretenen Probleme eingehen. Dafür werden zu Beginn die Änderungen am bestehenden Realwelt-Testbett vorgestellt, bevor der Aufbau der entwickelten Modell-Applikation und des Testservers dargestellt wird. Anschließend wird auf die Hardware des Demonstrators und die für ihn nötige Ergänzung an der Android-App eingegangen. Zuletzt wird die Verbesserung an der Bluetooth-Positionierung erläutert.

5.1. Änderungen am Testbett

Im Realwelt-Testbett musste sowohl der Java-Server als auch der PHP-Server um verschiedene Funktionen erweitert werden.

5.1.1. Java-Server

Wie in 4.1.1 erwähnt, musste zur Integrierung des Kommunikationsprotokolls zwischen Modell-Applikation und Java-Server nur eine weitere Klasse implementiert werden. Diese hat den Namen `TaskTypeModel` und erbt von der abstrakten Klasse `TaskType`, die neben der Instanziierung der Aufgabentypen über die statische Methode `getInstance` auch als Interface für sie dient.

Für den Start der Modell-Applikation wird im Konstruktor von `TaskTypeModel` mit den entsprechenden `Java-ProcessBuilder`-Funktionen ein Prozess der Modell-Applikation gestartet. Sobald eine Aufgabe mit Aufgabentyp „Model“ zum ersten Mal ausgelöst wird, startet der Code des Aufgabentyps also auch das Modell. Damit mehrere Aufgaben mit eigenem Modell parallel existieren können, ohne dass sie sich gegenseitig stören, wird der Modell-Applikation beim Start per Kommandozeilen-Parameter die ID der jeweiligen Aufgabe und eine eindeutige Port-Nummer mit übergeben. Auf dies Weise hat jede Modell-Instanz einen eigenen Port für ihre Verbindung mit dem Java-Server.

Die Kommunikationsverbindung wird direkt nach dem Start aufgebaut. Falls dabei ein Fehler auftritt, geht das `TaskTypeModel`-Objekt in einen Fallback-Modus, in dem es sich so verhält wie der Standard-Aufgabentyp und in jeder Runde alle Sensoren aktiviert. Das gleiche passiert, wenn die Modell-Applikation aus irgendeinem Grund abstürzen sollte.

Jeder Aufgabentyp im Java-Server muss die Methode `getBuoys()` implementieren, die als Rückgabewert eine Liste mit allen virtuellen Sensoren enthält, für die der Java-Server

Messwerte anfordern soll. Für jeden Sensor wird dabei auch vermerkt, welche Sensortypen verlangt sind. Die Klasse `TaskTypeModel` liefert hier also immer alle Sensoren zurück, die vom Modell ausgewählt (aktiviert) wurden. Dies geschieht durch Auswerten der zusätzlichen Spalte `buoy_active` in der Datenbanktabelle der virtuellen Sensoren, in der die Modell-Applikation die aktivierten Sensoren durch Einfügen einer 1 markiert. Da jede Erfassungsaufgabe ihre eigenen Sensoren in der Datenbank gespeichert hat, auch wenn sie sich an genau der gleichen Stelle befinden, kann es hier nicht zu Problemen mit parallel ausgeführten anderen Aufgaben kommen. In Abbildung 5.5 auf Seite 50 ist eine Übersicht aller Änderungen an der Datenbank wie dieser (Abbildung 5.5a) zu sehen.

Bei der Ausführung einer Erfassungsaufgabe instanziiert der Java-Server in jeder Periode ein Objekt der Klasse `TaskJob` und führt dessen `execute`-Methode aus. Dort wird vor dem Verwenden von `getBuoys` und dem Verschicken der Anfragen an die mobilen Knoten die Methode `onBeforeRound` des jeweiligen Aufgabentyps aufgerufen. Diese Methode wird nun durch Überschreiben dafür verwendet, abzuwarten, bis das Modell mit der Nachricht `CALCULATION_FINISHED` (vgl. Abschnitt 4.1.1) meldet, dass es mit seiner Berechnung fertig ist.

Nach 80% der Zeit einer Periode wird die Methode `onAfterRound` aufgerufen, die dazu genutzt wird, dem Modell per Socket-Nachricht `ROUND_FINISHED` mitzuteilen, dass das Ende der Periode jetzt erreicht wurde und es mit seinen Berechnungen beginnen kann.

Wird eine Erfassungsaufgabe in der GUI des PHP-Servers vorzeitig vom Benutzer beendet, ruft der Java-Server die Methode `onInterrupt` auf, in der das `TaskTypeModel`-Objekt die Nachricht `TASK_FINISHED` an die Modell-Applikation übermittelt, sodass sie sich selbst beendet. Da der Ausführungskontext der jeweiligen Aufgabe den Methoden `onBeforeRound` und `onAfterRound` übergeben wird, kann das `TaskTypeModel`-Objekt auch erkennen, wenn planmäßig keine weiteren Ausführungen der Aufgabe anstehen und somit auch in diesem Fall `TASK_FINISHED` an die Modell-Applikation senden.

Abbildung 5.1 zeigt die hier angesprochenen Klassen und den Standard-Aufgabentyp `TaskTypeDefault` zusammen mit ihren wichtigsten Methoden. Die Parameter der Methoden wurden der Übersichtlichkeit wegen ausgeblendet.

5.1.2. PHP-Server

Da der PHP-Server nach dem MVC-Prinzip entworfen wurde, ist er sehr modular aufgebaut und kann durch einfache Ergänzungen an den entsprechenden Stellen erweitert werden.

Ergänzungen an der Karte

Für die Anzeige der Sensormesswerte in den PopUps der virtuellen Sensoren wurde bei der Entwicklung des Realwelt-Testbetts ein *View* erstellt, der für jedes PopUp per Ajax geladen wird. Jedes dieser kleinen Fenster ist also unabhängig von den anderen. Da jetzt der gleiche Inhalt nur mit anderem Layout in kleinen Info-Boxen unterhalb der virtuellen

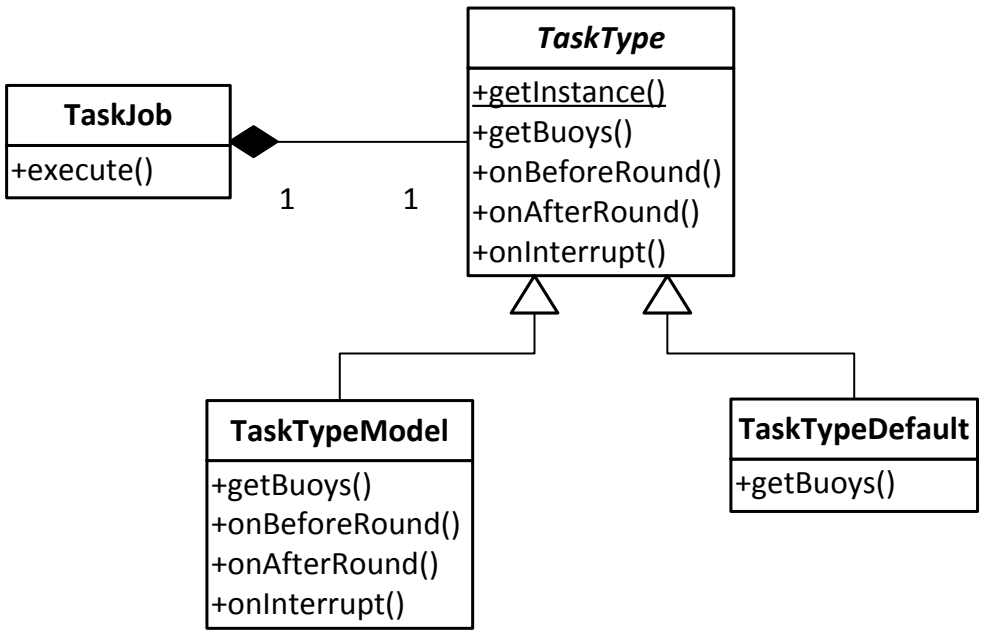


Abbildung 5.1.: Klassenhierarchie der Aufgabentypen

Sensoren angebracht werden sollte, wurde ein zweites Layout, namens `plain`, für diesen View erstellt. Das bedeutet, die gleichen Informationen werden nur in anderer Form dargestellt. Durch die MVC-Architektur kann man für die Info-Boxen den gleichen Ajax-Request wie für die PopUps verwenden, es muss nur der Parameter `layout=plain` noch mit in der URL übergeben werden. Die Erstellung der Info-Boxen selbst war mit den API-Funktionen von Google-Maps möglich, mit denen sie sich ganz einfach bei den virtuellen Sensoren positionieren ließen. Periodische Ajax-Requests aktualisieren die Boxen ständig mit den neuesten Daten, wobei berechnete Werte stets mit pinker und alle anderen mit gelber Hintergrundfarbe angezeigt werden. Möglich ist dies mit Hilfe der zusätzlichen Spalte `data_virtual` in der Datenbanktabelle für Messwerte (siehe Abbildung 5.5b), die von der Modell-Applikation entsprechend befüllt wird.

Um den Schieberegler für die Anzeige älterer Daten zu realisieren, wurden mit den Google-Funktionen für Kontrollschaltflächen am oberen Rand ein horizontaler Regler erstellt, der die Ajax-Requests von den Info-Boxen um einen zusätzlichen Zeitstempel ergänzt, der von der Position des Reglers abhängt. Um diesen Zeitstempel zu beachten, war nur eine kleine Änderung an den Datenbankabfragen für Messwerte durchzuführen. Wird kein Zeitstempel übergeben, liefern sie weiterhin immer die jüngsten Daten zurück.

Der Regler muss zusätzlich vor der Verwendung noch durch eine Checkbox aktiviert werden, um die periodischen Ajax-Requests zu stoppen. Dies hat den Grund, dass damit der Regler

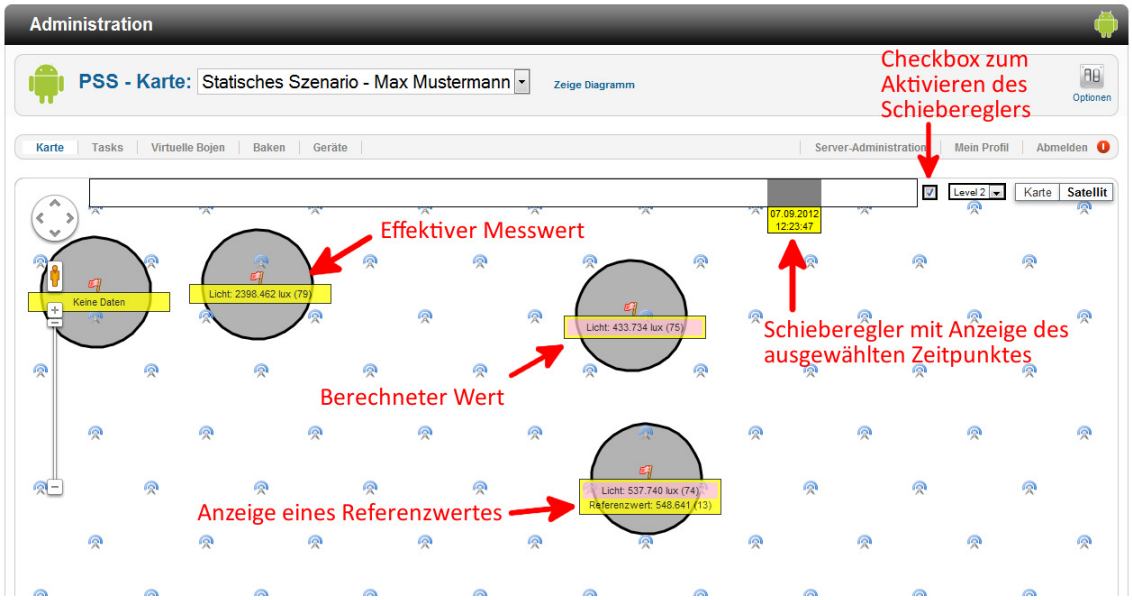


Abbildung 5.2.: Übersicht über neue Funktionen in der Karte

auf einen bestimmten Zeitraum festgelegt werden kann: Das linke Ende entspricht dann dem Startzeitpunkt der Erfassungsaufgabe und das rechte Ende dem Moment, in dem die Checkbox aktiviert wurde. Auf diese Weise kann man die Anzeige auch einfrieren, um bestimmte Daten genauer begutachten zu können.

In Abbildung 5.2 ist die Karte der GUI mit den hier erwähnten Ergänzungen zu sehen.

Aufgabenmanager

In der Ansicht zum Hinzufügen und Bearbeiten von Erfassungsaufgaben wurde ein Abschnitt hinzugefügt, in dem für jeden Aufgabentyp zusätzliche Einstellungen getätigt werden können. Dies wurde dabei über eine neue Datenbanktabelle (siehe Abbildung 5.5c) so realisiert, dass auch später noch weitere Aufgabentypen mit eigenen Einstellungen hinzugefügt werden können, ohne dass am PHP-Code etwas geändert werden muss. Dafür muss nur im Ordner `administrator/components/com_pss/models/forms` des Servers eine Datei namens `tasktype.<Aufgabentyp-Name>.xml` hinzugefügt werden, in dem die Optionen nach den Vorgaben des Frameworks im `XML`-Format spezifiziert sind. Der Aufgabenmanager übernimmt dann das Rendern der Optionen im Formular und das Abspeichern in der Datenbank.

Außerdem wurde ein Button hinzugefügt, mit dem die gesammelten Sensordaten einer Erfassungsaufgabe gelöscht werden können, falls man eine Testreihe neu starten möchte. In Abbildung 5.3 sieht man ihn zusammen mit dem neuen Abschnitt für zusätzliche Einstellungen.

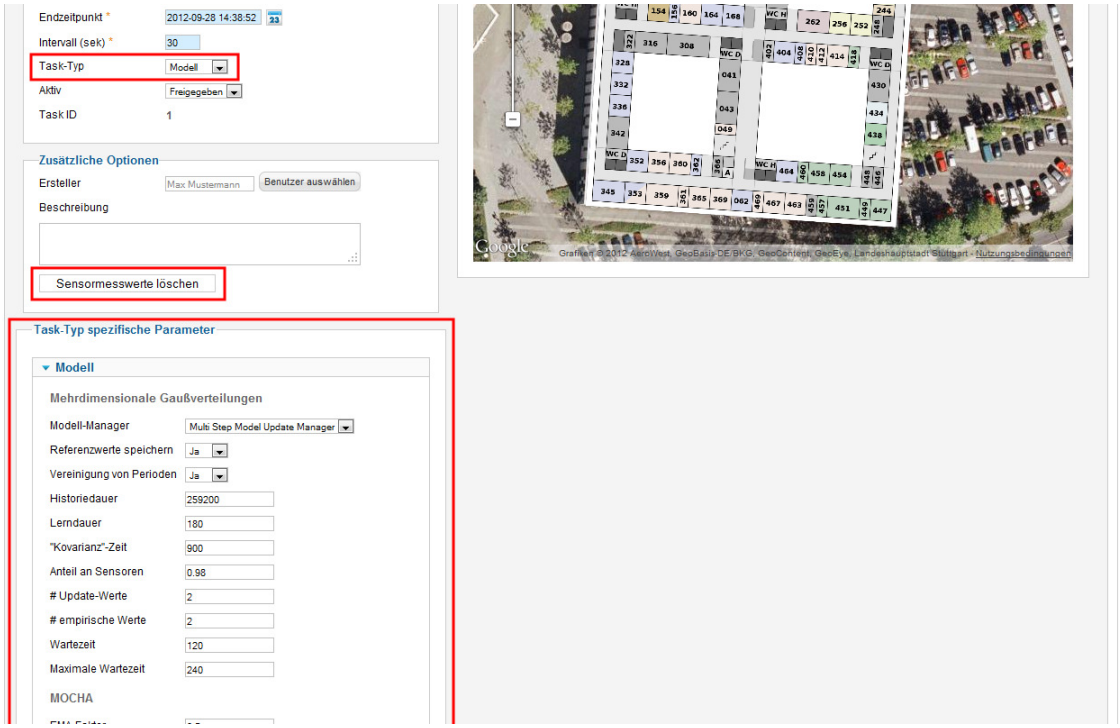


Abbildung 5.3.: Neue Optionen in der Ansicht zum Bearbeiten von Erfassungsaufgaben

Referenzwerte

Um Referenzwerte im Testbett verwalten zu können, wurde ein neuer Datentyp eingeführt, der in der Datenbank genau wie andere Typen, wie Licht oder Lautstärke, verwaltet wird. Auf diese Weise kann er in der GUI auch direkt neben den berechneten Werten der gleichen Runde angezeigt werden und es war keine weitere Datenbankänderung notwendig.

Ist die Option für das Aufzeichnen von Referenzwerten bei einer Aufgabe aktiviert, fordert der Java-Server unabhängig von der Auswahl des Modells immer Daten von allen Sensoren an. Das Umwandeln von einem normalen Datentyp in diesen zusätzlichen Typ wird von ihm am Ende der jeweiligen Periode erledigt. Dabei ändert er in der Datenbank den Typ aller Messwerte von deaktivierten Sensoren kurz bevor er der Modell-Applikation die Nachricht über das Periodenende übermittelt. Dies ist der Grund, warum der Java-Server jetzt auch in die Datenbank schreiben und nicht mehr nur aus ihr lesen muss.

5.2. Modell-Applikation

Die Modell-Applikation dient als Wrapper für den eigentlichen Modell-Code. Je nach Zustand greift sie auf die Datenbank zu und ruft die API-Funktionen des Modell-Codes auf. Des Weiteren kümmert sie sich um die Socket-Verbindung zum Java-Server.

Dabei kommen die vorhandenen Socket-Funktionen von C++ zum Einsatz, die bei Einbinden der jeweiligen h-Dateien problemlos sowohl unter Linux als auch unter Windows funktionieren. Für eine zuverlässige Verbindung wird *TCP* (Transmission Control Protocol) als Netzwerkprotokoll zwischen den Sockets eingesetzt

Für die Verbindung zur Datenbank wird die MySQL++-Bibliothek [You] eingesetzt. Es existieren zwar mehrere solcher Bibliotheken für C++, die zum Teil aufeinander aufbauen, aber leider nicht sowohl für Linux als auch für Windows entwickelt wurden bzw. schlicht nicht unter beiden Betriebssystemen liefen. Die einzige gefundene Bibliothek, bei der dies funktioniert hat, war die MySQL++-Bibliothek. Sie funktioniert ähnlich wie der JDBC-Treiber für *Java*.

5.2.1. Software-Architektur

Wie in 4.1.2 erläutert, wurden die Aufgaben der Modell-Applikation in verschiedene Bereiche aufgeteilt, für die jeweils eine Klasse angelegt wurde. Die Klassenhierarchie ist in Abbildung zu sehen. Die Applikation wurde so implementiert, dass sie sowohl das einfache Modell als auch das adaptive Modell aus [PSDR12] unterstützt.

Die Klasse ServerConnector

Die Klasse ServerConnector verwaltet die TCP-Socket-Verbindung zum Java-Server und baut diese direkt nach dem Start des Modells über den per Kommandozeilen-Parameter mitgelieferten Port auf. Sie wird in der main-Funktion des Programms instanziiert. Als Adresse wird localhost verwendet, da der Java-Server die Modell-Applikation als Prozess auf dem gleichen Rechner startet. Außerdem wird mit der ebenfalls per Kommandozeilen-Parameter übergebenen Aufgaben-ID ein Objekt der Klasse Model instanziiert, von dem die Methode onPrepareRound aufgerufen wird, damit Vorbereitungen für die erste Periode getroffen werden. Jeweils nach Empfangen der Nachricht ROUND_FINISHED ruft der ServerConnector die Methode onRoundFinished und danach die Methode onPrepareRound auf. Sollte die TCP-Verbindung zusammenbrechen oder wird die Nachricht TASK_FINISHED empfangen, beendet der ServerConnector die Modell-Applikation.

Die Klasse Model

Die Klasse Model ist für den Zugriff auf die Datenbank zuständig. So markiert sie in onPrepareRound die ausgewählten Sensoren in der Datenbank, fragt in onRoundFinished die effektiven Messwerte der Periode ab und speichert die berechneten Sensorwerte mit data_virtual = 1 dort ab. Die Informationen dafür erhält sie von einem Objekt der Klasse ModelConnector. Die Model-Klasse übernimmt außerdem das in 4.1.2 erläuterte optionale Vereinigen von Perioden, indem sie in Lernphasen die vergangenen Perioden mitzählt und jeweils nach drei Perioden die Messwerte der zwei vorherigen Perioden der dritten Periode zuspricht. Dies wird durch eine Änderung der Zeitstempel in der Datenbank realisiert. Sind

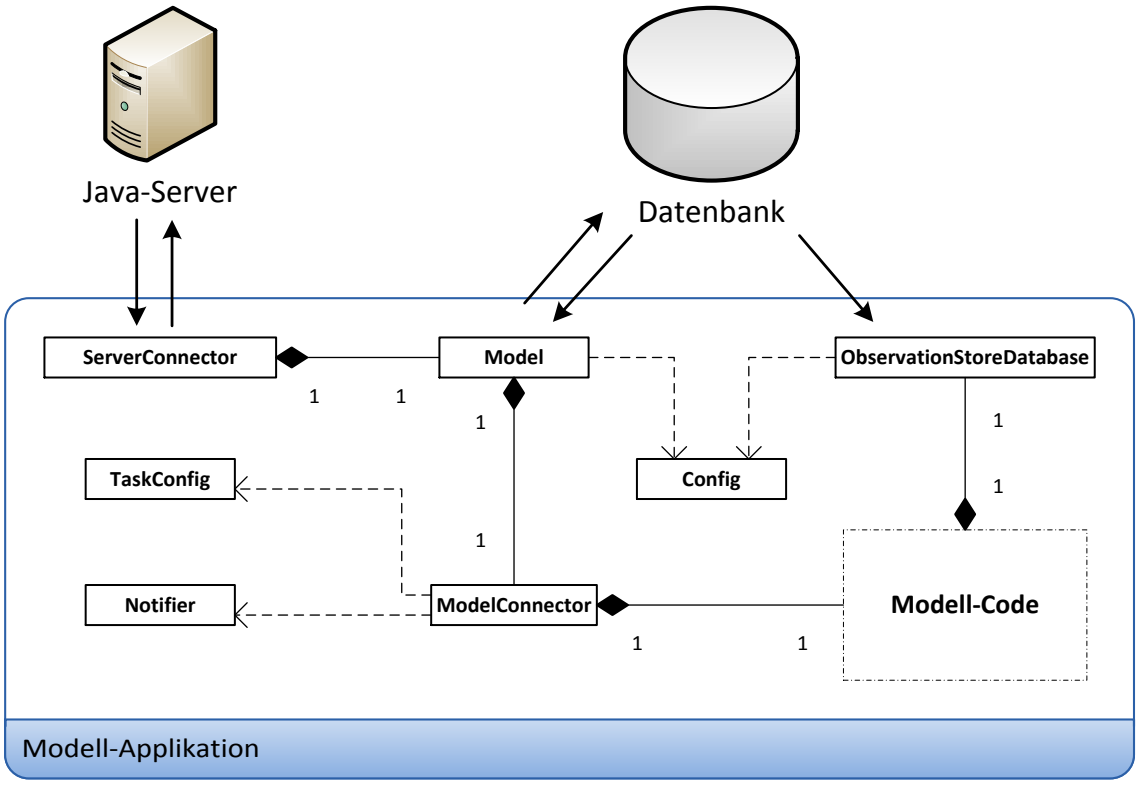


Abbildung 5.4.: Klassenhierarchie der Modell-Applikation mit Beziehungen zu Java-Server und Datenbank

zum Beispiel zwei mobile Knoten im Einsatz, können damit sechs virtuelle Sensoren pro effektiver Periode für das Modell abgedeckt werden.

Die Klasse ModelConnector

Beim Instanziieren des `ModelConnectors` wird zunächst mit Hilfe der Aufgabeneinstellungen der Modell-Code initialisiert. Die Einstellungen werden dabei mit der Hilfsklasse `TaskConfig` zur Verfügung gestellt. Danach steht das `ModelConnector`-Objekt zur Verfügung, um ausgewählte Sensoren abzufragen, sowie um Inferenzen und Validitätsprüfungen durchführen zu lassen. In Lernphasen werden dabei einfach alle vorhandenen Sensoren zurückgegeben und in Optimierungsphasen die Funktion des Modells zur Auswahl von Sensoren aufgerufen und dessen Ergebnis zurückgegeben. Der `ModelConnector` merkt sich dabei stets, welche Sensoren zu welchem Zweck ausgewählt wurden, um am Ende der Runde die jeweils richtigen empfangenen Sensormesswerte, die vom `Model`-Objekt aus der Datenbank geholt und übergeben werden, für die Inferenz und die Validitätsprüfung des Modells zu verwenden.

Eine weitere Funktion stellt dabei sicher, dass wenn möglich immer mindestens ein Kontrollwert zur Verfügung steht, der für die Überprüfung des Modells verwendet werden kann. Falls kein Kontrollwert aufgenommen werden konnte, aber mindestens zwei effektive Messwerte vorhanden sind, wird einer davon zufällig ausgewählt und in einen Kontrollwert umgewandelt. Wenn ursprünglich mehrere Kontrollwerte angefordert wurden, können auch mehrere effektive Messwerte umgewandelt werden, aber nie mehr als die Hälfte von ihnen. Optional kann diese Funktion auch sicherstellen, dass mindestens ein effektiver Messwert vorhanden ist, indem sie einen Kontrollwert in einen effektiven Wert umwandelt, wenn nur Kontrollwerte erhalten wurden.

Außerdem ist der `ModelConnector` dazu imstande, statistische Daten über jede Runde in Log-Dateien zu schreiben (z. B. wie viele Sensoren ausgewählt wurden und wie viele davon verfügbar waren).

Die Klasse ObservationStoreDatabase

Zum Zugriff auf die Datenbank für die Abfrage aller aufgenommenen Messwerte bis zu einem bestimmten Alter wurde die Klasse `ObservationStoreDatabase` geschrieben. Dies war nötig, da der Modell-Code im Gegensatz zur Abfrage der Messwerte am Ende jeder Periode (vgl. Klasse `Model`) dafür direkt auf den Ort zugreift, an dem alle Messwerte gespeichert werden, dem sogenannten *ObservationStore*. In den simulierten Umgebungen war das bisher eine Klasse, die die Daten während der Ausführung selbst verwaltet. Nun muss dafür ein Zugriff auf die Datenbank durchgeführt werden. Dafür erbt die Klasse `ObservationStoreDatabase` vom eigentlichen ObservationStore und greift in den Methoden zur Abfrage von Messwerten auf die Datenbank zu. Damit waren im Modell-Code praktisch keine Änderungen durchzuführen; die betroffenen Methoden im ursprünglichen ObservationStore mussten dort nur zusätzlich als „virtual“ markiert werden.

Hilfsklassen

Die Klasse `Config` wurde hinzugefügt, um globale Konfigurationseinstellungen, wie zum Beispiel die Zugangsdaten zur Datenbank, über statische Funktionen zur Verfügung zu stellen. Dazu greift die Klasse auf die gleiche `ini`-Datei zu, die auch der Java-Server für diese Daten verwendet.

Für Einstellungen, die für jede Erfassungsaufgabe unterschiedlich aussehen können, gibt es eine andere Klasse namens `TaskConfig`. Sie liest die Aufgabeneinstellungen aus der Datenbank aus und stellt sie genauso zur Verfügung, wie die `Config`-Klasse.

Die Klasse `Notifier` implementiert die in 4.1.2 vorgestellte Audio-Signalisierung und stellt dafür eine statische Funktion zur Verfügung, mit deren Parameter das auszugebende Signal festgelegt werden kann. Verwendet wird diese Funktion im `ModelConnector`, da dieser die jeweiligen Phasen-Übergänge anhand der Statusänderungen des Modells erkennen kann. Zu beachten ist, dass die Signalisierung nur stattfindet, wenn die Option zur Vereinigung

der Perioden aktiv ist, da die Wechsel zwischen den Phasen nur dann interessant für den Benutzer sind. Zudem wird diese Option wohl nur beim Demonstrator benötigt, bei dem der Rechner, der das Audio-Signal ausgibt, in der Nähe des Benutzers steht. In anderen Fällen hat also auch das Signal für einen aufgetretenen Fehler keinen Nutzen. Um die durch Beep-Codes realisierten Signale sowohl unter Windows als auch unter Linux verwenden zu können, mussten Präprozessor-Weichen für den Aufruf der entsprechenden Beep-Funktionen eingebaut werden, da dafür keine allgemeingültigen `C++`-Funktionen existieren.

5.3. Testserver für Simulationen

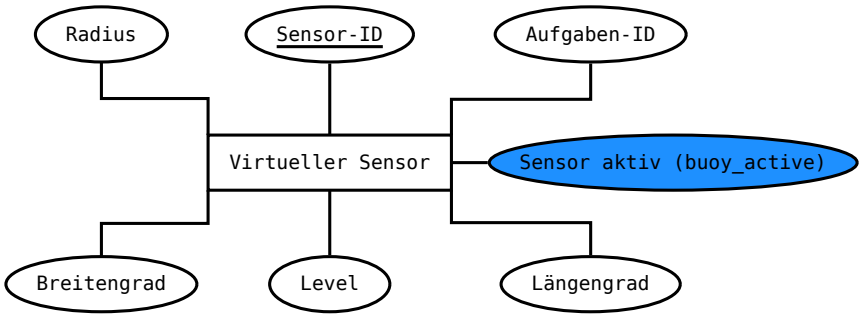
Der ebenfalls in Java implementierte Testserver übernimmt einige Klassen des Java-Servers: Die Hilfsklassen zum Auslesen und Bereitstellen der Datenbankzugangsdaten aus einer `ini`-Datei, die Klassen zur Repräsentation von virtuellen Sensoren und Erfassungsaufgaben sowie die Klasse, die die Anbindung zur Datenbank abstrahiert. Dazu kommt zum einen eine Klasse mit der `main`-Methode, in der alle speziell für den Testserver notwendigen Einstellungen aus den Kommandozeilen-Parametern gelesen werden und in der der Socket-Server für die Verbindung zur Modell-Applikation erstellt wird, und zum anderen eine Klasse, in der nach dem Verbindungsaufbau eine einfache Schleife zur Abarbeitung der Perioden läuft.

In jedem Schleifendurchlauf wird dabei zunächst auf die Nachricht `CALCULATION_FINISHED` gewartet, bevor die Messdaten der virtuellen Sensoren in die Datenbank geschrieben werden. Um diesen Vorgang zu vereinfachen, wird vorausgesetzt, dass der Test-Datensatz bereits in einer zweiten Tabelle vorliegt, aus der die Daten zum richtigen Zeitpunkt nur kopiert werden müssen. Je nachdem, welche Sensoren vom Modell aktiviert wurden, werden Daten als normale Werte kopiert oder als Referenzwerte eingetragen. Da Zeitstempel in den Tabellen des Realwelt-Testbetts nur mit einer Auflösung von einer Sekunde gespeichert werden, achtet der Testserver mit `Thread.sleep()`-Aufrufen darauf, dass eine Periode mindestens eine Sekunde lang andauert. Danach wird die Nachricht `ROUND_FINISHED` an die Modell-Applikation gesendet.

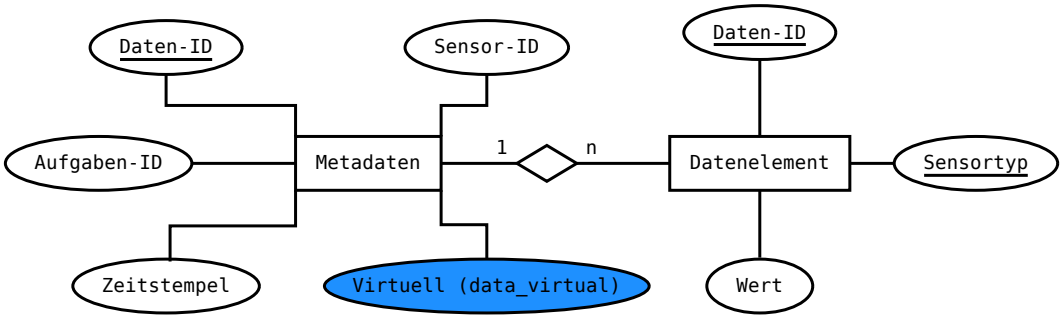
Die zweite Tabelle ist nur eine Hilfstabelle mit drei Spalten (vgl. Abbildung 5.5d). Je nachdem in welcher Form ein Test-Datensatz vorliegt, kann ein Skript geschrieben werden, das Sensormesswerte zusammen mit einer Sensor-ID und der Periodennummer, zu dem der Wert gehören soll, in diese Hilfstabelle einfügt.

5.4. Demonstrator

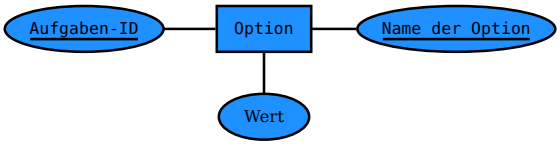
Der Demonstrator bildet das Systemmodell in minimaler Ausführung ab. Die dafür verwendete Hardware und der Aufbau werden im folgenden Abschnitt beschrieben. Danach werden die Location-Provider vorgestellt, die für die NFC-Lokalisierung und die statische Positionierung mit der Android-App entwickelt werden mussten.



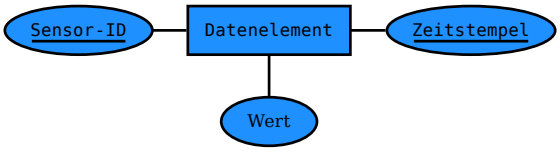
(a) Schema der Tabelle für virtuelle Sensoren



(b) Schema der Tabellen für Messwerte. Gleichzeitig aufgenommene Messwerte verschiedener Sensoren werden mit einem Satz Metadaten verwaltet.



(c) Schema für die neue Tabelle der aufgabenspezifischen Einstellungen



(d) Schema der Hilfstabelle für den Testserver

Abbildung 5.5.: Datenbankschemata geänderter und neuer Tabellen, neu hinzugekommene Spalten und Tabellen sind blau markiert.

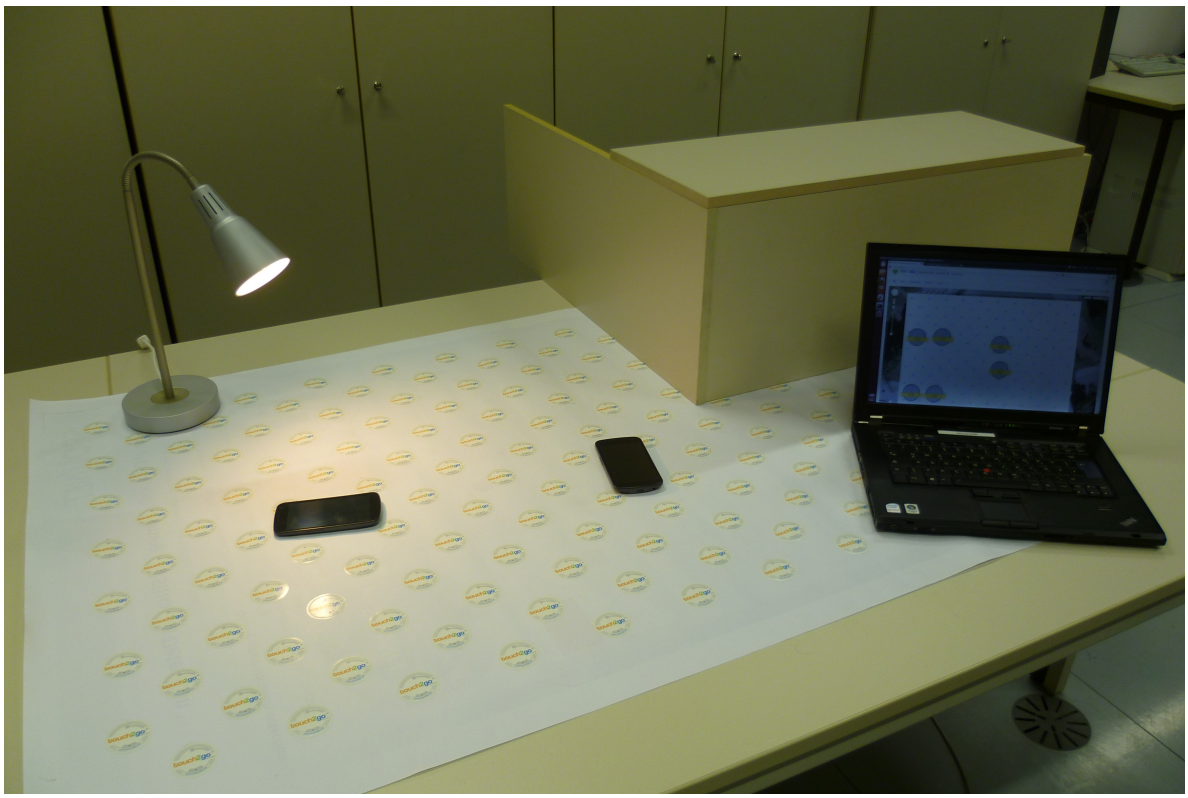


Abbildung 5.6.: Der Demonstrator im Einsatz

5.4.1. Hardware

Der PHP-Server, der Java-Server und das Modell laufen auf einem Laptop mit Netzwerkanbindung. Als mobile Knoten kommen mindestens zwei Smartphones mit NFC-Lesegerät zum Einsatz, die durch das Testgebiet bewegt werden können. Als Testgebiet wurde ein Plakat des Formats DIN-A0 nach dem in Abbildung 4.5 dargestellten Muster mit NFC-Tags beklebt. Da nur eine begrenzte Anzahl an Tags zur Verfügung stand, musste der Abstand zwischen den Tags im Vergleich zum optimalen Abstand etwas vergrößert werden. Die Wahrscheinlichkeit, dass ein Smartphone in genau einer der Positionen auf das Plakat gelegt wird, in denen es keinen Tag erkennen kann, ist aber sehr gering, da das Plakat trotzdem sehr gut abgedeckt ist.

Um verschiedene Helligkeitswerte über das Testgebiet legen zu können, kommen ein oder zwei Schreibtischlampen sowie eine kleine Vorrichtung zur Abschattung eines Teilgebietes zum Einsatz. Der gesamte Demonstrator ist in Abbildung 5.6 zu sehen.

5.4.2. Location-Provider für NFC-Lokalisierung

Der LocatingService der Android-App instanziiert je nach Einstellung eine konkrete Location-Provider-Klasse, die das Interface `ILocationProvider` implementieren muss, und übergibt sich selbst als Objekt. Auf diese Weise kann der Location-Provider nach erfolgreicher selbstständiger Positionierung die Methode `updatePositionData` des Services aufrufen, um die neue Position im System bekannt zu machen. Danach muss auch die Methode `processPositioning` aufgerufen werden, in der die Position an den Server übermittelt wird.

Kann ein Location-Provider wie die entworfene NFC-Lokalisierung (oder z. B. auch die Bluetooth-Lokalisierung) seine Position nicht selbstständig bestimmen, muss nur die Methode `processPositioning` aufgerufen werden, der noch zusätzliche Informationen übergeben werden können, die dem Server übermittelt werden sollen. Kann der Server damit die Position bestimmen, wird sie danach automatisch über `updatePositionData` dem System bekannt gemacht.

Auf den NFC-Tags wurde für die Lokalisierung jeweils die URL

```
psdemo:positioning/tag?id=ID
```

gespeichert, wobei `ID` eine aufsteigende Zahlenfolge zur eindeutigen Identifizierung der Tags ist. Das Präfix `psdemo` wird dafür verwendet, bei Erkennen eines Tags das Handling von ihm an die Public-Sensing-App zu übergeben. Die NFC-Funktionen von Android machen das über eine einfache Angabe in der Installationsdatei möglich. Damit wird die Public-Sensing-App nur dann über einen neuen Tag informiert, wenn es sich um einen Tag für die NFC-Lokalisierung handelt.

Da in Android Services nicht dafür verwendet werden können, auf NFC-Tags zu reagieren, musste eine Applikation geschrieben werden, die den Inhalt eines Tags annimmt und ihn an den Location-Provider weiterleitet. Diese Applikation hat ansonsten keine Aufgaben und besitzt auch keine GUI. Sie wird direkt nach dem Weiterleiten wieder beendet, sodass der Benutzer nichts von ihr mitbekommt.

Das Dekodieren der Informationen auf dem Tag führt die Klasse `AbstractNFCLocationProvider` durch, die auch das Interface `ILocationProvider` implementiert. Für diese abstrakte Klasse wurden zwei erbende Klassen geschrieben, die die Positionierung über NFC statisch und flexibel durchführen können. Dafür überschreiben sie jeweils die abstrakte Methode `processNFCPositioning`, in der sie die erhaltenen Informationen des Tags verarbeiten:

- Der `FlexibleNFCLocationProvider` verwendet die auf dem Tag gespeicherte ID, um eine MAC-Adresse zu generieren, die über `processPositioning` an den PHP-Server übermittelt wird. Der Server sendet daraufhin mit Hilfe der in der Datenbank gespeicherten Information die jeweilige Position zurück an die App. Dieser Location-Provider wird momentan für den Demonstrator verwendet.

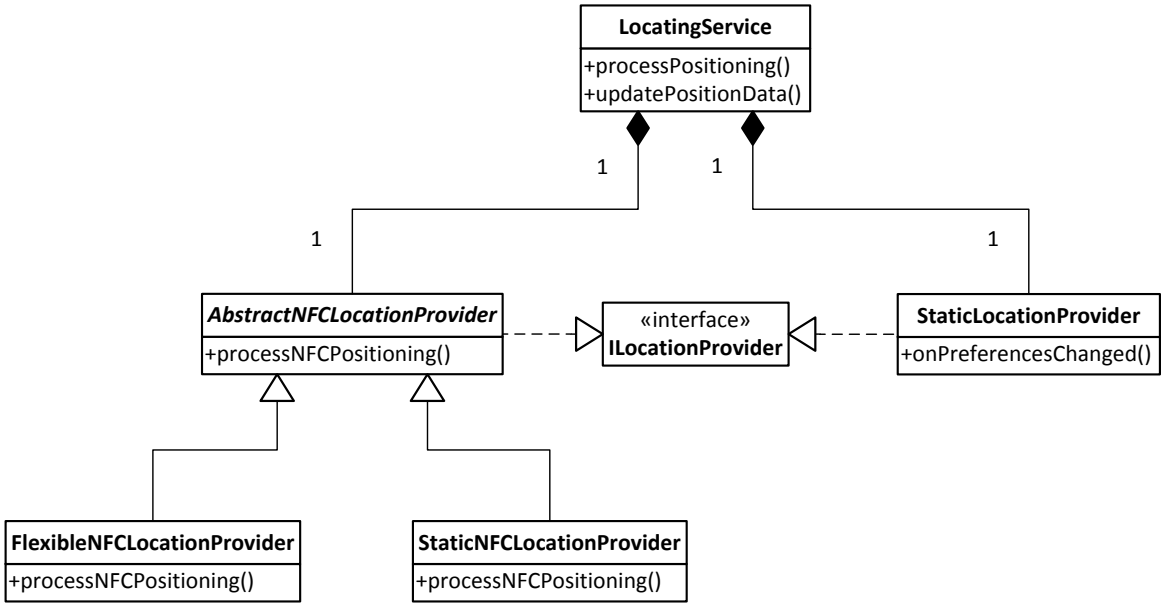


Abbildung 5.7.: Klassenhierarchie des LocatingService mit NFC- und statischer Positionierung

- Der `StaticNFCLocationProvider` erwartet dagegen, dass statt einer ID Längengrad, Breitengrad und die Ebenennummer direkt auf dem NFC-Tag gespeichert sind. Dadurch sind die Tags nicht so flexibel einsetzbar, die Positionierung kann aber ohne Hilfe des Servers durchgeführt werden.

In Abbildung 5.7 ist die entstandene Klassenhierarchie zu sehen, wobei die bereits vorhandenen Location-Provider für GPS- und Bluetooth-Positionierung ausgeblendet wurden. Zusätzlich eingetragen ist der im folgenden Abschnitt vorgestellte `StaticLocationProvider`.

5.4.3. Location-Provider für statische Positionierung

Um Geräte statisch an einer beliebigen Stelle positionieren zu können, wurden die Einstellungsmöglichkeiten der App erweitert, um dort Längengrad, Breitengrad und Ebenennummer oder die ID eines NFC-Tags eingeben zu können. Der `StaticLocationProvider` ist ein weiterer neuer Location-Provider, der die dort eingegebenen Daten für das Festlegen einer Position verwendet. Auf diese Weise ist es möglich, auch Smartphones ohne NFC-Lesegerät für den Demonstrator einzusetzen.

Die Funktion `onPreferencesChanged` des Location-Providers wird über die Android-API als Callback für Einstellungsänderungen registriert und wird damit automatisch aufgerufen,

wenn die Einstellungen der App geändert werden. Somit kann auch nach dem Start des LocatingServices die Position des Smartphones aktualisiert werden.

5.5. Bluetooth-Positionierung

Für eine bessere Bluetooth-Positionierung im Gebäude war es nötig, den Algorithmus zur Berechnung der Position anzupassen (siehe 4.3). Dafür musste die `_locate`-Funktion der Klasse `PssModelLocate` im PHP-Server so ergänzt werden, dass zunächst die empfangenen und bekannten Beacons absteigend nach der Signalstärke sortiert und dann alle Beacons bis auf die drei mit den größten Signalstärken verworfen werden. Die restliche Positionierung läuft danach wie bisher ab. Durch die zusätzlichen Beacons im Gang konnte die Bluetooth-Positionierung damit so weit verbessert werden, dass sie im abzudeckenden Teil des Gebäudes raumgenau funktioniert.

6. Evaluation

In diesem Kapitel soll die korrekte Integration der modellbasierten Erfassungsmethoden in das Realwelt-Testbett evaluiert werden. Dazu wurde zunächst ein simulierter Ablauf verwendet, um die korrekte Arbeit der Modell-Applikation zu bestätigen. Danach wurde das Realwelt-Testbett verwendet, um die modellbasierten Erfassungsmethoden mit dem Demonstrator zu validieren.

6.1. Modell-Applikation

Für die Simulation mit dem Testserver wurde der in [PSDR12] genutzte Intel-Lab-Datensatz [DGM+04] verwendet, der tatsächlich aufgenommene Temperaturwerte von 50 Sensoren über zehn Tage enthält. Die Messwerte wurden in variierenden Abständen von 10 bis 30 Sekunden aufgenommen, wobei nicht alle Sensoren jedes Mal einen Wert geliefert haben. Dadurch kann hier die Nicht-Verfügbarkeit von Sensoren mitgetestet werden.

Ein Skript hat die Daten in die Hilfstabelle für Simulationen der Datenbank übertragen, wobei die Sensornummern auf virtuelle Sensoren-IDs und die Zeitpunkte auf Periodennummern abgebildet wurden. Auf diese Weise kann der Testserver in jeder Periode die entsprechenden Temperaturwerte in die Datenbanktabelle für Messwerte kopieren.

Da jede Periode mindestens eine Sekunde dauerte, wurde für eine ganze Simulation mit Einbeziehung der Daten aller zehn Tage fast ein gesamter Tag benötigt. In Abbildung 6.1 sind die Ergebnisse des Test-Durchlaufs zu sehen. Für die Übersichtlichkeit wird nur ein Ausschnitt gezeigt.

Gezeichnet ist in blau der Verlauf der Durchschnittswerte von allen tatsächlich aufgenommenen Messwerten. Darüber liegt in lila der Verlauf der berechneten Werte, ebenfalls jeweils der Durchschnitt in jeder Periode. Zu erkennen ist, dass erst nach etwa 200 Runden berechnete Werte zur Verfügung stehen. Bis dahin befindet sich das Modell in der Lernphase. Zu diesem Moment entsteht auch ein Fehler zwischen den tatsächlichen Werten und den berechneten Werten (die roten Punkte im Diagramm), der größer wird, da die Durchschnittstemperatur immer weiter abnimmt.

Vermutlich morgens an einem Tag nimmt die Durchschnittstemperatur wieder zu und steigt stark an, wodurch der Fehler immer größer wird, bis der Validitätsprüfer befindet, dass das Modell nicht mehr valide ist. Durch eine kurze Lernphase passt sich das Modell dann wieder an. Dies passiert noch einmal bis der Fehler für einige Zeit sehr niedrig bleibt.

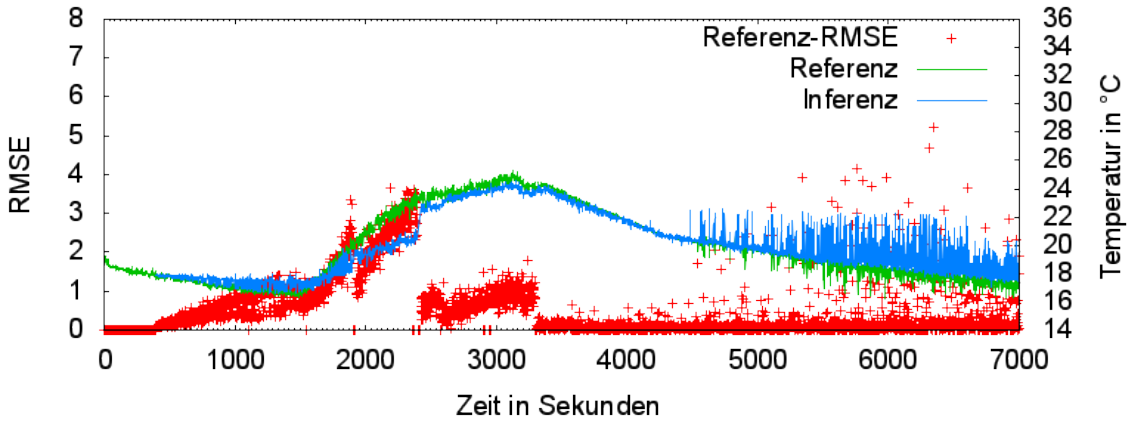


Abbildung 6.1.: Ausschnitt der Ergebnisse eines Test-Durchlaufs

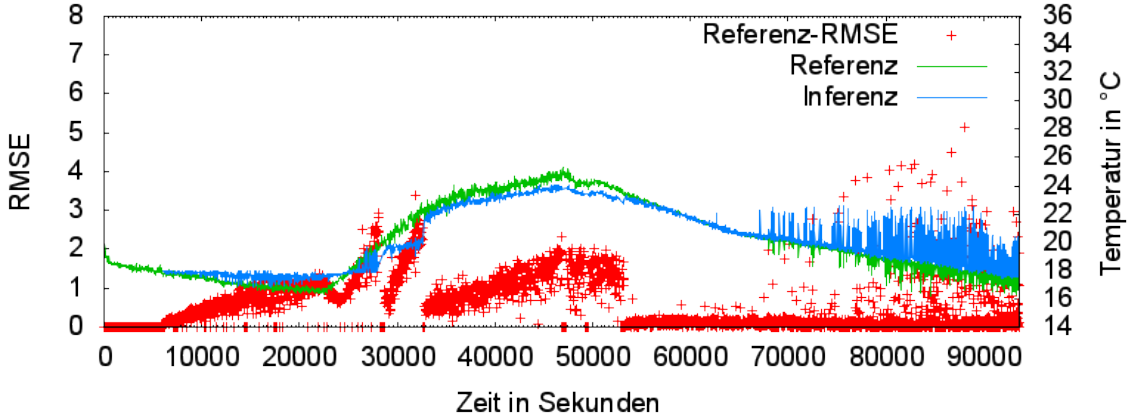


Abbildung 6.2.: Ausschnitt der Ergebnisse eines komplett simulierten Durchlaufs

In Abbildung 6.2 ist die Ausführung mit dem gleichen Test-Datensatz mit dem Simulator von [PSDR12] zu sehen. Man erkennt, dass der Verlauf sehr ähnlich aussieht wie der mit der Modell-Applikation. Um die Ergebnisse noch besser vergleichen zu können, werden für die Ausführung mit der Modell-Applikation die gleichen Metriken berechnet wie in [PSDR12]. Konkret sind das *Qualität* und *Effizienz*. Um die Qualität zu berechnen wird der mittlere quadratische Vorhersagefehler (engl. Root Mean Square Error, RMSE) mit einem festgelegten Schwellwert S verglichen, der auch für den Validitätsprüfer des Modell-Codes verwendet wird. Die Qualität ergibt sich dann aus dem Verhältnis der Anzahl von Perioden P in Optimierungsphasen, in denen der Schwellwert nicht überschritten wird, zu der Gesamtanzahl an Perioden in Optimierungsphasen (siehe Gleichung 6.1). Die Effizienz ist durch das Verhältnis von Periodenanzahl in Optimierungsphasen zur Gesamtanzahl aller Perioden definiert (siehe Gleichung 6.2).

(6.1)
$$\text{Qualität} = \frac{|\{P|P \text{ ist in Optimierungsphase} \wedge RMSE(P) \leq S\}|}{|\{P|P \text{ ist in Optimierungsphase}\}|}$$

(6.2)
$$\text{Effizienz} = \frac{|\{P|P \text{ ist in Optimierungsphase}\}|}{|\{P\}|}$$

Der Testserver wurde drei Mal mit den gleichen Modell-Einstellungen wie in [PSDR12] gestartet und die Ergebnisse gemittelt. In Tabelle 6.1 sind sie den Werten der komplett simulierten Umgebung von [PSDR12] gegenüber gestellt. Die Abweichungen dabei sind nur minimal, die Modell-Applikation arbeitet also wie erwartet.

	Qualität	Effizienz
Modell-Applikation	93,80%	86,20%
komplett simuliert	93,77%	85,54%

Tabelle 6.1.: Vergleich der Metriken von Modell-Applikation und der komplett simulierten Umgebung aus [PSDR12]

6.2. Demonstrator

Für den Demonstrator wurden zwei Szenarien entworfen, um einerseits zu zeigen, dass das Modell realitätsnahe Werte liefert und andererseits, dass der Validitätsprüfer eine Änderung der Umgebung erkennt und dann eine neue Lernphase einleitet. Bei beiden Szenarien werden Helligkeitswerte in *Lux* gemessen, die mit Hilfe von einer Lampe und einer Abschattung im Testgebiet beeinflusst werden.

6.2.1. Statisches Szenario

Das statische Szenario diente zunächst dafür, angemessene Parameter für den Modell-Code festzulegen. Dabei war vor allem das Finden eines passenden Schwellwertes für den Validitätsprüfer nötig, wofür zunächst Testwerte gesammelt wurden. Bei insgesamt sechs virtuellen Sensoren, von denen zwei durch eine Lampe erhellt und zwei durch eine Abschattung verdunkelt wurden, sind für jeden virtuellen Sensor jeweils 40 Messwerte aufgenommen worden. Für jeden Sensor wurden dann die Mittelwerte der gemessenen Helligkeiten berechnet und die Abweichung von diesen Mittelwerten bestimmt. Auf diese Weise konnte in Erfahrung gebracht werden, wie stark die von den Lichtsensoren des Smartphones abgegebenen Werte schwanken. Die Abweichungen vom Mittelwert wurden dabei in fünf Gruppen eingeteilt, die in Abbildung 6.3 auf der x-Achse zu sehen sind. Darüber aufgetragen ist, wie viele Messungen jeweils in diese Gruppen fallen. Daran kann abgelesen

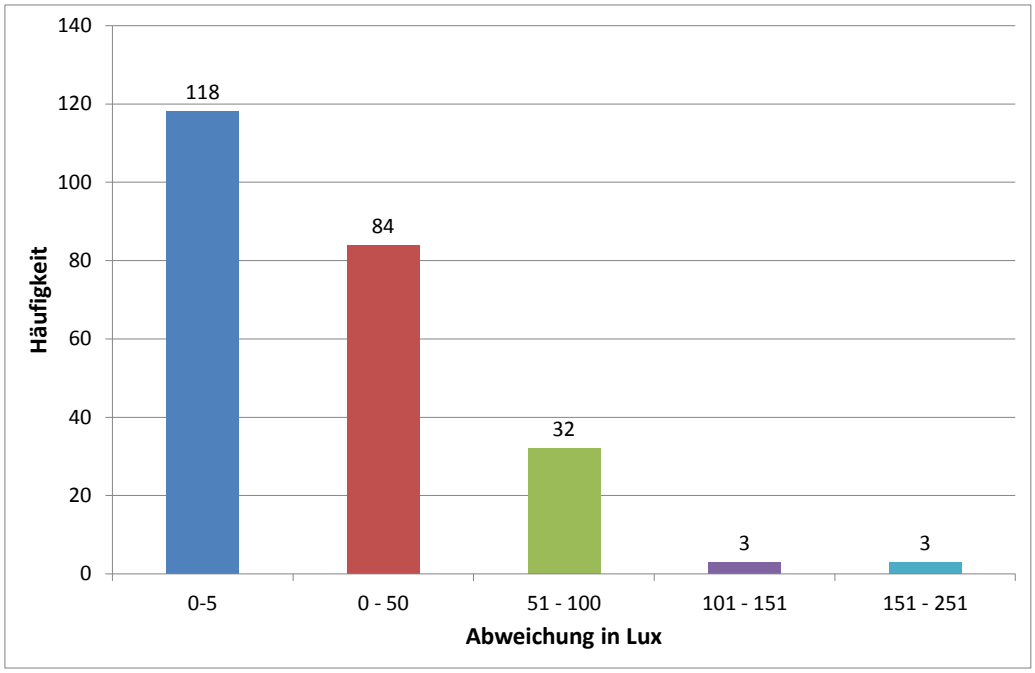


Abbildung 6.3.: Anzahl an Abweichungen für verschiedene Gruppen von den Mittelwerten

werden, dass die meisten Abweichungen nicht größer als 50 Lux sind. In den Bereich 51 bis 100 Lux Abweichung fallen nur wenige und darüber gibt es nur noch Ausreißer.

Betrachtet man den relativen Fehler, stellt man fest, dass er nahezu konstant ist. Also je heller es ist, desto mehr schwanken die Helligkeitswerte von den Lichtsensoren und desto größer muss der Schwellwert für den Validitätsprüfer sein.

Bei einer Aufgabenausführung mit Modell wurde das System so konfiguriert, dass die Wahrscheinlichkeit sehr groß ist, Referenzwerte aufzeichnen zu können. Dies ist möglich, wenn das Modell nur einen effektiven virtuellen Sensor sowie einen Kontrollsensor auswählen darf. Als effektiven Sensor wählt das Modell meist in jeder Runde den gleichen Sensor aus, sodass eines der beiden Smartphones an dessen Position gelegt werden kann. Bei insgesamt sechs virtuellen Sensoren ist die Wahrscheinlichkeit hoch, dass das andere Smartphone nicht an der Stelle des zufällig ausgewählten Kontrollsensors liegt, sondern bei einem anderen Sensor. Dort wird deshalb ein Referenzwert aufgenommen. Nach der Lernphase können in der GUI des PHP-Servers also Referenzwerte mit berechneten Werten verglichen werden. Dadurch, dass die Daten alle in der Datenbank abgespeichert werden, können sie auch später noch ausgelesen und genauer analysiert werden.

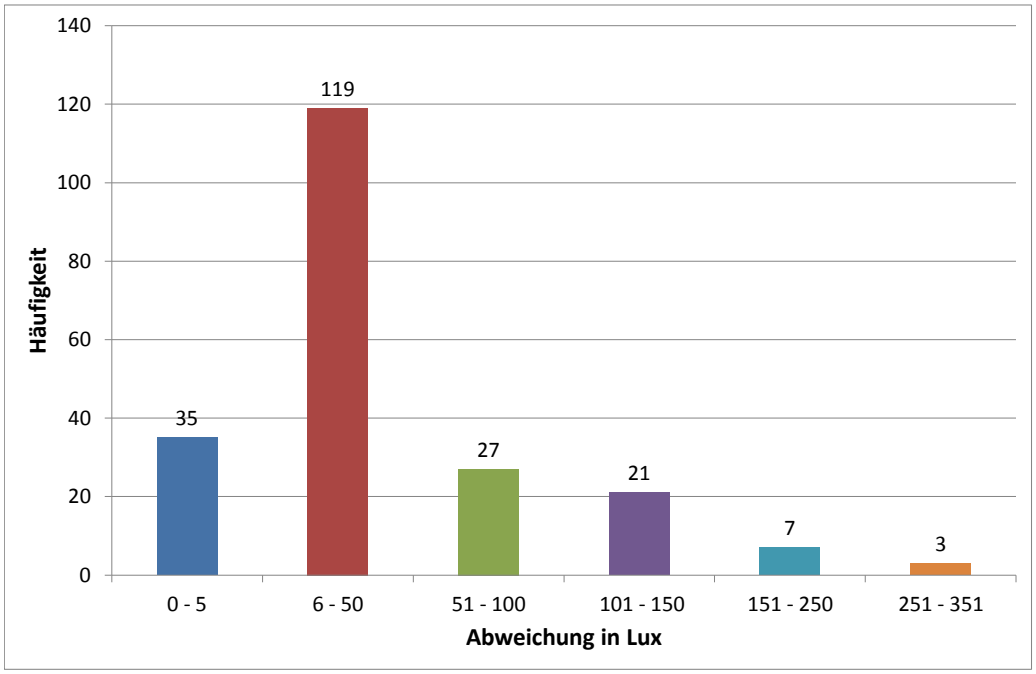


Abbildung 6.4.: Anzahl an Abweichungen für verschiedene Gruppen von den Referenzwerten

In diesem Fall wurden die Abweichungen der berechneten Werte von den Referenzwerten in Abbildung 6.4 übertragen und ebenfalls in Gruppen eingeteilt. Eine leichte Verschiebung zu größeren Abweichungen ist zu erkennen, sie bleibt aber im erwarteten Rahmen. Zu erklären ist das dadurch, dass jetzt neben den Fehlern bei der Messung auch noch die Fehler des Modells hinzukommen. Mit Hilfe des Schaubildes lässt sich sagen, dass für eine kontrollierte Ausführung, wie es hier der Fall war, der Schwellwert für den Validitätsprüfer zwischen 100 und 200 festgelegt werden sollte, um auf Dauer akzeptable Inferenzen vom Modell zu erhalten.

6.2.2. Dynamisches Szenario

Im dynamischen Szenario soll gezeigt werden, dass der Validitätsprüfer bei einer Änderung in der Umgebung erkennt, dass das momentane Modell nicht mehr gültig ist, und deswegen eine neue Lernphase einleitet. Danach sollte das Modell wieder akzeptable Werte liefern. Um die Änderung in der Umgebung zu simulieren, wurde die Lampe nach einiger Zeit im Testgebiet versetzt.

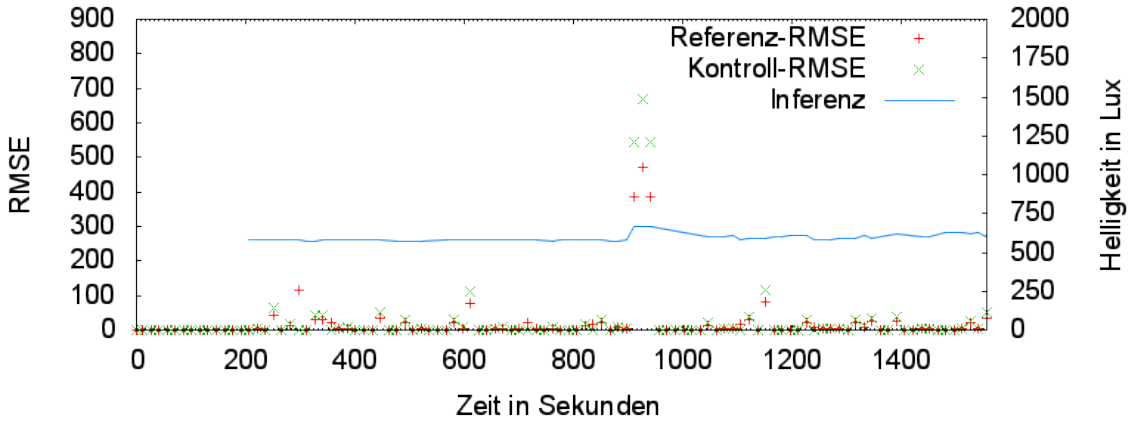


Abbildung 6.5.: Verlauf des dynamischen Szenarios

Für dieses Szenario wurden vier Kontrollwerte angefordert, um die Wahrscheinlichkeit zu erhöhen, dass in jeder Periode sowohl ein effektiver Messwert als auch ein Kontrollwert zur Verfügung standen. Da es sich um eine kontrollierte Umgebung handelte, wurde der Schwellwert auf 100 Lux festgesetzt. Kontrolliert bedeutet hierbei, dass die Smartphones jedes Mal, wenn sie zu einem virtuellen Sensor zurückkehren, wieder an genau der gleichen Stelle und in der gleichen Ausrichtung abgelegt werden müssen, da sonst aufgrund des kleinen Lichtpegels der Lampe schnell große Fehler in den Messungen entstehen.

In Abbildung 6.5 sind neben den Durchschnittswerten der Inferenzen der RMSE zu den Referenzwerten zu sehen und der RMSE, den der Validitätsprüfer des Modells mit Hilfe der Kontrollwerte berechnet. Die erste Optimierungsphase begann ungefähr bei Sekunde 200. Danach ist zu sehen, dass sowohl der Referenz-RMSE als auch der Kontroll-RMSE fast nie über den Schwellwert treten; das Modell liefert also akzeptable Werte. Bei Sekunde 900 wurde die Lampe von ihrer ursprünglichen Position zu den nicht abgedunkelten virtuellen Sensoren versetzt. Daraufhin verzeichnet das Modell drei große Fehler oberhalb des Schwellwertes in Folge, wodurch das System in eine neue Lernphase geht und bei Sekunde 1045 wieder in die Optimierungsphase. Das Modell hat sich damit an die neue Umgebung angepasst und liefert wie erwünscht wieder Werte unterhalb des Schwellwertes.

6.3. Fazit

Die Evaluation zeigt, dass die Integration der modellbasierten Erfassungsmethoden in das Realwelt-Testbett erfolgreich war. Durch den Demonstrator konnten ihre Funktionen sogar bereits mit realer Hardware bestätigt werden. Jetzt kann das Realwelt-Testbett verwendet werden, um verschiedenste Erfassungsaufgaben zu optimieren und damit die modellbasierten Erfassungsmethoden weiter zu verbessern und für die Realität anzupassen.

7. Zusammenfassung & Ausblick

Im Folgenden wird die vorliegende Arbeit zusammengefasst und ein Überblick über mögliche weitere Arbeiten gegeben, die darauf aufbauen.

7.1. Zusammenfassung

Mit dieser Arbeit wurden modellbasierte Erfassungsmethoden in ein Realwelt-Testbett für Public-Sensing-Systeme integriert, um sie in realer Umgebung evaluieren zu können. Dazu wurde der Modell-Code in eine Anwendung gebettet, die die Verbindung mit dem Realwelt-Testbett übernimmt. Hauptaufgaben dabei sind die Kommunikation über TCP-Sockets mit dem Public-Sensing-Server und die Möglichkeit des Zugriffs auf die Datenbank, um dort Sensorwerte und andere Informationen zur Ausführung von Erfassungsaufgaben zu verwalten.

Zur Überprüfung, ob das Modell auch in diesem Kontext richtig arbeitet und die Implementierung der Anwendung und der Verbindung zum Server korrekt ist, wurde ein Testserver entwickelt, mit dem größere Datensätze schnell abgearbeitet und mit den Ergebnissen von Simulationen des Modell-Codes verglichen werden können.

Um das Realwelt-Testbett zusammen mit der Modell-Anbindung im kleinen Rahmen präsentieren zu können, wurde in diesem Zuge auch ein Demonstrator entwickelt, der das Public-Sensing-System auf die Größe eines Tisches verkleinert. Dafür wurde eine Lokalisierung basierend auf NFC-Tags entwickelt, um Smartphones auf wenige Zentimeter genau auf dem Tisch positionieren zu können. Auf diesem Weg wurden Ergänzungen und Verbesserungen an der GUI des Public-Sensing-Servers vorgenommen, die auch anderen Anwendungen des Realwelt-Testbetts zu Gute kommen können: Zum Beispiel werden die gesammelten Sensordaten jetzt übersichtlicher dargestellt und es ist zudem möglich, mit einem Regler ältere Daten anzeigen zu lassen.

Mit dem Demonstrator wurden erfolgreich zwei Szenarien mit Helligkeitsmessungen durchgespielt, die die Funktionen des Modells zeigen. Beim statischen Szenario wurde mit Hilfe von aufgenommenen Referenzwerten gezeigt, dass das Modell in der Optimierungsphase plausible Werte für die Umgebung liefert. Mit dem dynamischen Szenario wurde dagegen demonstriert, wie das Modell Änderungen in der Umgebung erkennt und sich durch neue Lernphasen daran anpasst.

7.2. Weiterführende Arbeiten

Aus Mangel an genügend Smartphones und Freiwilligen konnte das System noch nicht im Informatik-Gebäude verwendet werden, um zum Beispiel Helligkeits- oder Lautstärkewerte aufzunehmen und dafür Modelle zu lernen. Sollten sich genügend Freiwillige mit eigenen Smartphones finden, die ihre Geräte einige Zeit lang im zweiten Stock des Gebäudes umhertragen, kann das System mit Hilfe der Bluetooth-Positionierung dort getestet werden.

Damit das Public-Sensing-System auch in noch größerem Maßstab getestet wird, sollte es in Zukunft zum Beispiel auch auf dem Campus verwendet werden. Dort findet die Lokalisierung dann mit Hilfe von GPS statt.

Um die Positionierung innerhalb des Gebäudes noch genauer zu gestalten, ist geplant, die Bluetooth-Positionierung mit bereits laufenden Arbeiten über Innenraumpositionierung mit Hilfe von WLAN-Fingerprinting zu kombinieren. Man verspricht sich dadurch eine noch bessere Innenraumlokalisierung als nur raumgenau zu erreichen.

Im Laufe dieser Arbeit hat sich herausgestellt, dass verschiedene Smartphone-Hersteller oder auch verschiedene Serien einer Marke auf stark unterschiedliche Hardware für die Sensoren zurückgreifen. Dies hat zur Folge, dass Sensorwerte von verschiedenen Geräten leider nicht direkt miteinander verglichen werden können. Um diesem Problem vorzubeugen, müssen Ideen entwickelt und umgesetzt werden, wie unterschiedliche Hardware trotzdem gemeinsam in einem System verwendet werden kann.

A. Anhang

A.1. HowTo zum Einrichten des Testbetts auf einem Linux-System

Hier werden die Schritte aufgelistet, um den Server des Demonstrators auf einem Linux-System (Ubuntu) einzurichten. Dabei wird zunächst erklärt, wie die benötigten Komponenten zum Kompilieren der Modell-Applikation installiert werden. Zeilen, die im Terminal auszuführende Befehle beinhalten, werden durch ein vorangehendes $-Zeichen markiert. Die Anleitung ist in Englisch gehalten, um sie vielen Benutzern zugänglich zu machen.

A.1.1. Setting up the Development Environment

1. Installing requirements and model code from repository https://ipvs.informatik.uni-stuttgart.de/pubsens/repos:
   ```
   $ aptitude install subversion
   $ aptitude install g++
   $ aptitude install make
   $ aptitude install libmysql++-dev
   $ aptitude install fftw3-dev
   $ aptitude install gfortran
   $ aptitude install liblapack-dev (should automatically also install libblas-dev)
   $ aptitude install libitpp-dev
   $ cd to arbitrary directory
   $ svn co [...]/branches/GausseanProcessModel-altpk GausseanProcessModel
   $ svn co [...]/trunk/PublicSensingTestbed/Model Model
   $ cd GausseanProcessModel/Make
   $ make
   $ cd ../../Model/Make
   $ make
   ```
2. Reboot the system.
3. Testing the environment:
   ```
   $ cd Model/Make
   $ ./psmodel.exe
   ```
   If above command doesn't complain about missing shared object files you have successfully set up the environment! (You should see the output „No task ID found“ (Usage: ./psmodel.exe <Task ID> <Socket Port>)

A.1.2. Setting up the Public Sensing Server

1. Installing MySQL and Apache with PHP and MySQL module:
   ```
   $ aptitude install mysql-server mysql-client
   $ aptitude install apache2 php5
   $ aptitude install php5-mysql php5-curl php5-json
   ```
2. Change user and group in `/etc/apache2/envvars`:
   APACHE_RUN_USER=<your user name (publicsensing?)>
   APACHE_RUN_GROUP=<your user group (publicsensing?)>
3. Change upload_max_filesize in /etc/php/apache2/php.ini (should be the same as post_max_size, at least 8M).
4. Restarting the Apache server:
   ```
   $ /etc/init.d/apache2 restart
   ```
5. Adding new group so that your user is able to modify files of web server:
   ```
   $ groupadd www
   $ adduser <your user name> www
   $ chgrp www /var/www
   $ chmod g+w /var/www
   ```
6. Log out and in again.
7. Download latest version of Joomla! 2.5 from http://www.joomla.org (or from http://www.jgerman.de for package with English and German language files) and unpack it in /var/www/ (with your user!, not root).
8. With your favourite browser visit http://localhost and follow the instructions for installing Joomla!.
9. Go on with the documentation of Public Sensing Testbed.
10. Copy psmodel.exe into /var/www/media/pss/java/ for attaching the PSModel to the Java server.
11. Change permission of psmodel.exe so that it can be executed by the owner.
12. The following two lines are unnecessary if the development environment (see top of this document) has been set up correctly:
    a) Copy shared library objects into an arbitrary directory.
    b) Edit cron job to execute 'export LD_LIBRARY_PATH=<shared library objects directory>;' afore calling the PS cron job.
13. Setting up beeps:
    ```
    $ aptitude install beep
    $ modprobe pcspkr
    ```
    Remove `blacklist pcspkr` out of file `/etc/modprobe.d/blacklist.conf`.

A.2. Dokumentation zur Durchführung der Szenarien

Hier wird kurz zusammengefasst, wie der Demonstrator für das *statische* und das *dynamische Szenario* eingerichtet wird. Außerdem werden die Durchführungen der beiden Szenarien erläutert und die Beep-Codes der Modell-Applikation dokumentiert.

A.2.1. Einrichtung

Wenn man im Backend eingeloggt ist, kann man durch Anhängen von `&reset=1` an die URL das System in einen definierten Zustand zurücksetzen. Dabei werden alle selbst eingetragenen und gesammelten Daten gelöscht! Im Moment besteht der definierte Zustand aus den 140 NFC-Beacons für die Lokalisierung im Demonstrator sowie dem statischen und dem dynamischen Szenario mit den Einstellungen von Tabellen A.1 und A.2.

Da beim *statischen Szenario* jeweils nur ein Messwert für die Inferenz und ein Kontrollwert angefordert werden und 'ensurephysicalreading' auf 'Ja' steht, wird man in den meisten Fällen einen Referenzwert (GroundTruth) aufnehmen können und diesen in der Karte sehen.

Durch die hohe Anzahl an Kontrollwerten und 'ensurephysicalreading' beim *dynamischen Szenario* ist sichergestellt, dass in fast jeder Runde sowohl ein Wert für die Inferenz als auch ein Kontrollwert zur Verfügung stehen. Der Threshold = 100 ist ausgelegt auf eine *sehr* kontrollierte Umgebung. In anderen Fällen sollte er auf 200 oder noch mehr erhöht werden.

Bei beiden Szenarien werden jeweils 6 virtuelle Sensoren in 2er-Gruppen angelegt. Über eine der 2er-Gruppen wird eine Lampe aufgestellt, eine andere wird möglichst abgedunkelt.

Kurz vor dem Start eines Durchlaufs muss außerdem sichergestellt werden, dass die Smartphones eine aktive Internet- oder WLAN-Verbindung zum Server haben, ihre Services laufen und dass sie am Server angemeldet sind (dazu genügt es, auf 'Handy am Server anmelden' zu drücken und den Status der Services abzurufen).

Danach kann man per Aufgabenmanager eine der Erfassungsaufgaben starten.

A.2.2. Durchführung

In den Lernphasen müssen die Smartphones über die virtuellen Sensoren wandern, wobei jeweils drei Perioden zu einer zusammengefasst werden, um alle sechs virtuellen Sensoren für das Modell zum „gleichen" Zeitpunkt abzudecken. Damit dies gut funktioniert, sollte man darauf achten, die Smartphones immer direkt nach der Messwertabgabe (Anzeige „Verbindung aufgebaut!" auf dem Display) zu verschieben und keine Runde zu verpassen.

In den Optimierungsphasen wählt das Modell zumeist einen der hellsten Messpunkte aus. Eines der Smartphones sollte deswegen dort liegen bleiben, während das andere frei über das Testgebiet wandern kann.

Option	Wert
acceptableviolations	5
aggregativelearningphase	Ja
countcontrol	1
countreading	1
emafactor	0.5
empiricalcovariancematrixsave	
ensurephysicalreading	Ja
fracnodesforlearning	0.98
historyduration	259200
maxwaittime	240
modelmanager	Multi Update
numempiricalreadings	2
numupdatereadings	2
readinglogfilename	StaticReadingLog
rounds	1
statlogfilename	StaticStatLog
storegroundtruth	Ja
targetmaxvariance	100
threshold	100
waittime	120
windowsize	10

Tabelle A.1.: Einstellungen des statischen Szenarios

Statisches Szenario

Während den Optimierungsphasen sollte man auf dem Laptop in der Karte die Referenzwerte mit den berechneten Werten vergleichen können, ohne dass eine neue Lernphase eintritt.

Dynamisches Szenario

Nach dem Beginn der Optimierungsphase sollte das Modell gute Inferenzen liefern. Zu einem beliebigen Zeitpunkt sollte die Lampe von seiner ursprünglichen Stelle zu den nicht abgedunkelten virtuellen Sensoren umgestellt werden. Wenn dort dann auch ein Smartphone platziert wird, sollte MOCHA die Änderung ziemlich schnell erkennen und eine neue Lernphase einleiten. Hier darf man nicht vergessen, dass die Smartphones sofort wieder kontinuierlich bewegt werden müssen, um alle virtuellen Sensoren abzudecken (genau wie in der ursprünglichen Lernphase). Diese Lernphase sollte jetzt nicht länger dauern, als dass man jeden virtuellen Sensor genau zwei Mal besuchen kann (zumindest mit den oben genannten Einstellungen).

Option	Wert
acceptableviolations	2
aggregativelearningphase	Ja
countcontrol	4
countreading	-1
emafactor	0.5
empiricalcovariancematrixsave	
ensurephysicalreading	Ja
fracnodesforlearning	0.98
historyduration	259200
learningduration	100
modelmanager	Simple Update
readinglogfilename	DynamicReadingLog
rounds	1
samplecovariancetime	100
statlogfilename	DynamicStatLog
storegroundtruth	Ja
targetmaxvariance	100
threshold	100
windowsize	5

Tabelle A.2.: Einstellungen des dynamischen Szenarios

A.2.3. Beep-Codes

Die genauen Daten zur jeweiligen Frequenz, Länge und Anzahl findet man in der Datei 'util/Notifier.cpp' der Modell-Applikation. Das Modell gibt nur dann Beeps aus, wenn die Option 'aggregativelearningphase' auf 'Ja' steht, da dies normalerweise beim Demonstrator der Fall ist.

- Start des Modells: Ein langer Beep mit etwas höherer Frequenz
- Eintritt in die Lernphase: Fünf kurze Beeps mit gleicher Frequenz
- Eintritt in die Optimierungsphase: Ein langer Beep
- Bei einem schweren Fehler: Vier kurze Beeps mit höherer Frequenz

⇒ Nach dem Aktivieren einer Aufgabe im Web-Interface dauert es maximal 20 Sekunden, bis das Modell gestartet wird und es ertönt ein langer Beep gefolgt von fünf kurzen Beeps, da direkt mit einer Lernphase begonnen wird. Sollte nur der Lange Beep zu hören sein, hat bei der Initialisierung des Modells oder bei der Verbindung zum Java-Server etwas nicht funktioniert. Die Erfassungsaufgabe sollte dann abgeschaltet und nach etwa einer Minute neu gestartet werden.

Ertönt der Code für einen größeren Fehler, wird das Modell und MOCHA zurückgesetzt, womit wieder mit einer Lernphase begonnen wird.

Sollte das Modell durch einen Fehler ganz beendet werden, aktiviert der Java-Server seinen Fallback-Modus, in dem einfach in jeder Periode jeder virtuelle Sensor angefragt wird. In diesem Fall sollte ebenfalls alles neu gestartet werden.

Diese Probleme könnten bei fehlerhaften Einstellungen der Erfassungsaufgaben auftreten oder zum Beispiel, wenn man nach dem Start des Modells vergisst, Smartphones in das Testgebiet zu legen, ihre Services zu starten oder sie am Server anzumelden.

A.2.4. Reset

Falls es nötig ist, den Zustand, der bei einem Reset wiederhergestellt wird, in irgendeiner Form abzuändern, ist dies in der Datei
`administrator/components/com_pss/sql/reset.mysql.utf8.sql`
des PHP-Servers möglich. Darin können die MySQL-Queries, die bei einem Reset ausgeführt werden, beliebig abgeändert werden.

Am einfachsten ist das, indem man den Zustand, den man erhalten will, per Web-Interface herstellt und danach einen MySQL-Dump aus den Tabellen mit Präfix '#__pss_' erstellt (dabei muss das jeweilige Tabellenpräfix von Joomla! durch '#__' ersetzt werden).

Dieser Dump kann dann statt den 'INSERT INTO'-Queries in die Reset-Datei eingefügt werden. Dort sollten nur die 'TRUNCATE'-Queries und die 'UPDATE'-Query bestehen bleiben. In der UPDATE-Query können schließlich noch die globalen Einstellungen des Servers, die bei einem Reset gesetzt werden sollen, abgeändert werden.

Literaturverzeichnis

[AMT12] P. Alt, S. Maass, M. Tilk. Project-INF: Development of a Real-World Public Sensing Testbed Based on Android, 2012. (Zitiert auf den Seiten 18 und 25)

[BWD11] P. Baier, H. Weinschrott, F. Dürr. Effiziente automatisierte Erstellung von Straßenkarten. In *7.GI/ITG KuVS-Fachgespräch. Ortsbezogene Anwendungen und Dienste.*, S. 85–92. Logos Verlag Berlin GmbH, Berlin, 2011. (Zitiert auf Seite 15)

[BWDR11] P. Baier, H. Weinschrott, F. Dürr, K. Rothermel. MapCorrect: Automatic Correction and Validation of Road Maps Using Public Sensing. In *36th Annual IEEE Conference on Local Computer Networks (LCN 2011)*, S. 1–8. IEEE Computer Society, Bonn, Germany, 2011. (Zitiert auf Seite 15)

[CEL+06] A. T. Campbell, S. B. Eisenman, N. D. Lane, E. Miluzzo, R. A. Peterson. People-centric urban sensing. In *Proceedings of the 2nd annual international workshop on Wireless internet*, WICON '06. ACM, New York, NY, USA, 2006. (Zitiert auf Seite 15)

[CEL+08] A. Campbell, S. Eisenman, N. Lane, E. Miluzzo, R. Peterson, H. Lu, X. Zheng, M. Musolesi, K. Fodor, G.-S. Ahn. The Rise of People-Centric Sensing. *Internet Computing, IEEE*, 12(4):12 –21, 2008. (Zitiert auf den Seiten 13 und 14)

[CHK08] D. Cuff, M. Hansen, J. Kang. Urban sensing: out of the woods. *Commun. ACM*, 51(3):24–33, 2008. (Zitiert auf Seite 13)

[CIL06] K. C. Cheung, S. S. Intille, K. Larson. An Inexpensive Bluetooth-Based Indoor Positioning Hack, 2006. (Zitiert auf Seite 22)

[CKK+08] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, N. Triandopoulos. Anonysense: privacy-aware people-centric sensing. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, S. 211–224. ACM, New York, NY, USA, 2008. (Zitiert auf Seite 14)

[CMT+08] S. Consolvo, D. W. McDonald, T. Toscos, M. Y. Chen, J. Froehlich, B. Harrison, P. Klasnja, A. LaMarca, L. LeGrand, R. Libby, I. Smith, J. A. Landay. Activity sensing in the wild: a field trial of ubifit garden. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, S. 1797–1806. ACM, New York, NY, USA, 2008. (Zitiert auf Seite 14)

[DGM+04] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, W. Hong. Model-driven data acquisition in sensor networks. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, S. 588–599. VLDB Endowment, 2004. (Zitiert auf Seite 55)

[EML+07] S. B. Eisenman, E. Miluzzo, N. D. Lane, R. A. Peterson, G.-S. Ahn, A. T. Campbell. The BikeNet mobile sensing system for cyclist experience mapping. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, S. 87–101. ACM, New York, NY, USA, 2007. (Zitiert auf Seite 11)

[FKZL03] S. Feldmann, K. Kyamakya, A. Zapater, Z. Lue. An Indoor Bluetooth-Based Positioning System: Concept, Implementation and Experimental Evaluation. In *International Conference on Wireless Networks*, S. 109–113. 2003. (Zitiert auf Seite 22)

[GKS05] C. Guestrin, A. Krause, A. P. Singh. Near-optimal sensor placements in Gaussian processes. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, S. 265–272. ACM, New York, NY, USA, 2005. (Zitiert auf den Seiten 16 und 17)

[HW08] M. Haklay, P. Weber. OpenStreetMap: User-Generated Street Maps. *Pervasive Computing, IEEE*, 7(4):12 –18, 2008. (Zitiert auf Seite 15)

[KBP+08] E. Kanjo, S. Benford, M. Paxton, A. Chamberlain, D. S. Fraser, D. Woodgate, D. Crellin, A. Woolard. MobGeoSen: facilitating personal geosensor data collection and visualization using mobile phones. *Personal Ubiquitous Comput.*, 12(8):599–607, 2008. (Zitiert auf Seite 15)

[KBRL09] E. Kanjo, J. Bacon, D. Roberts, P. Landshoff. MobSens: Making Smart Phones Smarter. *Pervasive Computing, IEEE*, 8(4):50 –57, 2009. (Zitiert auf Seite 15)

[Kra08] A. Krause. *Optimizing Sensing: Theory and Applications*. Dissertation, Carnegie Mellon University, 2008. (Zitiert auf Seite 17)

[LBD+05] B. Liu, P. Brass, O. Dousse, P. Nain, D. Towsley. Mobility improves coverage of sensor networks. In *Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, MobiHoc '05, S. 300–308. ACM, New York, NY, USA, 2005. (Zitiert auf Seite 13)

[LEM+08] N. D. Lane, S. B. Eisenman, M. Musolesi, E. Miluzzo, A. T. Campbell. Urban sensing systems: opportunistic or participatory? In *Proceedings of the 9th workshop on Mobile computing systems and applications*, HotMobile '08, S. 11–16. ACM, New York, NY, USA, 2008. (Zitiert auf Seite 14)

[LLEC10] H. Lu, N. D. Lane, S. B. Eisenman, A. T. Campbell. Bubble-sensing: Binding sensing tasks to the physical world. *Pervasive and Mobile Computing*, 6(1):58 – 71, 2010. (Zitiert auf Seite 15)

[LML+10] N. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, A. Campbell. A survey of mobile phone sensing. *Communications Magazine, IEEE*, 48(9):140 –150, 2010. (Zitiert auf Seite 13)

[MLEC07] E. Miluzzo, N. D. Lane, S. B. Eisenman, A. T. Campbell. CenceMe: injecting sensing presence into social networking applications. In *Proceedings of the 2nd European conference on Smart sensing and context*, EuroSSC'07, S. 1–28. Springer-Verlag, Berlin, Heidelberg, 2007. (Zitiert auf Seite 15)

[MSN+09] N. Maisonneuve, M. Stevens, M. E. Niessen, P. Hanappe, L. Steels. Citizen noise pollution monitoring. In *Proceedings of the 10th Annual International Conference on Digital Government Research: Social Networks: Making Connections between Citizens, Data and Government*, dg.o '09, S. 96–103. Digital Government Society of North America, 2009. (Zitiert auf Seite 15)

[PDR11] D. Philipp, F. Dürr, K. Rothermel. A Sensor Network Abstraction for Flexible Public Sensing Systems. In *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, S. 460 –469. 2011. (Zitiert auf den Seiten 14 und 25)

[PSDR12] D. Philipp, J. Stachowiak, F. Dürr, K. Rothermel. Towards Optimized Public Sensing Systems using Data-driven Models. Technischer Bericht, Institut für Parallele und Verteilte Systeme der Universität Stuttgart, 2012. (Zitiert auf den Seiten 9, 16, 17, 27, 29, 46, 55, 56 und 57)

[STY05] A. Schwaighofer, V. Tresp, K. Yu. Learning Gaussian Process Kernels via Hierarchical Bayes. In L. K. Saul, Y. Weiss, L. Bottou, Herausgeber, *Advances in Neural Information Processing Systems 17*, S. 1209–1216. MIT Press, Cambridge, MA, 2005. (Zitiert auf Seite 16)

[You] W. Young. MySQL connector for C++. MySQL++ is a C++ wrapper for MySQL's C API: http://tangentsoft.net/mysql++/. (Zitiert auf Seite 46)

Alle URLs wurden zuletzt am 12.09.2012 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit
selbstständig verfasst und nur die angegebenen
Quellen benutzt zu haben.

(Patrick Alt)