Software Construction

RWTH AACHEN UNIVERSITY

Fakultät für Mathematik,
Informatik und
Naturwissenschaften

Lehr- und Forschungsgebiet
Informatik 3
Softwarekonstruktion

Bachelorarbeit

Entwicklung eines BeOut-Algorithmus im CiBo-Projekt

Development of a BeOut-Algorithm in the CiBo-Project

vorgelegt von

Richard Alex Sabau

aus Kronstadt (Brasov), Rumänien

am 8. April 2015

Gutachter

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Betreuer

Dipl.-Inform. Andreas Steffens

Hiermit erkläre ich, Richard Alex Sabau, an Eides statt, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keinem anderem Prüfungsamt vorgelegt und auch nicht veröffentlicht.

Aachen, 8. April 2015 (Richard Alex Sabau)

Danksagung

Zunächst möchte ich mich bei Herrn Prof. Dr. rer. nat. Horst Lichter für die Möglichkeit, diese Bachelorarbeit an seinem Lehrstuhl, dem Lehr- & Forschungsgebiet Informatik 3 Sotwarekonstruktion, verfassen zu können, und das Erstgutachten dieser Arbeit bedanken. Auch möchte ich bei Herrn Prof. Dr. rer. nat. Bernhard Rumpe für das Zweitgutachten bedanken.
Ein besonderer Dank geht an meinen Betreuer, Dipl.-Informatiker Andreas Steffens. Er stand mir nicht nur stets mit bestem Rat und bester Tat zur Seite und bat mir die Möglichkeit, ihn jederzeit um Hilfe und Ratschläge zu bitten, sondern hat mich auch in allen richtigen Momenten zu mehr Arbeit gefordert. Ohne seine herzliche und freundliche Art und seine umfassende und tatkräftige Betreuung wäre diese Arbeit nicht zu der geworden, die sie heute ist.
Daneben möchte ich mich bei meiner Familie bedanken, die mich immer moralisch und finanziell unterstützt hat. Besonders möchte ich auch meiner Freundin danken, für ihre Geduld und ihr Verständnis, in den letzten Wochen und Monaten keinen guten Freund gehabt zu haben. Vielen Dank meinem Bruder für die „Korrekturlesesessions“ und auch Jens Böttcher, der mir bei der Einarbeitung in dieses Projekt geholfen hat.

Richard Alex Sabau

Kurzdarstellung

Deutsch

Im Rahmen der Entwicklung eines mobilen Ticketing-Systems, das nach dem Check-In Be-Out-Prinzip den automatischen Ticketkauf von mobilen Fahrtentickets im öffentlichen Personennahverkehr ermöglichen soll, konzipiert und implementiert diese Arbeit einen Algorithmus, der mittels der GPS- und mobilen Internettechnologie den Ausstieg eines Fahrgastes aus einem Fahrzeug in Echtzeit erkennt. Der zum Zeitpunkt dieser Arbeit bestehende Prototyp ermöglichte die Erkennung des Ausstiegs, konnte sie jedoch nicht schnell und robust genug bestimmen. Ein neues Datenmodell und eine umfassende Konzeption führen in dieser Arbeit zu einem schnelleren und robusteren Algorithmus im Vergleich zum bestehenden Prototyp.

English

In the course of the development of a mobile ticketing-system which operates as a Check-In Be-Out-system to enable the automatic dispatch of mobile tickets in the short-distance public transport, this thesis deals with the conception and implementation of an algorithm, which detects the exit of an user out of a vehicle in realtime via GPS- and mobile internet-technology. The existing prototype was able to detect the exit of an user but was not able to do this fast and robust enough for using the mobile ticketing-system productively. A new datamodel and a comprehensive conception lead to a faster and more robust algorithm in comparison to the old prototype.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Liste der Quelltexte

1 Einleitung

Contents

Zu Zeiten des Smartphonebooms stellt die Möglichkeit, den Fahrgästen des öffentlichen Personennahverkehrs ein mobiles und vollautomatisches Ticket-System zur Nutzung der Transportmöglichkeiten anzubieten, nicht nur einen zukunftsorientierten, sondern auch einen zeitgemäßen Ansatz dar, um die Mobilität des öffentlichen Personentransportes mit der Mobilität seiner Fahrgäste zu verschmelzen und ihn somit attraktiver für die Kunden zu gestalten. Viele wissenschaftliche Studien haben bereits bestätigt, dass Handys bzw. Smartphones eine wichtige Rolle in unserem Arbeits- und Lebensumfeld eingenommen haben [LMET09]. Integrierte Lösungen im öffentlichen Personentransport können somit die Mobilität und den Komfort der Nutzer des öffentlichen Transportwesens steigern.

Im Jahr 2007 beschreiben Axel Zietz und Thomas Petersen in ihrem Artikel den noch fehlenden Durchbruch eines solchen mobilen Ticket-Systems in Deutschland, obwohl schon damals im Ausland ausgereifte und anspruchsvolle mobile Ticket-Systeme etabliert worden sind. [ZP08] So konnten sich bis dato die verschiedenen Verkehrsverbände und Aufgabenträger nicht für eine umfassende und flächendeckende Lösung eines mobilen Ticket-Systems einigen. Einen weiteren Grund nennen die Autoren in der Unklarheit, wer die Kosten für die Entwicklung, Implementierung und den laufenden Betrieb der Systeme tragen soll.

Seitdem wurde in Deutschland zwar ein mobiles Ticket-System etabliert, doch existiert bis heute kein mobiles Ticket-System, welches den Ticketkauf für den Transport im öffentlichen Personennahverkehr komplett automatisiert zur Verfügung stellt. Und auch bis heute steht der Kostenaspekt solcher Systeme im Konflikt mit dem Interesse, ein solches System zu entwickeln und zur Verfügung zu stellen.

Mit dieser Vision begann die Firma IVU Traffic Technologies AG in Zusammenarbeit mit dem Lehr- und Forschungsgebiet Informatik 3 der RWTH Aachen im Jahr 2014 an der Entwicklung eines mobilen Ticketing-Systems in Aachen. Diese Arbeit beschäftigt sich mit einer der Komponenten des Projektes - dem Entwurf und der Implementierung eines Webservices, der die Middleware-Komponente des Projektes bilden soll und einen Algorithmus' verwendet, der den Ausstieg des Nutzers aus einem Transportmittel über die GPS-Position des Nutzers und des Transportmittels erkennt.

1.1 Aufbau der Arbeit

Die Arbeit teilt sich in drei logische Abschnitte auf.

Zunächst führen Kapitel *Hintergrund* (2) und *Motivation* (3) in die Hintergründe der Arbeit ein. Sie sollen ein grundsätzliches Wissen zu mobilen Ticket-Systemen und dem Projekt der IVU schaffen. Kapitel 3 stellt neben der Motivation in die Arbeit auch die Anforderungen an den in dieser Arbeit entwickelten Algorithmus vor. Das Kapitel *Related Work* (4) stellt dann ähnliche Projekte und Forschungen vor, um einen Vergleich zu erfolgreichen oder auch nie produktiv eingesetzten mobilen Ticket-Systeme ziehen zu können.

Den zweiten Abschnitt bilden die Konzeption des Algorithmus und der Entwurf der Architektur des Webservices. Dazu schafft Kapitel *Grundlagen der Konzeption* (5) ein theoretisches Fundament zum Verständnis der Problematiken der Ausstiegserkennung und der Funktionalität des entwickelten Algorithmus. Es stellt auch eine grafische Visualisierung vor, die im weiteren Verlauf für die Konzeption des Algorithmus' verwendet wird. Das Kapitel *Modellierung des Konzepts* (6) entwickelt dann gemäß eines iterativen und inkrementellen Vorgehensmodells das Konzept für den Algorithmus. Darufhin folgt in Kapitel *Entwurf* (7) der Entwurf der Softwarearchitektur für den Webservice.

Den dritten Abschnitt der Arbeit bildet die *Umsetzung* in Kapitel (8) und die Evaluation des konzipierten Algorithmus bezüglich der gesetzten Anforderungen in Abschnitt *Evaluation* (9). Das Kapitel *Zusammenfassung* (10) fasst im Anschluss die Arbeit noch einmal zusammen und stellt Möglichkeiten zur Erweiterung und Verbesserungen des in dieser Arbeit entwickelten Algorithmus' vor.

2 Hintergrund

> ‚I want to wake up in a city that never sleeps': Noch Frank Sinatras New-York-Hymne an die Stadt, die niemals schläft, setzte für Handel und Wandel zweierlei voraus: unendliche Kommunikation und grenzenlose Osmose der Stadtbewohner. Also geistige und räumliche Mobilität.
>
> Gerhard Matzig, Journalist und Redakteur der „Süddeutschen Zeitung“

Contents

In diesem Kapitel wird in die Hintergründe dieser Arbeit eingeführt. In den Abschnitten 2.1 und 2.2 wird dazu in den Bereich des Mobile Ticketings eingeführt. Daraufhin folgt eine Beschreibung der Projekt-Vision der IVU AG und die Vorstellung des Prototyps, der im Sommer 2014 entwickelt worden ist.

2.1 E-Ticketing

Dominik Haneberg beschreibt in seinem Artikel das E-Ticketing als eine Form des elektronischen Kommerzes, welches eine neue Form des Kaufs und der Austeilung verschiedenster Arten von Tickets darstellt [Han08]. Die neue Form wird durch den elektronischen Verkauf und die Speicherung eines Tickets in einem elektronischen Speichergerät wie einem Handy charakterisiert. Um welche Art Tickets es sich dabei handelt, spielt für die Bedeutung eines E-Tickets keine Rolle. Es kann sich um Tickets in Form von Fahrtenscheinen für den Personennah- und Fernverkehr oder um Tickets in öffentlichen oder kulturellen Institutionen handeln.

Für die Speicherung von Tickets kommen laut Dominik Haneberg verschiedene elektronische Endgeräte in Frage. So können die Tickets z.B. in Smartcards, in

Smartphones oder auch PDAs gespeichert werden [Han08]. Eine Smartcard ist eine mit einem Mikrocomputer ausgestattete Plastikkarte, die durch den Mikrocomputer dazu in der Lage ist, Daten zu speichern und mit Lesegeräten Daten auszutauschen [HNS+02]. Typische Beispiele für Smartcards sind z.B. Bankkarten.

In den letzten Jahren fand ein sehr starker Anstieg des Smartphonemarktes statt [Gar12]. Darum spielen Smartphones als elektronische Endgeräte eine besonders interessante Rolle im Bereich des E-Ticketings. Durch diese spaltet sich im Bereich des Personentransportes auch eine speziellere Untergruppe des E-Ticketings ab: das „Mobile Ticketing“. In dieser Form des E-Ticketings verwendet der Nutzer sein mobiles Endgerät, z.B. sein Smartphone, um Tickets im Personentransport zu kaufen [BMSW05]. Um diese Form des E-Ticketings geht es auch in dieser Arbeit.

2.2 Mobile Ticketing

Im Rahmen des Mobile Ticketings entstanden in den letzten Jahren verschiedene Ansätze, um den Ticketkauf von Fahrgästen im ÖPNV zu vereinfachen oder auch komplett zu automatisieren. Diese unterscheiden sich grundsätzlich darin, ob der Ticketkauf des mobilen Tickets für ein Transportmittel durch eine aktive Handlung des Fahrgastes eingeleitet bzw. abgeschlossen wird, z.B. indem er mit seiner Smartcard mit einem Lesegerät interagieren muss, oder die Erstellung und/oder der Abschluss des mobilen Tickets durch den Ein- bzw. Ausstieg des Fahrgasts in bzw. aus dem Transportfahrzeug durchgeführt wird. Im Folgenden werden die verschiedenen Mobile Ticketing Ansätze auf Basis der Dissertationsarbeit von Dr. Christian Ordon kompakt vorgestellt [Ord07]:

- Check-In und Check-Out (CICO): Der Fahrgast checkt sich mit seinem elektronischen Mobilgerät manuell vor einer Fahrt oder beim Einstieg in das Transportmittel ein. Der Check-In erstellt das mobile Ticket, durch welches dem Fahrgast der Fahrtantritt mit dem Transportmittel erlaubt wird. Er kann z.B. über einen Chipkartenleser bei den Ein- und Ausstiegstüren des Transportfahrzeugs realisiert werden, mit denen der Fahrgast mit seinem elektronischen Mobilgerät interagieren kann. Beim Ausstieg muss sich der Fahrgast erneut manuell aus dem Transportmittel auschecken. Dies entspricht also zwei aktiven Handlungen des Nutzers. Nach dem Check-Out berechnet ein Hintergrundsystem des CICO-System den Ticketpreis und schließt den Ticketkauf ab.

- Be-In und Be-Out (BIBO): Das mobile Ticket des Fahrgastes wird durch das Betreten eines Empfangsbereiches des Transportmittels erstellt. Dies können z.B. WLAN-Antennen sein, die im Transportmittel angebracht sind. Während der Fahrt befindet sich der Fahrgast innerhalb des Empfangsbereichs von den im BIBO-System verwendeten Antennen. Durch anschließendes Verlassen dieses Empfangsbereichs wird der Endstopp der Fahrt erkannt und der Ticketpreis berechnet. Dies entspricht keiner aktiven Handlung des Fahrgastes zum Kauf eines

Tickets und wird im Report der GWT zu BIBO-Systemen deshalb auch als „hands free“ bezeichnet [Lor09].

- Walk-In und Walk-Out (WIWO): Wie auch in einem BIBO-System wird hier das mobile Ticket durch den Ein- und Ausstieg des Fahrgastes in ein Transportmittel erstellt und der Preis berechnet. Der Unterschied zu einem BIBO-System liegt in der Art der Erkennung des Ein- und Ausstiegs. In BIBO-Systemen findet die Überprüfung der Anwesenheit des Fahrgastes während der Fahrt in bestimmten Zeitabständen statt. Sobald die Anwesenheit des Fahrgastes nicht mehr empfangen werden kann, gilt er als ausgestiegen. In WIWO-Systemen findet nur beim Ein- und Ausstieg eine Datenübertragung statt. Auch wenn sowohl BIBO- als auch WIWO-Systeme eigene Vor- und Nachteile haben, haben sich bislang eher die BIBO-Systeme durchgesetzt, wie in Kapitel 4 erläutert wird. Für genauere Differenzierungen sei hier auf die Ausarbeitung von Thomas Gyger und Olivier Desjeux verwiesen [GD01].

- Check-In und Be-Out (CIBO): Ein CIBO-System entspricht einer Mischung eines CICO- und eines BIBO-Systems. In einem CIBO-System wird das mobile Ticket durch eine aktive Handlung des Fahrgastes erstellt. Wie auch in einem CICO-System checkt sich der Fahrgast manuell vor einer Fahrt oder beim Einstieg in das Transportmittel mit seinem elektronischen Mobilgerät ein. Wie in einem BIBO-System befindet sich der Fahrgast während der Fahrt innerhalb des Empfangsbereichs von im CIBO-System verwendeten Antennen. Sobald der Fahrgast den Empfangsbereich verlässt, berechnet das CIBO-System den Ticketpreis und schließt den Ticketkauf ab.

2.3 Das CIBO-Projekt der IVU

Die IVU AG[1] entwickelt ein CIBO-Projekt im öffentlichen Personennahverkehr des kommunalen Busbetreibers ASEAG[2] im Verkehrsbereich AVV. Die Entwicklung des Projektes findet in Zusammenarbeit mit dem Lehr- und Forschungsgebiet Informatik 3 der RWTH Aachen statt. So wurde im Sommersemester 2014 im Rahmen eines Softwareprojektpraktikums ein erster Prototyp des CIBO-Projektes entwickelt.

Das CIBO-Projekt der IVU zeichnet sich dadurch aus, dass es nur wenig zusätzliche Anforderungen an neue Hardware aufwirft. Für die Nutzung des mobilen Ticketing-Systems benötigt der Fahrgast lediglich ein Smartphone. Daneben müssen die Busse mit GPS-Empfängern ausgestattet werden. Wie aus Abschnitt 2.2 hervorgeht, benötigt die Entwicklung eines mobilen Ticketing-Systems oft bestimmte Hardwareanforderungen bzw. -austattungen der Transportmittel oder auch der Fahrgäste. Die Transportmittel müssten z.B. mit Laserschranken im Falle von WIWO- oder mit WLAN-Antennen im Falle von BIBO/CIBO-Systemen ausgestattet werden.

[1] Homepage der IVU Traffic Technologies AG: http://www.ivu.de/
[2] Homepage zum Busunternehmen ASEAG: http://www.aseag.de/

Werden Smartcards verwendet, benötigen die Fahrgäste zunächst eine solche Smartcard, um das Mobile Ticketing-System verwenden zu können.

Das CIBO-Projekt der IVU nutzt dagegen die Tatsache, dass durch den Smartphoneboom der größte Teil der Fahrgäste über ein Internet- und GPS-fähiges Smartphone verfügt. Die Vision ist, das mobile Ticketing durch mobiles Internet und einem GPS-Datenabgleich zwischen dem Fahrgast und dem Bus zu entwickeln. Dazu hat die IVU die Busse der ASEAG mit GPS-Empfängern ausgestatt. So wurde im Rahmen des Softwareprojektpraktikums ein erster Prototyp entwickelt, der den Ausstieg eines Fahrgastes aus einem Bus über die GPS-Daten des Fahrgastes erkennt.

2.4 Der erste Prototyp

Der erste Prototyp wurde in Form eines Location-Based Service entwickelt. Unter einem Location-Based Service versteht man einen Service, welcher die Informationen der Position eines Handys mit anderen Informationen kombiniert, um dem Handynutzer einen bestimmten Nutzen anzubieten [SV04]. Der im Prototyp entwickelte Location-Based Service verwendet dazu die GPS-Daten des Smartphones des Users und die Einfahrt der Busse in eine Haltestelle. Die Haltestellen sind dafür mit GPS-Empfängern ausgestattet. Sobald ein Bus in einen bestimmten Radius einer Haltestelle einfährt, sendet er diese Information an den „URA-Server“, der in Abschnitt 2.4.4 erläutert wird.

Für den Location-Based-Service wurden im ersten Prototyp des CIBO-Projektes drei Komponenten gemäß einer Client/Server-Architektur entwickelt[Som07]. Die Komponenten kommunizieren jeweils über REST[3]-Schnittstellen miteinander und werden im Folgenden vorgestellt.

2.4.1 Android-App

Die Android-App bildet das Frontend des CIBO-Prototyps. Hier soll sich der Nutzer mit seinen Nutzerdaten anmelden und nach Busverbindungen an einer Haltestelle suchen können. Anschließend soll er sich über eine Nutzereingabe in einen Bus einchecken können. Während der Fahrt soll er die Fahrt mittels der App durch ein mobiles Ticket bei einem Fahrkartencontrolleur verifizieren können. Die Nutzereingaben werden über eine REST-Schnittstelle an den im Prototyp entwickelten Webservice gesendet.

2.4.2 Middleware

Die Middleware des ersten Prototyps wird durch einen Webservice, dem sogenannten Checkout-Server gebildet. Der in Node.js entwickelte Webservice implementiert eine REST-Schnittstelle, über die die App mit dem Webservice kommunizieren kann. Der

[3]Representational State Transfer, oder kurz REST, ist ein architektonisches Paradigma, nach dem Web Services implementiert werden können. Eine nähere Beschreibung findet sich in Kapitel 8.1.2

Checkout-Server, implementiert die Logik des mobile Ticketings. Dabei kommuniziert er über eine REST-Schnittstelle mit dem sogenannten URA-Server[4]. Der URA-Server enthält die Bus- und Haltestellendaten der ASEAG.

Der Checkout-Server ist für die Erstellung einer Nutzerfahrt sowie ihrer Beendigung zuständig. Daher implementiert diese Komponente die Erkennung des Ausstiegs eines Fahrgastes nach dem Be-Out-Prinzip aus 2.2. Nach der Beendigung einer Fahrt erzeugt der Checkout-Server Daten der durchgeführten Fahrt, die er über eine REST-Schnittstelle an die dritte Komponente, die Statistik- und Businesskomponente übergibt.

Die Erkennung des Ausstiegs des Fahrgastes findet über den sogenannten „BeOut-Algorithmus“ statt, welcher den Kern dieser Arbeit bildet.

2.4.3 Business-Backend

Die Statistik- und Businesskomponente bildet das Backend des CIBO-Prototyps und wurde in JavaEE und AngularJS entwickelt und fungiert als ein klassisches Informationssystem zum Persistieren von Fahrtinformationen und Nutzerinformationen. Hier werden gefahrene Nutzerfahrten abgespeichert und statistisch ausgewertet. Die Auswertungen können über eine REST-Schnittstelle durch Webapplikationen eingesehen werden.

2.4.4 Der URA-Server

Der URA-Server bildet eine REST-Schnittstelle zu Echtzeit-Datenbeständen der ASEAG. Das sind u.a. Informationen zu Bushaltestellen, also z.B. ihre Geo-Positionen, IDs und ihr Name, und Informationen von Bussen, wie ihre ID und die von ihnen befahrene Strecke. Fährt z.B. ein Bus in eine Haltestelle ein, sendet der Bus die entsprechende Information an den URA-Server. Diese Informationen können andere Webservices über eine REST-Schnittstelle vom URA-Server anfragen und können aus der Antwort Informationen herausfiltern wie: Der Bus mit ID x ist zum Zeitpunkt t bei der Haltestelle y eingefahren ist. Der Erhalt dieser Echtzeitdaten ermöglicht die Implementierung eines mobilen Ticketing-Systems nach dem **??**-Prinzip.

Eine Übersicht der Grobarchitektur des ersten Prototyps ist in Abbildung 2.1 zu sehen.

2.5 Zusammenfassung

In diesem Kapitel wurde in den Bereich des E-Ticketings und des mobilen Ticketings eingeführt. Man unterscheidet zwischen vier Ansätzen von mobilen Ticketing-Systemen: Das CICO-Prinzip erfordert einen manuellen Ein- und Ausstieg eines Nutzers über sein mobiles Speichergerät, z.B. über Lesegeräte an den Ein- und Ausstiegstüren des Transportmittels. Die BIBO- und WIWO-Prinzipien erfordern keine aktive Handlung

[4]URA steht für Unified Realtime API

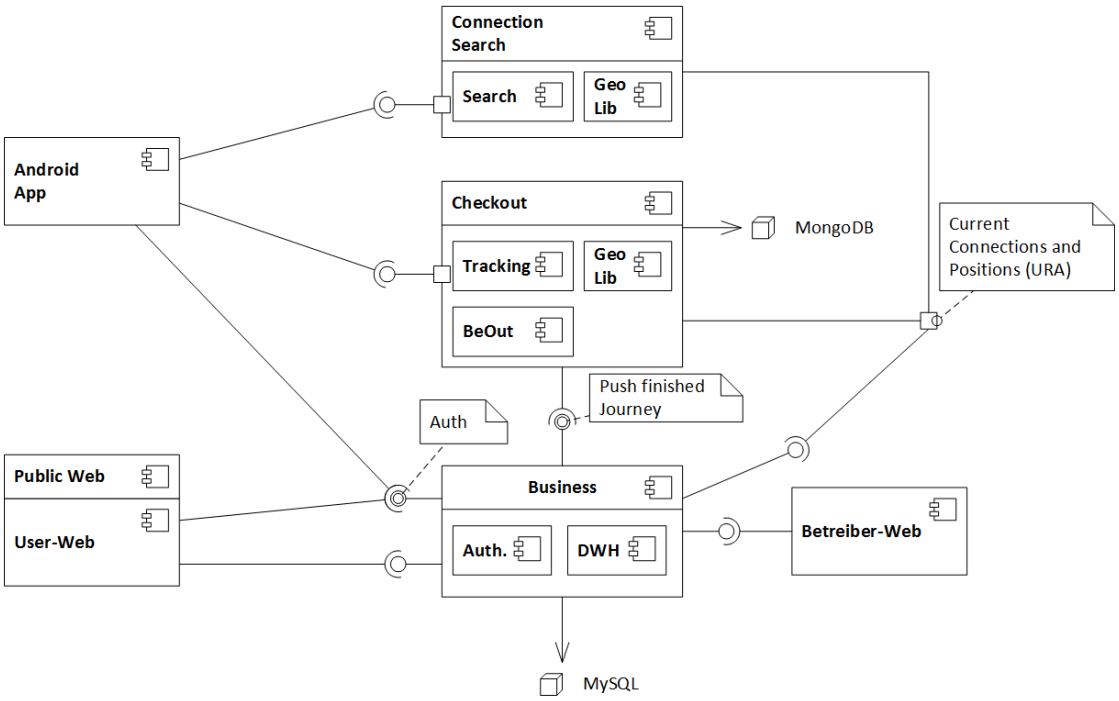

Abbildung 2.1: Grobarchitektur des ersten Prototyps

des Nutzers. Dieser kann mit seinem elektronischen Speichergerät einfach in ein Fahrzeug ein- und aussteigen, welche das BIBO- oder WIWO-Prinzip unterstützt. Das CIBO-Prinzip erfordert eine manuelle Handlung des Nutzers in Form eines Check-Ins. Zum Abfertigen des mobilen Tickets muss der Nutzer nur aus dem Transportmittel aussteigen.

In Abschnitt 2.3 und 2.4 wurde in die Projekt-Vision der IVU AG und den bestehenden Prototyp eingeführt. Der bestehende Prototyp besteht aus drei Komponenten, App, Middleware und Business-Backend, und implementiert das CIBO-Prinzip durch die Einfahrten der Busse in die Haltestellen. Die benötigten Informationen erhält der Checkout-Server, der die Middleware-Komponente bildet, vom URA-Server.

Nachdem das Hintergrundwissen über das Projekt geschaffen worden, auf dessen Basis diese Arbeit stattfindet, folgt im folgenden Kapitel eine Motivation in die Arbeit.

3 Problemstellung

„Geforderte Randbedingungen“ - das ist der Ausdruck, den ich viel lieber anstelle von „nicht-funktionalen Anforderungen“ verwende. Und meine Definition dafür ist einfach: alles, was dem Designer Freiheitsgrade entzieht beim Erfüllen der funktionalen Anforderungen.

Dr. Peter Hruschka

Contents

Dieses Kapitel führt in die Ziele und die Gründe dieser Arbeit ein. Dazu folgt in Abschnitt 3.1 eine Motivation für das Projekt. Im darauffolgenden Abschnitt 3.2 werden die Anforderungen beschrieben, die der neue BeOut-Algorithmus erfüllen soll.

3.1 Motivation

In Kapitel 2 wurde der erste Prototyp im CIBO-Projekt der IVU beschrieben. Dieser entstand auf Basis der ersten Version des URA-Servers. In dieser ersten Version besaßen lediglich die Haltestellen einen GPS-Empfänger, nicht aber die Busse. Sobald die Busse in eine Haltestelle einfuhren, sendeten sie diese Information an den URA-Server, der die Information abspeicherte und damit anderen Nutzern und Webservices den Zugriff auf die Echtzeitdaten ermöglichte.

Seitdem hat die IVU auch die Busse der ASEAG mit GPS-Empfänger ausgestattet und die Funktionalität des URA-Server erweitert. Somit senden die Busse nun in bestimmten Zeitabständen ihre aktuelle Geo-Position an den URA-Server. Dieser speichert die Informationen wie gehabt ab und ermöglicht damit das Tracken der Busse in Echtzeit.

Die Erweiterung des URA-Server von Version 1 auf Version 2 schafft neue Möglichkeiten, die BeOut-Erkennung durchzuführen. Während bislang die Erkennungszeit zwischen einem Ausstieg eines Fahrgastes und der Erkennung des Ausstiegs durch den

BeOut-Algorithmus durch die Wegzeit des Busses zu seiner nächsten Haltestelle nach unten beschränkt war, kann nun ein BeOut-Algorithmus konzipiert werden, der das Echtzeit-Tracking der Busse durch den URA-Server für die Erkennung des BeOut-Status verwendet.

Die Entwicklung des ersten BeOut-Algorithmus' auf Basis der ersten Version des URA-Servers hat also gezeigt, dass die automatische Erkennung des Ausstiegs eines Fahrgastes aus einem Transportmittel über die GPS- und mobile Internettechnologie möglich ist. Die Entwicklung dieses Algorithmus' zeigte aber auch, dass die bloße Verwendung von GPS-Daten der Bushaltestellen nicht ausreicht, um bestimmte Qualitätsanforderungen an den BeOut-Algorithmus zu erfüllen. In dieser Arbeit wird erforscht, ob die Erweiterung des URA-Servers dies ermöglicht. Dafür werden im folgenden Abschnitt die Qualitätsanforderungen an den neuen BeOut-Algorithmus definiert.

3.2 Anforderungen

„Der Nutzen, den uns eine Software bringt, liegt in ihrer Funktion, genauer in der Funktion des Programms“ [LL10]. Gemäß dieses Zitates von Ludewig und Lichter beginnt dieser Abschnitt mit der Formulierung der funktionalen Anforderungen an den BeOut-Algorithmus bzw. den Checkout-Server.

3.2.1 Funktionale Anforderungen

In Kapitel 2.4 wurden zwei Stakeholder des Checkout-Servers beschrieben: Die App und das Business-Backend. Neben diesen sind auch die Nutzer der App und die Betreiber, also die ASEAG und IVU, Stakeholder des Prototyps.

Eine funktionale Anforderung an den BeOut-Algorithmus wurde bereits mehrfach genannt: Die automatische Erkennung des Ausstiegs eines Nutzers. Dabei muss aber auch der Check-In der App durch den Checkout-Server verarbeitet werden. Nach der Erkennung sollte der Nutzer der App auf seinem Handy nachverfolgen können, dass sein Ausstieg erkannt worden ist. Dazu benötigt die App eine Schnittstelle zum Checkout-Server. Gleichermaßen benötigt das Business-Backend statistische Daten einer Fahrt, um Nutzern und Betreiber statistische Auswertungen liefern zu können.

Damit ergeben sich die folgenden funktionalen Anforderungen:

F1 Der Checkout-Server muss den Ausstieg des Fahrgastes im Sinne des Be-Out-Systems (vgl. Kapitel 2.2) erkennen. Dies soll voll automatisch, d.h. ohne eine zusätzliche Handlung des Nutzers geschehen.

F2 Der Checkout-Server muss der App eine Kommunikationsschnittstelle zur Verfügung stellen. Über die Kommunikationsschnittstelle soll die App in der Lage sein, beim Checkout-Server einen Check-In anzumelden. Nach automatischer Erkennung des Ausstiegs soll der Checkout-Server der App über diese Schnittstelle

über die Erkennung des Ausstiegs informieren. So lange der Nutzer nicht ausgestiegen ist, soll der Checkout-Server die App über die Schnittstelle darüber informieren können, dass der Nutzer sich noch im Bus befindet.

F3 Hat der Checkout-Server den Ausstieg des Fahrgastes erkannt, muss er Daten an das Business-Backend über eine Schnittstelle senden, die das Business-Backend dem Checkout-Server zur Verfügung stellt.

Aus einer Ist-Analyse wird ersichtlich, dass die funktionalen Anforderungen an den Checkout-Server im ersten Prototyp erfüllt worden sind:

zu F1 Der Checkout-Server erkennt den Ausstieg des Fahrgastes ohne einer zusätzlichen Aktion des Fahrgastes bzw. der App. Dazu wurden verschiedene Konzepte entworfen, um den Ausstieg zu erkennen, die allesamt die GPS-Positionsdaten des Handys des Fahrgastes für die Ausstiegserkennung verwenden.

zu F2 Die App kann über eine REST-Schnittstelle mit dem Checkout-Server kommunizieren. Dabei sendet die App die Daten an verschiedene URIs, abhängig davon, ob sie einen Check-In oder ein Update der Appdaten durchführt.

zu F3 Nach Beendigung einer Fahrt, sendet der Checkout-Server über eine REST-Schnittstelle des Business-Backends die erforderlichen Daten an das Business-Backend.

Der erste Prototyp hat also die funktionalen Anforderungen des Projektes erfüllt und somit gezeigt, dass eine vollautomatische Ausstiegserkennung im öffentlichen Personennahverkehr anhand eines Abgleichs von GPS-Daten möglich ist. Um Aussagen über die Qualität des Prototyps machen zu können, müssen nun nicht-funktionale Anforderungen an den Checkout-Server bestimmt werden.

3.2.2 Nicht-funktionale Anforderungen

Die Qualitätsanforderungen an den BeOut-Algorithmus sind sehr fachlich und lassen sich damit in den ISO/IEC 25000-Standard einordnen, auch wenn sie sich gemäß des Qualitätenbaums von Ludewig und Lichter prinzipiell in den Bereich der Zuverlässigkeit und Nützlichkeit einordnen lassen [LL10]. Darum wird, bevor diese Qualitätsanforderungen herausgearbeitet werden, eine nicht-funktionale Anforderung an den Checkout-Server formuliert, der in den Bereich der Wiederverwendung einer Komponente fällt.

Wie in Kapitel 2.4 beschrieben wurde, verwendet die App im ersten Prototyp eine REST-Schnittstelle zur Kommunikation mit dem URA-Server. Während F1 fordert, dass der Checkout-Server eine generelle Kommunikationsschnittstelle zur Verfügung stellt, formulierte folgende nicht-funktionale Anforderung die Erhaltung der Schnittstelle zwischen den beiden Komponenten:

NF1 Um die Middleware-Komponente adaptierbar zu halten, soll die Art der Kommunikation zwischen der Middleware und der App über eine REST-Schnittstelle beibehalten werden.

Um nun die Qualitätsanforderungen an den Algorithmus zu formulieren, müssen ihre genauen Bedeutungen festgehalten werden.

Die Qualität des Algorithmus variiert mit der Effizienz der BeOut-Erkennung. Auch wenn „Effizienz“ ein guter Ausdruck zur Beschreibung der fachlichen Qualität der BeOut-Erkennung ist, bietet sich der Begriff als Qualitätsanforderung nicht an, da die Bedeutung der Effizienz im softwaretechnischen Sinne zu schwammig in Bezug auf die gemeinte Effizienz des Alogrithmus' ist. Die Effizienz gilt nämlich im softwaretechnischen Sinne als hoch, „wenn die Software kaum mehr Rechenzeit benötigt, als minimal erforderlich wäre“ [LL10]. Um sich von diesem Effizienz-Begriff abzugrenzen, wird die Qualitätsanforderung der fachlichen Effizienz des Algorithmus von jetzt an beschrieben als die Erkennungsgeschwindigkeit des BeOut-Algorithmus.

Erkennungsgeschwindigkeit

Die *Erkennungsgeschwindigkeit des BeOut-Algorithmus* entspricht der zeitlichen Differenz zwischen dem Ausstieg eines Fahrgastes und der Erkennung des Ausstiegs durch den BeOut-Algorithmus.

Damit lässt sich die erste Qualitätsanforderung an den BeOut-Algorithmus definieren:

NF2 Der BeOut-Algorithmus soll den BeOut-Status schnellstmöglich erkennen. Das bedeutet, dass die zeitliche Differenz zwischen dem Ausstieg des Fahrgastes und der Erkennung des BeOut-Status minimal gehalten werden soll.

Der Nutzer, als ein Stakeholder des Prototyps, soll nicht zu lange auf die BeOut-Erkennung warten müssen, da sonst die Attraktivität und damit die Verwendung des mobilen Ticketing-Systems leidet. Neben der Erkennungsgeschwindigkeit spielt auch eine zweite Qualitätsanforderung eine wichtige Rolle, die zur Attraktivität des Ticketing-Systems beim Nutzer beiträgt; die Robustheit der Erkennung des BeOut-Algorithmus.

Robustheit

Eine hohe Robustheit des BeOut-Algorithmus steht dafür, dass das Verhältnis zwischen korrekt erkannten Ausstiegen und fehlerhaft erkannten Ausstiegen sehr gering ist. Im Idealfall soll der BeOut-Algorithmus also niemals einen Ausstieg zu früh oder zu spät erkennen. Dies ist nicht nur für den Nutzer, sondern auch für den Betreiber des mobilen Ticketing-Systems eine äußerst wichtige Qualitätsanforderung. Die Attraktivität des Ticketing-Systems beim Nutzer nimmt bei einer zu geringen Robustheit ab. Doch auch für den Betreiber bedeutet jedes fehlerhaft abgerechnete Ticket einen deutlich erhöhten Arbeitsaufwand durch Stornierungen und Korrekturen. Ganz abgesehen davon

müsste der Fahrkartenkontrolleur einen Schwarzfahrer von einem Fahrgast mit zu früh abgeschlossenem Ticket unterscheiden können.

Somit ergibt sich für die zweite Qualitätsanforderung an den BeOut-Algorithmus:

NF3 Die Robustheit des BeOut-Algorithmus soll maximal gehalten werden.

Um die zwei Qualitätsanforderungen effektiv testen und evaluieren zu können, folgt eine 4. nicht-funktionale Anforderung an den Algorithmus.

Testbarkeit

Im Sinne der Entwicklung des BeOut-Algorithmus ist es vorteilhaft, die Berechnung des Algorithmus stets automatisch und unabhängig der anderen Komponenten, also isoliert testen zu können, um gezielt die Effizienzgeschwindigkeit und Robustheit des BeOut-Algorithmus testen zu können. Mit der Forderung wird die 4. nicht-funktionale Anforderung an den BeOut-Algoritmus gesetzt.

NF4 Die Checkout-Komponente soll derart entworfen werden, dass der BeOut-Algorithmus isoliert und automatisch getestet werden kann. Dabei sollen die Qualitätsanforderungen gemessen werden können.

Zusammenfassung

Zu Zeiten des Smartphonebooms stellt ein attraktives Mobile Ticketing System einen sehr interessanten Bereich im E-Ticketing und dem öffentlichen Personennahverkehr dar. Der erste Prototyp des CIBO-Projektes der IVU AG hat gezeigt, dass ein BeOut-Konzept mit GPS-Abgleich möglich ist, doch fehlte die Möglichkeit, den Algorithmus schnell und robust zu gestalten. Darum wird ein neuer BeOut-Algorithmus auf Basis eines neuen URA-Servers entwickelt

Die Anforderungen an den neuen BeOut-Algorithmus sind:

Funktionale Anforderungen:

F1 Der Checkout-Server muss der App eine Kommunikationsschnittstelle zur Verfügung stellen.

F2 Der Checkout-Server muss den Ausstieg des Fahrgastes im Sinne des Be-Out-Systems erkennen. Dabei soll kein Zutun der App nötig sein.

F3 Hat der Checkout-Server den Ausstieg des Fahrgastes erkannt, muss er Daten an das Business-Backend über eine Schnittstelle senden, die das Business-Backend dem Checkout-Server zur Verfügung stellt.

Nicht-funktionale Anforderungen:

NF1 Die Art der Kommunikation zwischen dem Checkout-Server und der App soll beibehalten werden.

NF2 Der BeOut-Algorithmus soll den BeOut-Status schnellstmöglich erkennen. Das bedeutet, dass die zeitliche Differenz zwischen dem Ausstieg des Fahrgastes und der Erkennung des BeOut-Status minimal gehalten werden soll.

NF3 Die Robustheit des BeOut-Algorithmus' soll maximal gehalten werden.

NF4 Die Checkout-Komponente soll derart entworfen werden, dass der BeOut-Algorithmus isoliert und automatisch getestet werden kann. Dabei sollen die Qualitätsanforderungen gemessen werden können.

Im folgenden Abschnitt werden verschiedene Projekte aus dem Bereich Mobile Ticketing vorgestellt, um zu analysieren, ob ähnliche Konzepte bereits existieren und verwendet werden können.

4 Related Work

Contents

Im Inland wie im Ausland sind bereits Mobile Ticketing Systeme in Form von CICO-, WIWO- und BIBO-Systemen (vgl. 2.1) entwickelt worden. Auch eine Form eines CIBO-Systems existiert in London in Form des Oyster Card-Systems [ZP08]. Diese Systeme werden in den folgenden Abschnitten vorgestellt.

4.1 Oyster Card in London

Die Oyster Card ist eine kontaktlose Smart Card, die seit 2002 in London für den öffentlichen Personentransport verwendet wird und eine Kombination aus CICO und CIBO darstellt. Wird die Oyster Card bei Benutzung von Trams und U-Bahnen verwendet, muss sich der Fahrgast gemäß des CICO-Prinzips vor und nach der Fahrt über ein CICO-Terminal ein- bzw. auschecken. Bei Verwendung in Bussen ist ein CIBO-Prinzip implementiert. Der Fahrgast muss sich nur beim Einstieg in den Bus einchecken. Der Preis der Fahrt wird durch eine Buspauschale von der Oyster Card abgebucht [ZP08].

Die Oyster Card funktioniert über einen sogenannten Radio-Frequency-Identification[1]-, oder kurz, RFID-Chip [ZP08]. Ein RFID-Chip ist ähnlich einer Chipkarte und besteht aus einem auf der Karte angebrachten Datenträger. So können verschiedene Informationen, wie eine Identifikation des Karteninhabers auf der Karte abgespeichert werden und Daten des Datenträgers durch spezielle Lesegeräte ausgetauscht werden. Allerdings findet die Energieversorgung und der Datenaustausch eines RFID-Chips nicht wie bei herkömmlichen Chipkarten durch einen galvanischen Kontakt mit den Kontakfedern des Lesegerätes statt. Stattdessen verwendet ein RFID-Chip magnetische oder elektromagnetische Felder für die Energieversorgung und

[1] Die technischen Verfahren hinter Radio-Frequency-Identification wurden aus der Funk- und Radartechnik übernommen und bedeutet so viel wie „Identifikation durch Radiowellen“ [Fin12]

den Datenaustausch mit einem Lesegerät. Dadurch kann der Datenaustausch kontaktlos erfolgen [Fin12].

Somit muss die Oyster Card vor und nach jeder Fahrt lediglich vor ein dafür entwickeltes Lesegerät gehalten werden. Um den Preis der Fahrten auch beim Wechsel von Transportmitteln korrekt zu berechnen, existieren je nach Haltestelle gelbe und pinke Lesegeräte. Die gelben Lesegeräte stellen die Check-In- und Check-Out-Terminals dar. Die pinken Lesegeräte werden verwendet, wenn der Fahrgast das Transportmittel wechselt, z.B. wenn er die Durchfahrt von Zone 1, der teuersten öffentlichen Transportmittelzone in London, umgehen muss, denn Zone 1 darf nicht mit jeder Oyster Card befahren werden. Die Bezahlung der Fahrten erfolgt durch Aufladung eines Prepaid-Guthabens der Oyster Card. Ist das Guthaben der Karte aufgebraucht, signalisiert das Lesegerät durch ein rotes Licht, dass das Guthaben aufgebraucht ist. Ab dann kann der Fahrgast noch eine Fahrt mit der Oyster Card durchführen, ehe er die Karte wieder aufladen muss [Oys].

Das Mobile Ticketing über die Oyster Card hat sich über viele Jahre hinweg in London bewährt. Auch kann die positive Resonanz der Fahrgäste anhand des gesteigerten Gebrauchs der Oyster Card erkannt werden. Wurden im Jahr der Einführung der Oyster Card 2002/2003 nicht mal 1% der Fahrten mit öffentlichen Transportmitteln mit der Oyster Card getätigt, so waren es im Jahr 2006/2007 bereits 73% aller Fahrten [ZP08].

Auch wenn die Form des CIBO-Systems gemäß der Definition von Be-Out in Kapitel 2.2 nicht als ein wirkliches Be-Out bezeichnet werden kann (der Ausstieg des Fahrgastes wird nie tatsächlich von einem System erkannt) und somit nicht mit dem Be-Out-Prinzip dieser Arbeit verglichen werden kann, so sticht der Erfolg der Oyster Card positiv für die weitere Forschung am Mobile Ticketing heraus.

Neben der Oyster Card gibt es auch in Deutschland ein ähnliches CICO-Konzept, das sogar deutschlandweit verwendet wird: Das Touch&Travel-System der Deutschen Bahn.

4.2 Touch&Travel der Deutschen Bahn

Touch&Travel verwendet eine ähnliche Technik wie das Oyster Projekt. Allerdings verwendet Touch&Travel keine RFID-Chips sondern Near Field Communication, oder kurz NFC [DGG07].

Die Datenübertragung bei der Near Field Communication findet im hochfrequenten Bereich (13,56 MHz) statt und ermöglicht zwei NFC-fähigen Geräten den drahtlosen Datenaustausch auf einer Entfernung von bis zu 20cm. Bei dieser Technik handelt es sich eher um eine drahtlose Datenschnittstelle wie Bluetooth. Durch die Nähe der RFID-Technik ist NFC allerdings auch für RFID-Systeme interessant. So können nicht nur zwei NFC-Interfaces, wie z.B. zwei Handys, via NFC miteinander kommunizieren. Auch RFID-Lesegeräte können mit einem NFC-Interface Daten austauschen. Klaus Finkenzeller fasst die NFC-Interfaces wie folgt zusammen: „Ein NFC-Interface vereint [...] die Funktion eines Datentransceivers, eines RFID-Lesegerätes und eines RFID-Transponders in einer Funktionseinheit [...]“ [Fin12].

Das Touch&Travel-Projekt der Deutschen Bahn funktioniert nach dem CICO-Prinzip. Das Handy des Fahrgastes nimmt dabei die Funktion des elektronischen Speichermediums in Form eines NFC-Interfaces ein. Via NFC-Datenaustausch mit einem sogenannten Touchpoint der Deutschen Bahn, checkt sich der Fahrgast ein. Dabei sendet der Touchpoint Informationen der Haltestelle und die Uhrzeit des Check-Ins an das Hintergrundsystem weiter. Dazu hält der Fahrgast sein Handy nach Starten der Touch&Travel-App in die Nähe eines an den Haltestellen angebrachten Touchpoints. Nach manueller Bestätigung des Check-Ins, gilt der Fahrgast als fahrtberechtigt bzw. „eingecheckt“. Bei der Ankunft an seinem Ausstiegspunkt muss der Fahrgast noch einmal per Tastendruck die NFC-Kommunikation mit einem Touchpoint einleiten, um das Ticket abzuschließen. Dies entspricht dem Check-Out. Die Informationen, mit welchem Zug der Fahrgast gefahren ist, z.B. mit dem ICE oder dem Regionalverkehr, werden bei der Ticketkontrolle des Zugbegleiters an das Hintergrundsystem gesendet [DGG07].

Neben Touch&Travel existieren in Deutschland weitere Forschungen und Projekte im Bereich des Mobile Ticketings. Das Projekt Ring&Ride wurde bis 2008 erforscht und spielt eine besonders interessante Rolle, da es, wie auch die Vision der IVU AG, Fahrten mittels Handyortung ermöglicht.

4.3 Ring&Ride

Ring&Ride war ein vom Bundesministerium für Wirtschaft und Energie unterstütztes Forschungsprojekt im Bereich des Mobile Ticketings und wurde von 2005 bis 2008 durchgeführt [ELR09]. Da in Ring&Ride positionsabhängige Funktionen des Handys verwendet werden, gilt auch dieses Projekt als ein Location-Based Mobile Ticketing Service, vgl. Kapitel 2.4 [LMET09].

Ein wichtiger Bestandteil des Ring&Ride-Projektes ist die Verschmelzung von Nah- und Fernverkehr mittels Mobile Ticketings. Der Check-In wird durch einen Anruf des Fahrgastes an eine kostenlose Telefonnummer erzeugt. Für den Check-Out muss der Fahrgast die Nummer erneut anrufen. Beim Check-In sendet das Hintergrundsystem eine Ticketnummer per SMS an das Handy des Fahrgastes. Nach dem Check-Out sendet das Mobile Ticketing System (diese Komponente geht aus den Quellen nicht deutlich hervor) die Positionsdaten an ein Hintergrundsystem, genannt „route tracing“, welches Zugriff auf die geokodierten Positionsdaten der Bus- und Bahnstationen und die Zeitpläne der Busse und Bahnen hat. Um die gefahrene Strecke zu bestimmen, wird die Position des Fahrgastes zum Zeitpunkt der beiden Anrufe für den Check-In und den Check-Out über GSM/UMTS, GPS und WLAN bestimmt. Neben den beiden Positionsbestimmungen wird die Position des Fahrgastes zusätzlich in bestimmten Zeitintervallen während der Fahrt über die genannten Technologien bestimmt. Zusätzlich kann die Position auch via NFC bestimmt werden. Da dies aber eine aktive Handlung des Fahrgastes impliziert, baut das Projekt eher auf den anderen Technologien auf, allerdings befürworten die Autoren die Verwendung von NFC für die Zukunft [LMET09].

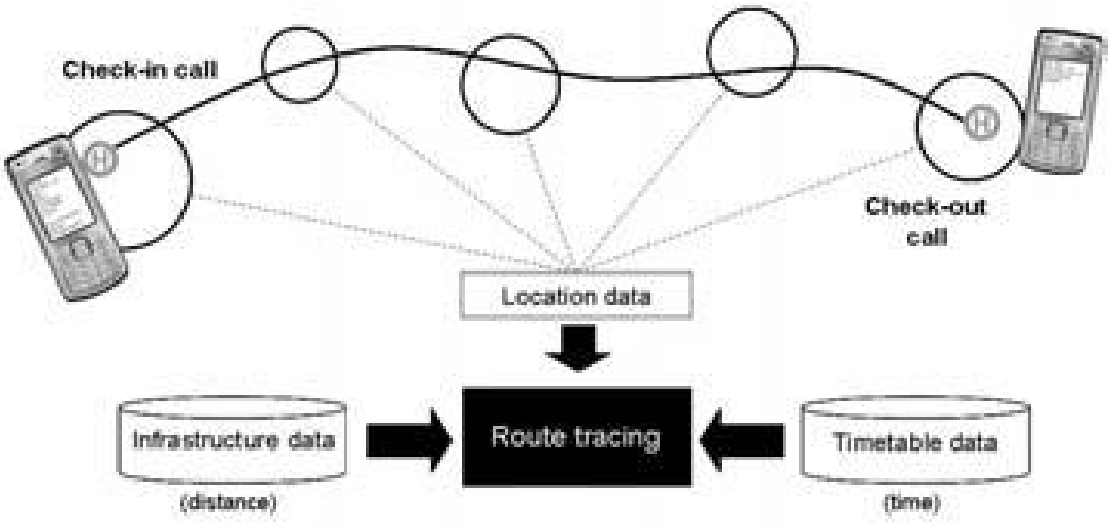

Abbildung 4.1: Das Konzept des Ring&Ride-Projekts (Quelle: [LMET09])

Abbildung 4.1 zeigt das Prinzip von Ring&Ride.

Bereits im Zuge des Ring&Ride-Projekts wurde erforscht, dass es bei Verwendung der genannten Technologien zum Rekonstruieren der gefahrenen Strecke zu Ungenauigkeiten und Fehlern bei der Bestimmung der gefahrenen Strecke kommen kann, wie in Tabelle 4.1 zu sehen ist. Daher verwendet das „Location Subsystem“ von Ring&Ride abhängig von der Verfügbarkeit der Daten verschiedene Technologien, genannt „hybrid positioning strategies“. Dabei hat GPS bei der Positionsbestimmung die höchste Priorität. Wenn der GPS-Empfang beim Update nicht gegeben ist, wechselt das Location Subsystem auf die Verwendung von WLAN. Kann auch WLAN nicht verwendet werden, nutzt das Subsystem GSM um die Position des Fahrgastes über das Mobilfunknetzt grob zu orten. [LMET09].

Technologisch ähnelt das Ring&Ride-Projekt stark mit der Projekt-Vision der IVU AG. Allerdings wird in Kapitel 6 festgestellt, dass es konzeptuell einen großen Unterschied macht, den Be-Out anhand der Positionsdaten in Echtzeit zu erkennen anstelle einen Check-Out zu implementieren. Der Hauptgrund liegt im zeitlichen Bezug der Daten. Durch den im Ring&Ride-Projekt implementierten Check-Out, muss die gefahrene Route erst im Nachhinein rekonstruiert werden. Der Einstiegs- und Ausstiegspunkt sind aber durch die manuelle Aktion des Fahrgastes eindeutig (auch wenn GPS-Fehler bei der Lokalisierung der Position auftreten) und damit eine harte Anforderung an den Service [LL10]. In der Projektvision der IVU AG muss der BeOut-Algorithmus hingegen Echtzeitdaten während der Fahrt auswerten und anhand derer erst den Ausstieg erkennen. Dies wird in Kapitel **??** näher beschrieben. Offensichtlich ist dies ein schwierigeres Problem als ein definites Signal des Handys des Fahrgastes zu erhalten und im Nachhinein die Positionsdaten auszuwerten und damit die Route zu rekonstruieren.

Criteria	GSM based localization		GPS based localization	
	Long distance travel	***Local public transport***	***Long distance travel***	***Local public transport***
Route accuracy	89%	67%	-	84%
Price correctness	91%	99%	-	100%

Tabelle 4.1: Korrektheit der Weg- und Preisbestimmung mit GSM und GPS im Ring&Ride-Projekt (Quelle: [ELR09])

Um den Blick vom Check-Out- hin zum Be-Out-System zu richten, werden noch zwei weitere Mobile Ticketing-System vorgestellt, die jeweils nach dem BIBO-Prinzip funktionieren. Eine Pionierarbeit leistete dabei das Projekt EasyRide aus der Schweiz.

4.4 EasyRide in der Schweiz

EasyRide war ein in der Schweiz entwickeltes Forschungsprojekt, das sowohl in Form eines WIWO- als auch in der Form eines BIBO-Projektes entwickelt und erforscht worden ist und gilt als erstes BIBO-Projekt, mit dem Feldtests stattgefunden haben [Lor09]. Allerdings verwenden beide Projekte die selbe Technologie und unterscheiden sich nur in ihrer Anwendung.

Im EasyRide-Projekt trägt der Fahrgast einen aktiven, mit einem RFID-Chip und Batterien ausgestatteten Transponder[2] in Form einer Karte mit sich (vgl. 4.1). In den Fahrzeugen sind verschiedene Antennen installiert, die auf unterschiedlichen Frequenzen Daten senden und empfangen. An allen Ein- und Ausgängen sind Sender installiert, die jeweils zwei „Low-Frequency“-Felder (LF) im Bereich von 125 KHz schaffen. Daneben existieren auch eine oder mehrere „Ultra-High-Frequency“-Transceivers (UHF), die ein UHF-Feld im Bereich von 434 MHz erzeugen. Die LF-Felder sind so an den Ein-/Ausgängen angeordnet, dass durch das Betreten der Felder die Bewegungsrichtung des Fahrgastes erkannt werden kann. Bezeichne Feld A das LF-Feld, das am nächsten an der Tür liegt und Feld B das LF-Feld, dass tiefer im Fahrzeug liegt. Dann kann der Einstieg des Fahrgastes dadurch erkannt werden, dass er zuerst durch Fald A und dann durch Feld B tritt. Beim Ausstieg gilt die umgekehrte Reihenfolge [GD01].

Nachdem die Karte des Fahrgastes die LF-Felder betreten hat, beginnt sie, Daten mit dem UHF-Transceiver auszutauschen. Der Transceiver kommuniziert während der Fahrt mit allen EasyRide-Karten und dem Bordcomputer des Fahrzeugs. Dem Bordcomputer

[2]Der Vorteil zu passiven Transpondern liegt darin, dass aktive Transponder auf einer höheren Reichweite mit hoher Datenrate Daten senden und empfangen können. Dafür ist der Stromverbrauch auch größer als bei passiven Transpondern (vgl. [GD01]).

sendet er die gesammelten Informationen aller EasyRide-Karten der Fahrgäste an den, welcher die Daten an ein Hintergrund-System weitersendet. Sobald der Ausstieg erkannt worden ist, rekonstruiert das Hintergrundsystem die gefahrene Strecke das Fahrgastes und berechnet den Ticketpreis [GD01].

Der Unterschied zwischen dem WIWO- und dem BIBO-System ist, wie auch in Kapitel 2.2 beschrieben, dass stetige Überprüfen der Position des Fahrgastes im BIBO-System. Im WIWO-System tauscht die EasyRide-Karte genau zwei mal Daten mit dem UHF-Transceiver aus, nämlich beim Ein- und beim Ausstieg, wenn er durch die LF-Felder tritt. Im BIBO-System tauschen die beiden Komponente immer wieder Daten miteinander aus. Daraus folgt auch der Vorteil vom WIWO- zum BIBO-System. Die EasyRide-Karten sind im WIWO-System deutlich energieeffizienter. Dafür beschreiben Gyger und Desjeux in ihrem Artikel den Vorteil des BIBO-Systems in der Einfachheit des Algorithmus' sowie dem robusteren Kommunikationsprotokoll [GD01].

Dem Report über CICO-, CIBO und BIBO-Systeme im ÖPNV ist zu entnehmen, dass das EasyRide-Projekt eine Erfassungsqualität von 99,2% im BIBO-System hatte und auf sehr positive Resonanz beim Nutzer gestoßen ist. Allerdings blieb es nur bei diesen Feldtests und wurde bis heute nicht produktiv entwickelt. Einen Grund dafür nennen die Autoren des Reports in den für die damalige Zeit viel zu hohen Kosten, um das EasyRide-System in der Schweiz produktiv zu etablieren [Eil14].

Das EasyRide-Projekt hat eine sehr gute Basis dafür gesetzt, dass Be-Out-Systeme im öffentlichen Personentransport verwendet werden können, auch wenn die Kosten in dem spezifischen Projekt zu hoch waren. Im Rahmen des Leitprojektes „intermobil Region Dresden“ wurde das Prinzip des Be-Out-Erkennung des EasyRide-Projektes erweitert und näher erfoscht, wie folgender Abschnitt zeigt.

4.5 ALLFA-Ticket in Dresden

Das ALLFA-Ticket des Projekts „intermobil Region Dresden“ war ein Pilotprojekt, das vom Fraunhofer Institut für Verkehrs- und Infrastruktursysteme und Siemens VDO u.a. in Zusammenarbeit mit dem Verkehrsverbund Oberelbe und den Dresdner Verkehrsbetrieben von März bis Ende Oktober 2005 durchgeführt worden ist.

Das BIBO-Prinzip wurde im intermobil-Projekt mittels einer vollautomatischen Erfassung des sogenannten ALLFA-Tickets, welches der Fahrgast mit sich führen muss, realisiert. Das ALLFA-Ticket ist dabei eine spezielle elektronische Karte oder auch ein spezielles Mobiltelefon [Fra07]. Wie die Erfassung des ALLFA-Tickets technologisch funktioniert wird aus einer keiner Quelle genau ersichtlich. Das Fraunhofer Institut erklärt, die Erfassung geschehe „mit Hilfe spezieller Antennen auf Basis der »Be-In/Be-Out Technologie«“ [Fra07]. Laut Matthias Wirtz hält es sich beim ALLFA-Ticket um eine Smartcard, welche er im Vorfeld auf Basis der RFID-Technologie einführt. Somit kann davon ausgegangen werden, dass die Erfassung des ALLFA-Tickets wie EasyRide auch auf der RFID-Technologie basiert [Wir14].

Die von Siemens VDO speziell entwickelte BIBO-Technologie verwendet für die Erkennung zwei verschiedene Arten von Antennen, die im Transportmittel

angebracht sind: mehrere sogenannte Low-Frequency-„Weckantennen“, die über jeder Tür angebracht sind und im Bereich von 6,78 MHz geschaltet sind, sowie einer zentralen Ultra-High-Frequency-„Erfassungsantenne“, die im Bereich von 868 MHz geschaltet ist [Fra07], [Lor09]. Hier wird auch gleich die Nähe zum EasyRide-Projekt ersichtlich, welches auch zwei verschiedene Arten von Antennen verwendet (vgl. 4.4).

Befindet sich der Fahrgast nicht in einem mit den Antennen ausgestatteten Fahrzeug, ist das ALLFA-Ticket im „Sleep-Mode“. Beim Betreten eines solchen Fahrzeugs senden die Weckantennen einen Weckruf an das ALLFA-Ticket, sodass dieses aus dem „Sleep-Mode“ erwacht. Sobald das Fahrzeug die Haltestelle verlässt, beginnt die Datenübertragung der „Erfassungsantenne", die sämtliche im Fahrzeug befindlichen ALLFA-Ticket-IDs an das Fahrzeug überträgt und somit den Fahrgast als solchen registriert. Dabei können nur die ALLFA-Tickets mit der Erfassungsantenne kommunizieren, die durch die Weckantennen erfolgreich aus dem Sleep-Mode geweckt worden sind. Solange sich das Fahrzeug bewegt, kommunizieren alle ALLFA-Tickets parallel mit der Erfassungsantenne. Damit kann die gesamte Fahrt aller Fahrgäste, die ein ALLFA-Ticket verwenden, getrackt werden. Sobald der Fahrgast das Fahrzeug verlässt, endet die Kommunikation mit der Erfassungsantenne, die dadurch das Ticket dieser ALLFA-Ticket-ID abschließen kann. An welcher Haltestelle der Fahrgast ein- und ausgestiegen ist, wird durch die GPS-Daten des Busses ermittelt. Der Bus sendet während der Fahrt in zyklischen Abständen alle Daten an das Hintergrundsystem weiter [Lor09].

Das intermobil-Projekt hat auf Basis von EasyRide ein äußerst robustes BIBO-System erschaffen, das nicht nur den automatisch Ein- und Ausstieg des Fahrgastes erkennt, sondern diese auch sehr robust und effizient umsetzt. Durch die Tatsache, dass die Registrierung eine ALLFA-Ticket-ID nicht bereits beim Eintritt in das Fahrzeug stattfindet, erlaubt das den Fahrgästen spontan aus dem Fahrzeug auszusteigen und das Transportmittel zu wechseln. Auch die Problematik, dass man fehlerhafterweise mit einem Fahrzeug fährt, weil man z.B. in der Nähe des Fahrzeugs stand, ist durch die später erfolgende Registrierung reduziert. Die Weckantennen senden auf einer solch niedrigen Frequenz (6.78MHz), dass sie nur die ALLFA-Tickets aufwecken können, die sich in den Eingangs- bzw. Ausgangstüren befinden [Lor09].

Auch wenn das intermobil-Projekt sehr ausgereift war und eine sehr hohe Erfassungsgenauigkeit von 99,5% erreicht hat, ging es, wie auch beim EasyRide-Projekt, nie über die Pilottests hinaus. Als Gründe nennnen die Ersteller des Reports über CICO-, CIBO und BIBO-Systeme im ÖPNV, dass das ALLFA-Ticket noch nicht massentauglich war und, wie beim EasyRide-Projekt, die Kosten der Fahrzeugausrüstung einfach zu hoch waren [Eil14].

Neben den vorgestellten CICO- und BIBO-Projekten hat auch ein Projekt in die Richtung von CIBO erforscht: Das ATRON-Projekt

4.6 CIBO im Projekt der Firma ATRON

Das einzig bekannte CIBO-Projekt ist ein Projekt der schweizer Firma ATRON[3] [Eil14]. In diesem Projekt verwendet der Fahrgast eine kontaktlose Smartcard für den Check-In. Nach dem Check-In findet in bestimmten Zeitabständen ein Datenaustausch zwischen der Smartcard und den WLAN-Antennen statt, indem die Antennen der Smartcard Datenpakete an die Smartcard schickt. So lange die WLAN-Antennen eine Antwort der Smartcard auf die Datenpakete erhalten, gilt der Fahrgast nicht als Be-Out [Lor09].

Dieses Projekt ging allerdings nicht über einen Prototyp für Forschungszwecke und Demonstrationen hinaus und ist in der Literatur auch nicht näher erläutert. Bis dato wurde die Funktionalität des Projektes auch nicht durch Feldtests bewiesen [Lor09].

Zusammenfassung

Projekte wie die Oyster Card oder Touch&Travel bestehen schon seit längerer Zeit und haben sich als konsistent und für den Fahrgast attraktiv bewiesen. Allerdings verwenden beide Projekte keinen Be-Out-Mechanismus (im eigentlichen Sinne des Be-Outs, vgl. Kapitel 2.2).

Neben den sich durchgesetzten CICO-Systemen gibt es bis heute allerdings kein Be-Out-Verfahren, das in der Praxis eingesetzt wird. Projekte wie EasyRide und intermobil haben gezeigt, dass ein Mobile Ticketing System, das die Technik eines Be-Outs implementiert, sehr komplex und schwer handhabbar sein kann. Die Ausstattung der Fahrzeuge mit zusätzlicher Hardware ist in den Lösungen der genannten Projekte unabdingbar, was eine weitere Schwierigkeit im Bezug auf die Finanzierung der Kosten eines solchen Projektes aufwirft.

Auf Basis dieses Wissens wird nun das Konzept und der Entwurf eines BeOut-Algorithmus' im IVU-Projekt vorgestellt, das lediglich das Smartphone des Fahrgastes und die GPS-Antenne des Busses für eine BeOut-Erkennung verwendet.

Dazu dient das folgende Kapitel zum Schaffen der Grundlagen, um die Entwicklung des Konzepts und das Konzept an sich besser verstehen zu können.

[3] Homepage der ATRON systems AG: http://www.atron-systems.ch/de/

5 Grundlagen zur Konzeption

> Im Schönen vereinigt sich, wie im höheren Handeln überhaupt, immer Theoretisches und Praktisches.
>
> Karl W. F. Solger

Contents

In diesem Kapitel wird die Problemstellung des BeOut-Algorithmus abstrahiert und formalisiert. Dies dient dem Verständnis der Schwierigkeiten, die durch die Konzeption des Algorithmus' im anschließenden Kapitel gelöst werden müssen.

Dazu werden in Abschnitt 5.1 die Problemfaktoren erläutert, die die BeOut-Erkennung erschweren. In den darauf folgenden Abschnitten wird ein atomares Modell einer Fahrt erschaffen, und mit Regeln beschrieben, um eine Fahrt bzw. die BeOut-Erkennung auf ein Minimum an Eigenschaften zu reduzieren. Auf Basis dieses Modells wird in Kapitel 6 der BeOut-Algorithmus konzipiert, indem die Regeln iterativ außer Kraft gesetzt werden. Dies erleichtert die Herangehensweise zur Lösung der BeOut-Erkennung trotz der Problemfaktoren. In Abschnitt 5.4 wird die BeOut-Erkennung im atomaren Modell veranschaulicht. Das bildet die Basis der weiteren Konzeption. Zum Schluss in Abschnitt 5.5 folgt die Beschreibung eines zu Visualisierungszwecken geschaffenen Modells, das in der weiteren Konzeption verwendet wird.

5.1 Problemfaktoren

Die Schwierigkeiten, die eine Konzeption für einen BeOut-Algorithmus erschweren, liegen an verschiedenen Problemfaktoren, die sich grob in drei Kategorien einordnen lassen:

1. äußere Faktoren höherer Gewalt, auf die man keinen Einfluss nehmen kann und zu besonderen Randfällen führen können - dazu gehören z.B. Internetverbindungsabbrüche oder Ausfälle der Bordmechanik.

2. äußere Faktoren niedriger Gewalt, auf die der Algorithmus indirekt Einfluss nehmen kann, indem er den Realfall mittels Heuristiken zu rekonstruieren versucht - das sind z.B. (hohe) Latenzen nach dem Senden eines Updates, (hohe) GPS-Ungenauigkeiten oder das Verhalten bzw. die Fortbewegung des Fahrgastes nach seinem Ausstieg.

3. innere Faktoren, auf die der Algorithmus unter gewissen Bedingungen direkten Einfluss nehmen kann - z.B. die Update-Intervalle der App.

Generell beachtet diese Bachelorarbeit die Faktoren der 1. Kategorie nicht weiter, da der Beout-Algorithmus auf diese Faktoren weder Einfluss nehmen kann, noch ohne Weiteres mittels Heuristiken die Fahrt des Users rekonstruieren kann und diese Faktoren somit außerhalb der Zuständigkeit des BeOut-Algorithmus liegen. Selbstverständlich kann ein Timeout oder ähnliches implementiert werden, um mit diesen Faktoren umzugehen, doch hat der Umgang mit diesen Faktoren keinen Einfluss auf die BeOut-Erkennung an sich.

Insgesamt lassen sich in diese drei Kategorien sehr viele Problemfaktoren einordnen, darum werden nun die einzelnen Problemfaktoren definiert, auf die im weiteren Verlauf der Ausarbeitung der Fokus liegt:

1. verschieden hohe Latenzen zwischen Senden und Empfangen von Updates

2. verschieden hohe Fehler in den empfangenen GPS-Koordinaten der App und des Busses

3. unregelmäßige Updatezeitpunkte des Busses

4. lange und schnelle Fahrten des Busses ohne Halt

5. der User bewegt sich nach dem Ausstieg in der Nähe des Busses, z.B. mit einem Fahrrad, weiter fort

Aufgrund dieser unterschiedlichen Problemfaktoren wurde ein iteratives Vorgehen beim Konzipieren der Lösung gewählt. Denn auch wenn einzelne Problemfaktoren leicht erkannt und behandelt werden können, ist die Kombination dieser Problemfaktoren deutlich komplexer zu behandeln. Man stelle sich z.B. eine schnelle Fahrt des Busses vor, in der hohe Latenzen auftreten und der Bus nur jede halbe Minute seine Position aktualisiert. Und in diesem Beispiel ist die Fehlerbehaftung der Positionsdaten noch nicht einmal mit eingeschlossen.

Um mit diesen Problemfaktoren in der Konzeption des BeOut-Algorithmus umgehen zu können, wird nun ein atomares Modell beschrieben, für das ein erstes Konzept vorgestellt wird. Das atomare Modell soll hier keineswegs den Anspruch auf ein minimales Modell setzen. Dennoch wurde der Begriff „atomar“ gewählt, da das Modell das minimale Modell bezüglich der Problemfaktoren darstellt.

5.2 Das atomare Modell

Das atomare Modell setzt alle Problemfaktoren aus Abschnitt 5.1 außer Kraft. Das bedeutet insbesondere, dass in diesem Modell die Zeitpunkte des Sendens und Empfangs eines Signals die selben sind, also keine Latenzen existieren, und Signale fehlerfrei gesendet und empfangen werden.

Im Detail setzt das Modell folgende Restriktionen für die oben genannten Faktoren der 2. und 3. Kategorie:

1. Es existieren keine Latenzen und eine unendlich hohe Bandbreite. Der Zeitpunkt t des Sendens des Updates der App entspricht dem selben Zeitpunkt t des Empfangs dieses Updates beim Server.

2. Die App sendet die Positionsdaten des Users ohne jeglicher Fehlerbehaftung, d.h. die Positionsdaten, die der Checkout-Server empfängt, entsprechen den tatsächlichen Positionsdaten des Users.

3. Nach dem Ausstieg des Users aus dem Bus fährt der Bus sofort weiter. Insbesondere handelt es sich nie um eine Haltestelle, bei der der Bus lange an der Haltestelle stehen bleibt, wie etwa einer Endhaltestelle.

4. Zwischen der Wiederweiterfahrt des Busses und dem nächsten Update seiner Positionsdaten fährt der Bus ungestört in seiner maximal erlaubten Geschwindigkeit. Das heißt, dass er nicht an einer Ampel oder dergleichen halten muss.

5. Der User bewegt sich nach dem Ausstieg bis zur BeOut-Erkennung zu Fuß weiter oder bleibt an der Bushaltestelle stehen.

6. Die App und der Bus updaten ihre Daten immer zeitgleich.

Auch wenn das Modell die Realität stark abstrahiert, bietet es den Vorteil, klar und einfach durch Regeln definiert werden zu können. Die folgenden Regeln und Gesetze formen das Modell und werden im weiteren Verlauf der Bachelorarbeit verwendet, um Unterschiede zwischen den iterativ entschärften Modellen aufzuzeigen.

5.3 Die Restriktionen des atomaren Modells

Zunächst folgen Definitionen von Variablen, die im Laufe der Modellierung verwendet werden.

Variablen, die Ereignisse beschreiben

A : Ereignis, in dem die App ihr Update versendet

B : Ereignis, in dem der Bus sein Update versendet

U : Ereignis, in dem der User den Bus verlässt

Variablen, die Zeitpunkte beschreiben

t : ein Zeitpunkt
Man beachte, dass beim Vergleich von und der Arbeit mit Zeitpunkten, stets auf Sekundenbasis gearbeitet wird, d.h. für einen Zeitpunkt t entspricht $t + 1$ eine Sekunde später.

$t_u(A)$: Zeitpunkt, in dem die App ihr Update versendet

$t_u(B)$: Zeitpunkt, in dem der Bus sein Update versendet

$t_{u+1}(A)$: Zeitpunkt des übernächsten Updates der App, wenn $t_u(A)$ der Zeitpunkt ihres nächsten Updates ist

$t_{u+1}(B)$: Zeitpunkt des übernächsten Updates des Busses, wenn $t_u(B)$ der Zeitpunkt seines nächsten Updates ist

$t_e(A)$: Zeitpunkt, in dem der Checkout-Server das Update der App empfängt

$t_e(B)$: Zeitpunkt, indem der Checkout-Server die Busdaten vom URA-Server empfängt

$t(U)$: Zeitpunkt des Ausstiegs des Users

Man beachte, dass $t_e(B)$ eine Vereinfachung des Realfalles ist, da im Realfall der Checkout-Server für die Busdaten den URA-Server anfragt und dieser mit den Busdaten antwortet und nicht der Bus die Daten an den Chekout-Server sendet, wie es aus der Variablendefinition hervorgeht. Somit gibt es im Realfall eine weitere Kommunikation, in der Latenzen auftreten können. Da es für den Checkout-Server aber uninteressant ist, wie hoch die Latenzen beim Senden der Busdaten zum URA-Server waren, wird hier die komplette Kommunikation zum Empfangen der Busdaten zu einer einzelnen Kommunikation reduziert.

Variablen, die Fehler beschreiben

Δ_{lat} : zufälliger Fehler in der Breitengrad-Koordinate

Δ_{lang} : zufälliger Fehler in der Längengrad-Koordinate

Variablen, die Positionen oder Flächen beschreiben

$p_a = \begin{pmatrix} lat_a \\ lang_a \end{pmatrix}$: gesendete Position der App, wobei lat_a der Breitengrad und $lang_a$ der Längengrad der App sind. Analog gelten die Definitionen für $lat, lang$ in den weiteren Definitionen

$p_b = \begin{pmatrix} lat_b \\ lang_b \end{pmatrix}$: gesendete Position des Busses

$\widetilde{p}_a = \begin{pmatrix} lat_a \cdot \Delta_{lat} \\ lang_a \cdot \Delta_{lang} \end{pmatrix}$: empfangene, fehlerbehaftete Position der App

$\widetilde{p}_b = \begin{pmatrix} lat_b \cdot \Delta_{lat} \\ lang_b \cdot \Delta_{lang} \end{pmatrix}$: empfangene, fehlerbehaftete Position des Busses

$r_b = r(\widetilde{p}_b)$: Der Radius um die empfangene Position des Busses

Variablen, die Abhängigkeiten zwischen Ereignissen und Zeitpunkten beschreiben

(A, t) : Update der App findet zum Zeitpunkt t statt

(B, t) : Update des Busses findet zum Zeitpunkt t statt

(p_a, t) : Position der App zum Zeitpunkt t

(p_b, t) : Position des Busses zum Zeitpunkt t

Diese Variablen definieren im folgenden die Regeln, die im Modell gelten.

Restriktionen der Zeit

(1) $t_u(A) = t_e(A)$

Bedeutung: Der Zeitpunkt des Updates der App ist gleich dem Zeitpunkt des Empfangs dieses Updates beim Checkout-Server

(2) $t_u(B) = t_e(B)$

Bedeutung: Der Zeitpunkt des Updates des Busses ist gleich dem Zeitpunkt des Empfangs dieses Updates beim Checkout-Server.

Die beiden wichtigen zeitlichen Forderungen des Modells werden durch folgende Regeln beschrieben:

(3) $B \leftrightarrow A \wedge (A,t) = (B,t)$

Bedeutung: Bus und App updaten immer zeitgleich.

(4) $t_e(A) = t_u(A) \wedge t_e(B) = t_u(B) \wedge t_e(A) = t_e(B)$

Bedeutung: Der Zeitpunkt des Empfangs beider Updates ist gleich dem Zeitpunkt des Sendens beider Updates.
Aus Regel 3 folgt damit, dass zu einem Zeitpunkt t, mit $t = t_u(A)$ dem Checkout-Server die Daten der App und die Daten des Busses zur Verfügung stehen.

Einfacher formuliert modellieren diese 4 Regeln, dass *im Modell alle Zeitpunkte des Sendens und Empfangs von Updates identisch sind* und *Bus und App immer gleichzeitig updaten.*

Auch auf Basis der Positionsdaten setzt das atomare Modell Restriktionen.

Anforderungen des Raumes

(5) $U \rightarrow \Big(\exists t : \big(t(U) < t \wedge t = t_u(B)\big) \wedge \big((B,t) : p_a \notin r_b\big)\Big)$

Bedeutung: Nach dem Ausstieg des Users muss nach endlicher Zeit ein Update des Busses erfolgen, bei welchem gilt, dass der User zum Zeitpunkt des Updates des Busses nicht mehr im Radius des Busses war.

(6) $U \rightarrow \Big(\exists t' : \big(t' = t_{u+1}(B) \wedge t(U) \ll t' \wedge (p_a, t') \ll (p_b, t')\big)\Big)$

Bedeutung: Nach dem Ausstieg des Users gibt es einen Zeitpunkt, der in der Zukunft liegt und bei dem das nächste Busupdate stattfinden wird und zu dem Zeitpunkt wird die Distanz zwischen App und User sehr groß sein.
Vereinfacht: Nach dem Ausstieg des Busses bewegt sich der User bis zum anschließenden Update nicht schnell in Fahrtrichtung des Busses, z.B. mit einem Auto, weiter.

Eine äußerst wichtige Einschränkung in diesem Modell macht außerdem die folgende Regel:

(7) $\widetilde{p}_a = p_a \wedge \widetilde{p}_b = p_b$

Bedeutung: Die empfangenen Positionsdaten entsprechen den gesendeten Positionsdaten. Alternativ:
$\forall A, B : \Delta_{lat} = 1, \Delta_{lang} = 1$

Diese Regeln beschreiben somit ein atomares Modell, in welchem das Senden und Empfangen von Signalen fehlerfrei und zeitgleich stattfindet und das Verhalten des

Users und des Busses nach dem Ausstieg des Users vorgeschrieben ist.

Wie in diesem Modell die funktionale Anforderung wie in Kapitel 3.2.1 definiert erfüllt werden kann und wie sich die nicht-funktionalen Anforderungen aus Kapitel 3.2.2 in dieser Lösung verhalten, beschreibt der nächste Abschnitt.

5.4 Das Konzept der BeOut-Erkennung am atomaren Modell

Die Entscheidung, ob sich der User zum Zeitpunkt seines Updates im Bus befand, lässt sich in diesem Modell trivial lösen. Sobald der Checkout-Server neue Positionsdaten der App und des Busses empfängt, muss er lediglich überprüfen, ob sich p_A in einem bestimmten Bereich um p_B herum befindet.
Nach dem Modell, ließe sich dies über ein Rechteck realisieren, dessen Länge und Breite denen des Busses entsprächen, da p_B und p_A die exakten Positionsdaten sind.
Im Sinne der weiteren Arbeit wird jedoch ein Kreis gewählt, der um p_B gezogen wird. Dies hat den Grund, dass dieser Abschnitt das Grundkonzept vorstellt, wie es auch in der weiteren Konzeption des Algorithmus verwendet wird, und es zur Zeit der Bearbeitung der Bachelorarbeit noch nicht möglich war, die Himmelsrichtung des Busses zum Zeitpunkt seines Updates vom URA-Server zu erfahren. Zwar ist die Funktionalität in der API des URA-Servers bereits dokumentiert, jedoch noch nicht implementiert, so dass es nicht möglich war, etwas genaueres als einen Radius für die Erkennung zu wählen. Die Fahrtrichtung mittels der letzten beiden empfangenen GPS-Positionen des Busses zu bestimmen wäre zu ungenau und dadurch fehleranfällig gewesen. Dadurch wäre es an dieser Stelle nicht zielführend, etwas anderes als einen Radius für die Konzeption im formalisierten Modell zu verwenden.
Der Radius des Kreises entspricht der Länge des Busses, da die GPS-Antenne der Busse über dem Fahrerhaus angebracht ist und der Radius somit den ganzen Bus abdeckt. Damit existiert zwar ein räumlicher Overhead, wie in Abbildung 5.1 zu sehen ist, im Sinne der Modellierung im formalisierten Modell stört er aber nicht. Allerdings wird er in den späteren Modellen eine wichtige Rolle spielen.

Im formalisierten Modell reichen also ein statisch großer Radius und eine simple Fallunterscheidung aus, um die funktionale Anforderung der automatischen Erkennung des Ausstiegs zu erfüllen:

1. $p_A \in r_B \Rightarrow$ BeOut-Status $= false$
2. $p_A \notin r_B \Rightarrow$ BeOut-Status $= true$

Abbildung 5.2 und 5.3 visualisieren die beiden Fälle. Beide Abbildungen zeigen ein Update zum Zeitpunkt t.

Nun ist zu überprüfen, wie sich bei diesem Konzept im formalisierten Modell die nicht-funktionalen Anforderungen Erkennungsgeschwindigkeit und Robustheit aus 3.2.2 verhalten.
Durch die Forderung des Modells, dass $\widetilde{p}_a = p_a$ und $\widetilde{p}_b = p_b$ gilt, erkennt diese Lösung den Ausstieg mit einer Genauigkeit von 100%. Dies liegt insbesondere auch daran, dass das formalisierte Modell die eigentlich harte Anforderung an den Algorithmus modelliert. Entweder ist der User ausgestiegen, oder er ist es nicht. Sobald die Positionsdaten der App außerhalb des Radius' empfangen werden, „weiß" der Checkout-Server, dass der User sich nicht mehr im Bus befinden kann. Dadurch sind jegliche Fehler ausgeschlossen.

Für die Erkennungsgeschwindigkeit ergibt sich ein ähnliches Bild. Dadurch, dass der Server beide Positionsdaten gleichzeitig und zum Zeitpunkt des Updates erhält, kann er der App sofort zurückmelden, ob der BeOut-Status erkannt worden ist, oder nicht. Insbesondere muss er keine zusätzlichen Überprüfungen durchführen, da die funktionale Anforderung in diesem formalen Modell eine harte Anforderung ist [LL10].
Durch die Wahl eines Radius' anstelle eines Rechtecks büßt das Konzept zwar ein wenig an Erkennungsgeschwindigkeit ein, so könnte der Algorithmus nämlich nicht immer gleich nach dem Ausstieg zurückmelden, dass der User ausgestiegen ist, wie es bei einem Rechteck der Fall wäre. Durch die Forderung des Modells, dass der Bus alsbald nach dem Ausstieg weiterfährt und der User sich nicht schnell in Richtung des Busses fortbewegt, ist die Einbuße der Erkennungsgeschwindigkeit aber relativiert.

In einem formalisierten Modell ist es also sehr leicht möglich, die funktionalen und nicht-funktionalen Anforderungen an den Algorithmus zu erfüllen. Bevor der Blick weiter zum Konzept des Realmodells geworfen wird, wird das Sequenzenmodell vorgestellt, das im Rahmen dieser Arbeit entwickelt worden ist.

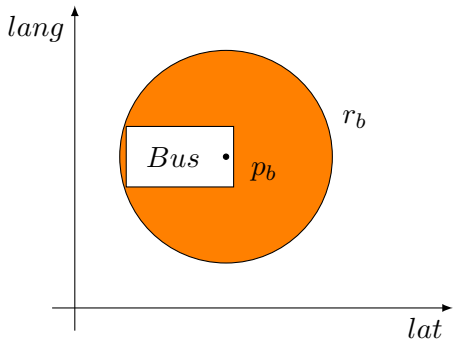

Abbildung 5.1: Overhead durch den Radius r_b

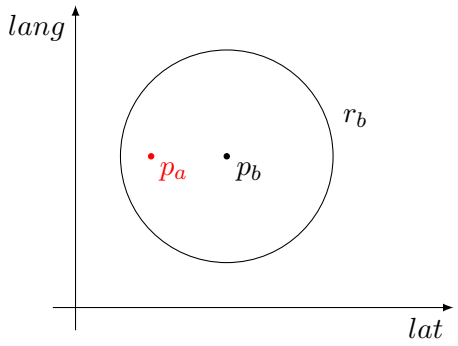

Abbildung 5.2: BeOut-Status = false zum Zeitpunkt $t_U(A)$

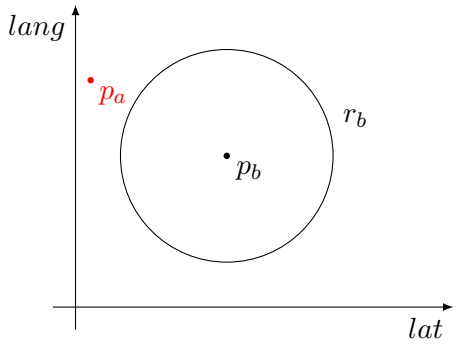

Abbildung 5.3: BeOut-Status = true zum Zeitpunkt $t_U(A)$

5.5 Das Sequenzenmodell

Um die Folgen der Entschärfungen im Folgenden besser verstehen zu können, wird das Sequenzenmodell vorgestellt. Ziel ist es dabei, eine Abstrahierung eines Trips in Form von **Sequenzenreihen** zu schaffen. Die Sequenzenreihen können dann in Form einer graphenähnlichen Visualisierung auf zeitlicher Ebene mit einander in Bezug gesetzt werden.

Das Sequenzenmodell definiert für jedes Update des Busses und der App eine spezifische, eindeutige Kennung, genannt Sequenznummer. Mit dieser Kennung kann nicht nur jedem Update ein eindeutiger Zeitpunkt zugeordnet werden. Hiermit können insbesondere Latenzen und die Folge dieser im Konzept visualisiert werden. Durch die Bildung sogenannter Sequenzenreihen können auch im Falle hoher Latenzen die einzelnen Updates zeitig genau zugeordnet werden. Dies wird für die Konzeption des Algorithmus eine essentielle Rolle spielen.

Definition:

Definiere für einen Trip die **Sequenznummern** S_{a_n} und S_{b_m}, mit $m, n \in \mathbb{N}$ wie folgt: S_{b_m} sei die Sequenznummer des m-ten Busupdates, analog S_{a_n} die Sequenznummer des n-ten Appupdates.

Der **Wert einer Sequenznummer** wird definiert durch:
$S(i), i \in N$ für eine Bussequenznummer, und $S(j), j \in [A_1, B_1, C_1, \ldots, Z_1, A_2, B_2, \ldots]$ für eine Appsequenznummer.

Der **Zeitpunkt des Updates** bzw. der **Zeitpunkt des Empfangs** einer Sequenznummer wird definiert durch $t_u(X)$, bzw. $t_e(X)$. X kann dabei eine Sequenznummer oder der Wert einer Sequenznummer wie oben definiert sein.
Im Folgenden stehen $\boldsymbol{X}$ und $\boldsymbol{Y}$ stets für die Möglichkeit, eine Sequenznummer oder den Wert einer Sequenznummer anzunehmen.

Eine **Sequenzenreihe** S_a für die App, analog S_b für den Bus, ist eine *aufsteigend sortierte Folge von Sequenznummern*, beginnend bei A für S_a und bei 1 für S_b, wobei A und 1 jeweils für das erste Update der App bzw. des Busses während eines Trips stehen.

Ein **passendes Sequenzenpaar** ist ein Paar von zwei Sequenznummern (S_{b_i}, S_{a_j}), $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$, die durch bestimmte Eigenschaften zusammengehören. Analog kann ein Sequenzenpaar auch durch den Wert der Sequenznummern $(S(i), S(j))$ dargestellt werden.

Dann kann ein **Trip** dargestellt werden als **zwei Sequenzenreihen** $S_a = S_{a_1}, \ldots, S_{a_n}$ und $S_b = S_{b_1}, \ldots, S_{b_m}$, wobei jede Sequenznummer aus S_b zu mindestens einem **passenden Sequenzenpaar** gehört. S_{a_n} ist die Sequenznummer desjenigen Appupdates, durch welches der BeOut-Status erkannt wurde.

Ein Trip besteht somit aus zwei Sequenzenreihen, die jeweils die Updatefolge des Busses und der App repräsentieren. Ein passendes Sequenzenpaar stellt die beiden Updates dar, die zur Erkennung des BeOut-Status miteinander vergleichen werden.

Abbildung 5.4 zeigt ein Beispiel, wie ein Trip in Form des Sequenzenmodells dargestellt werden kann. In der Visualisierung gilt $t_u(A) = t_e(A)$ und $t_u(B) = t_e(B)$.

Man beachte, dass die Sequenznummern der Appupdates eigentlich A_1, B_1 usw. benannt sein müssten. Da in allen aufgezeigten Beispielen in dieser Bachelorarbeit allerdings nie so viele Appupdates verwendet werden, dass eine genaue Identifizierung des Updates durch einen Index nötig ist, wird von jetzt an stets davon ausgegangen, dass eine Sequenznummer der App ohne Index gleichbedeutend mit einer Sequenznummer der App mit Index = 1 ist.

Die passenden Sequenzenpaare sind hier: $(1, A), (2, B), (2, C), (3, E), (3, F), (4, H)$ diese sind hier willkürlich ausgewählt und dienen der Veranschaulichung, dass es verschiedene Möglichkeiten gibt, zwei Sequenzennummern zu einem passenden Sequenzenpaar zusammenzufassen. Die Zeit wird in den Modellen stets von links nach

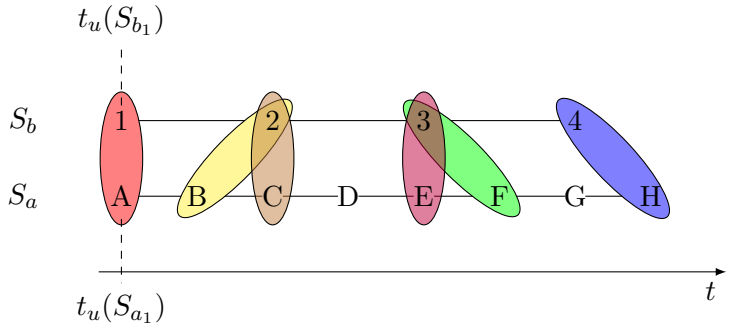

Abbildung 5.4: Visualisierung eines Trips durch das Sequenzenmodell

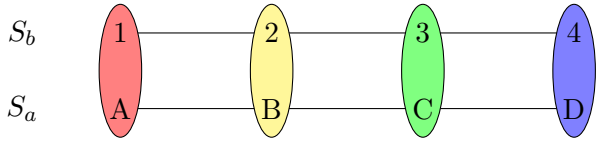

Abbildung 5.5: Visualisierung eines Trips im atomaren Modell durch das Sequenzenmodell

rechts angezeigt. Damit kann jeder Sendung und jedem Empfang einer Sequenznummer ein bestimmter Zeitpunkt während eines Trips zugeordnet werden (vergleiche dazu S_{b_1} und S_{a_1} in der Abbildung).

Inwieweit diese Definitionen bei der Konzeption des Algorithmus weiterhelfen, zeigt Abbildung 5.5, welches einen Trip im atomaren Modell und das Konzept der BeOut-Erkennung an diesem Trip durch die Wahl der passenden Sequenzenpaare darstellt.

Nach den Regeln (3) und (4) in 5.3 aktualisieren Bus und App ihre Daten immer zeitgleich und die Zeitpunkte des Sendens der Updates sind die selben Zeitpunkt wie die des Empfangs der Updates. Damit ergibt sich, dass die Anzahl der Elemente in S_b gleich der Anzahl der Elemente in S_a sein muss (Regel (3)) und sie auf der Zeitlinie immer übereinander liegen (Regel (4)).
Außerdem wurde im Konzept in 5.4 herausgearbeitet, dass bei jedem Update die Positionsdaten dieses Updates des Busses und der App miteinander verglichen werden. Damit ergeben sich auch direkt die passenden Sequenzenpaare, wie in der Abbildung zu sehen ist.

Zusammenfassung

In diesem Abschnitt wurden die Problemfaktoren der BeOut-Erkennung identifiziert. Daraufhin wurde ein atomares Modell eingeführt, in welchem diese Problemfaktoren nicht gelten und für dieses Modell ein Konzept entworfen, wie der BeOut-Algorithmus in diesem Modell den Ausstieg des Users erkennen könnte. Außerdem wurde das Sequenzenmodell vorgestellt, mit dem verschiedene zeitliche Ereignisse und in Relationen stehende Daten visualisiert werden können.

Im folgenden Kapitel werden diese Techniken eingesetzt, um das Konzept des BeOut-Algorithmus zu entwickeln.

6 Modellierung des Konzepts

Contents

In diesem Kapitel wird das Konzept des neuen BeOut-Algorithmus' entwickelt. Dabei wird der Algoritihmus entsprechend eines inkrementellen und iterativen Vorgehensmodell nicht in einem Zug konstruiert, sondern in einer Reihe von aufeinander aufbauenden, jeweils funktional erweiterbaren Ausbaustufen [LL10]. Dabei wird das Gesamtkonzept gemäß des iterativen Vorgehensmodell in kleinere Teilkonzepte gegliedert und ein BeOut-Algorithmus für die Teilkonzepte konzipiert.

Dazu werden in den folgenden Abschnitten einzelne Restriktionen des in Kapitel 5 konstruierten Modells entschärft oder verändert. Dann wird das Konzept so erweitert, dass es den Umgang mit dem dadurch entstehenden Problemfaktor aus Abschnitt 5.1 konzipiert. Die einzelnen Versionen der Modelle sind mit dem Kürzel V benannt. V1 steht also für das erste Modell.

6.1 V1: Verschiedenzeitige Updates

Als erster Schritt der iterativen Konzeption werden die Restriktionen (3) und (4) aus Kapitel 5.3 entschärft. Dazu wird Restriktion (3) in diesem Modell gestrichen und (4) durch (4′) ersetzt, das wie folgt definiert ist:

$$(4')\ \ t_u(A) = t_e(A) \wedge t_u(B) = t_e(B) \wedge \big(t_u(A) \neq t_u(B) \leftrightarrow t_e(A) \neq t_e(B)\big)$$

Im Modell V1 finden also Appupdates und Busupdates nicht mehr zeitgleich statt. Allerdings schließt das Modell V1 weiterhin Latenzen aus.

Darüber hinaus fordert V1, dass die Busupdates in regelmäßigen Zeitabständen stattfinden. Damit ergibt sich in diesem Modell auch eine neue Variable für die zeitliche Differenz zwischen eines App- und eines Busupdates, die im folgenden allgemein definiert

wird, um im weiteren Verlauf auch für andere zeitlichen Differenzvergleiche verwendet werden zu können:

> Die Differenz zweier Zeitpunkte von gesendeten oder empfangenen Updates ist definiert durch: $t_{diff}(p(X), q(Y)) = t_p(X) - t_q(Y)$
> Dabei beschreiben $p, q \in \{e, u\}$, ob es sich um den Zeitpunkt einer Sendung oder eines Empfangs eines Updates handelt. Für X und Y gilt die Definition aus 5.5[1].

Ein Beispieltrip mit ebenfalls regelmäßigen Appupdates im Modell V1 ist in Abbildung 6.1 zu sehen.

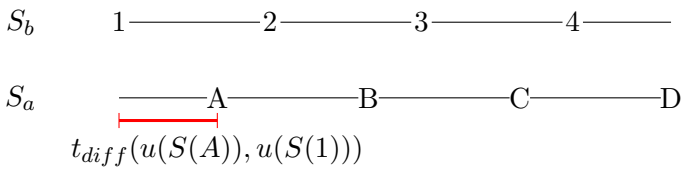

Abbildung 6.1: Trip nach V1 im Sequenzenmodell

Diese einfache Änderung in V1 hat bereits große Auswirkungen auf das Konzept. Nun ist nicht mehr klar, wann der Checkout-Server die BeOut-Überprüfung durchführen soll, schließlich wurde in 5.4 konzipiert, dass der Checkout-Server bei jedem Empfang neuer Positionsdaten die BeOut-Überprüfung durchführt. Die Unterscheidung, ob es sich dabei um ein App- oder Busupdate handelt, war nicht von Belang, da der Checkout-Server laut Modell ohnehin immer sowohl das App- als auch das Busupdate zum gleichen Zeitpunkt empfing.
Auch ist nun nicht mehr ersichtlich, welche Sequenznummern zu einem passenden Sequenzenpaar zusammengeführt werden sollten, schließlich konnten die Sequenzenpaare im atomaren Modell trivial mit den beiden jeweiligen Sequenznummer gebildet werden, die zur selben Zeit gesendet worden sind.

Dies ist also der Ausgangspunkt, um für das Konzept herauszuarbeiten, wann die BeOut-Überprüfung durchführen soll und wie die passenden Sequenzenpaare zu bilden sind. Da auch in der realen Welt die App- und Busupdates nicht zeitgleich stattfinden, wird das Konzept durchgängig in der weiteren Modellierung verwendet.

6.1.1 Auslöser des BeOut-Algorithmus

Der Checkout-Server sollte in der Praxis für viele, mehrere tausend Fahrten parallel den BeOut-Status erkennen können. Die nicht-funktionalen Anforderungen aus 3.2.2 fordern dabei eine hohe Erkennungsgeschwindigkeit und Robustheit an den Algorithmus. Die Wahl des Auslösers der BeOut-Erkennung ist also eine wichtige Entscheidung in der Konzeption, um die nicht-funktionalen Anforderungen zu erfüllen.

[1] Also X und Y können sowohl Sequenznummern oder der Wert einer Sequenznummer sein

Da die einzige Client-Server-Kommunikation zwischen der App und dem Checkout-Server besteht, bieten sich grundsätzlich zwei Möglichkeiten an.

1.) Pro Trip erstellt der Checkout-Server ein Polling-Objekt, welches in bestimmten Zeitintervallen den URA-Server nach Änderungen der VehicleID anfragt, die den Trip fährt. Dadurch könnte der Checkout-Server alle Änderungen der VehicleID relativ zeitnah erhalten. Technisch wäre diese Variante ein Tracking der Busse pro Trip.

2.) Sobald der Checkout-Server ein Update der App empfängt, fragt er den URA-Server nach Änderungen der VehicleID an. Der Zeitpunkt der Erkennung von Änderungen des Busses ist also abhängig vom Eintreffen des Appupdates.

Auch wenn 1.) auf den ersten Blick durch den Tracking-Charakter als eine gute Wahl erscheint, so zeigt es sowohl technisch als auch konzeptuell Nachteile.
Die Erstellung eines eigenen Polling-Objektes pro Trip ist konzeptuell nicht sauber. Betrachtet man eine Menge von 10000 parallelen Fahrten, existieren nach 1.) 10000 einzelne Polling-Objekte, die jeweils in regelmäßigen Zeitabständen den URA-Server anfragen. Allerdings wäre 1.) auch aus technischer Sicht kein Vorteil gegenüber 2.). Der Grund dafür liegt in den ohnehin vorhandenen Zeitabständen der Busupdates und in den Möglichkeiten, mit denen man den BeOut-Status erkennen kann. Dies wird im Folgenden erläutert.

Die einzigen konstruktiven Schlüsse, die der Checkout-Server ohne Weiteres bei der BeOut-Erkennung ziehen kann, folgen aus Appupdates, die zeitlich nahe an einem Busupdate liegen. Dazu betrachte man folgendes Beispiel:

Angenommen ein Bus aktualisiert alle 30 Sekunden seine Daten. Zum Zeitpunkt t sendet der Bus seine neueste Position. Zum Zeitpunkt $t+1$ sendet die App ihre neueste Position. Dann kann der Checkout-Server zum Zeitpunkt $t+1$ überprüfen, ob die App sich im Radius' des Busses befand[2]. War der User zu dem Zeitpunkt noch im Bus, sollte die *räumliche Distanz* der beiden Positionsdaten aufgrund der geringen *zeitlichen Distanz* nicht groß sein.
Updated die App zum Zeitpunkt $t+10$ ihre Daten erneut, so kann der Checkout-Server aus diesen Daten aber nicht zurückführen, ob sich der User zu diesem Zeitpunkt noch im Bus befand, da die zeitliche Distanz zwischen dieses Appupdates und des letzten Busupdates zu groß ist und sich somit auch die räumliche Distanz stark unterscheiden könnte. Erst Appupdates, die wenige Zeitpunkte vor oder nach dem Zeitpunkt $t+30$ empfangen worden sind, könnten für den räumlichen Vergleich wieder sinnvoll genutzt werden.
Würde der Algorithmus also durch ein Polling konzipiert und realisiert werden, gäbe es sehr viele Kommunikationen zwischen der App und dem URA-Server, die unnötig wären, da bei einem großen Teil der Anfragen keine Busupdates stattgefunden hätten und somit für die BeOut-Erkennung nicht verwendet werden könnten.

[2]Erinnerung: $t+1$ ist der Zeitpunkt t + eine Sekunde

Eine gute BeOut-Erkennung ist also von möglichst zeitnahen Updates der App und des Busses abhängig. Da die App die einzige Komponente ist, auf die der Checkout-Server Einfluss nehmen kann, bietet sich Möglichkeit 2.) für den Auslöser der BeOut-Erkennung besser an. So ist es möglich, der App den nächsten Updatezeitpunkt zu vermitteln, also die Updateintervalle der App zu verkürzen oder zu verlängern, und somit die Menge an unnötigen Kommunikationen zu reduzieren.

Mit dieser Erkenntnis werden nun zwei verschiedene Eigenschaften zum finden der passenden Sequenzenpaare vorgestellt.

6.1.2 Wahl der passenden Sequenzenpaare

In 5.5 wurde definiert, dass ein passendes Sequenzenpaar durch bestimmte Eigenschaften zusammengesetzt wird. Welche Eigenschaften das sind, wurde aber noch nicht spezifiziert. Tatsächlich ist gerade die Wahl geeigneter Eigenschaften für das Matching nicht trivial und ist ein wichtiger Faktor für die Erkennungsgeschwindigkeit des Algorithmus.
Da der BeOut-Algorithmus die passenden Sequenzenpaare für die BeOut-Erkennung verwendet, bieten sich intuitiv zwei verschiedene Varianten an, wie die Paare zusammengesetzt werden können.

1.) Ein Busupdate bildet mit dem ersten darauffolgenden Appupdate ein passendes Sequenzenpaar.

2.) Ein Busupdate bildet mit dem Appupdate ein passendes Sequenzenpaar, dessen zeitliche Differenz zum Busupdate am geringsten ist.

Der wesentliche Unterschied zwischen den beiden Varianten ist der zeitliche Bezug der Daten. Während für den BeOut-Algorithmus in der 1. Variante nur die Gegenwart relevant ist, bezieht er in der 2. Variante auch die Vergangenheit mit ein.

Dass sich der BeOut-Algorithmus nach 1.) nur auf die Gegenwart bezieht, ist nach 6.1.1 klar. Sobald der Checkout-Server ein Update der App empfängt, prüft er, ob es in der Vergangenheit ein neues Busupdate gab. Wenn nicht, wartet er auf das nächste Appupdate. Wenn doch, überprüft er die beiden Positionsdaten und bildet damit aus den beiden Positionen ein passendes Sequenzenpaar.

Inwieweit der BeOut-Algorithmus nach 2.) auch die Vergangenheit miteinbezieht, ist nicht ganz offensichtlich und ist in Abbildung 6.3 visualisiert.
Nach 6.1.1 kennt der Checkout-Server nach Empfang von A nur die Position des Busses von Sequenznummer 1. Da A das erste Update nach der 1 ist und vor der 1 kein Appupdate existiert, ist $t_u(S_{a_1}) - t_u(S_{b_1}) = min$ und bildet damit ein passendes Sequenzenpaar.
Nach Empfang des Appupdates mit Sequenznummer B erkennt der Checkout-Server nun, dass gilt: $t_u(S_{a_1}) - t_u(S_{b_2}) < t_u(S_{a_2}) - t_u(S_{b_2})$ und somit $t_u(S_{a_1}) - t_u(S_{b_2}) = min$.
Während der Checkout-Server nach 1.) also $(2, B)$ als passendes Sequenzenpaar gewählt

hätte, wählt er nach 2.) $(2, A)$ aufgrund der geringeren zeitlichen Differenz der beiden Updates.

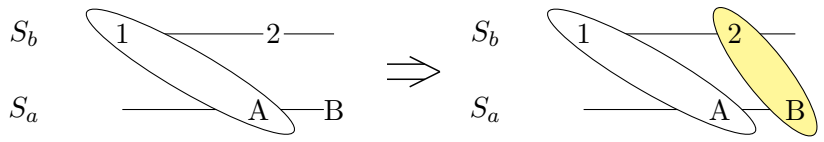

Abbildung 6.2: Wahl der passenden Sequenzenpaare nach 1.)

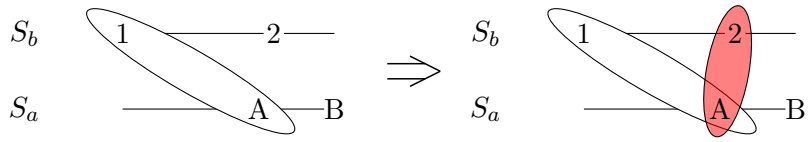

Abbildung 6.3: Wahl der passenden Sequenzenpaare nach 2.)

Wie in 6.1.1 festgestellt wurde, sollten für eine effiziente BeOut-Erkennung die Positionsdaten für die Überprüfung gewählt werden, welche zeitlich am nächsten beieinander liegen, was der Eigenschaftsdefinition 2.) entspricht. Darum wird diese Eigenschaft nun auch für die weitere Konzeption verwendet.

6.1.3 Updaterate der App

Anhand des Modells V1 wurde konzipiert, wie die BeOut-Erkennung ausgelöst wird und welche Positionsdaten miteinander verglichen werden. Allerdings kann in V1 ein weiteres Problem erkannt und behandelt werden: Die Updaterate der App.

Betrachtet man Abbildung 6.1, so fällt auf, dass die Appupdates auch bei geeigneter Wahl der passenden Sequenzenpaare gemäß Definition 2.) aus 6.1.2 im Worstcase immer genau zwischen zwei Busupdates stattfinden können. Bei einem Updateintervall von 20 Sekunden des Busses lägen also stets 10 Sekunden zwischen dem aktuellen Appupdate und dem letzten oder nächsten Busupdate, was die Erkennung erheblich erschweren würde.
Um das abzufangen, kann die Updaterate der App an die Updaterate des Busses angepasst werden. An dieser Stelle wird der trivialste Weg dazu gewählt:
Nach dem Checkin des Users updated die App jede Sekunde ihre Daten an den Checkout-Server. Sobald der Checkout-Server das erste Update des Busses während des

Trips empfangen hat, schickt er eine dementsprechende Nachricht mit dem nächsten Zeitstempel zurück, an welchem die App das nächste Mal updaten soll.
Abbildung 6.4 zeigt das Verfahren anhand des Sequenzenmodells.

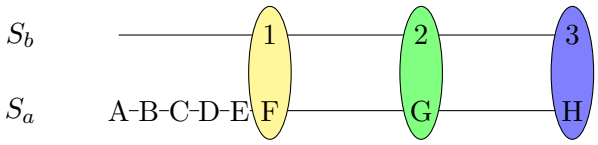

Abbildung 6.4: Eintaktung des Updateintervalls der App

Bevor Modell V2 vorgestellt wird, folgt eine Zusammenfassung der für das Gesamtkonzept relevante Punkte, die durch Modell V1 konzipiert wurden:

Zusammenfassung:

1.) Der Checkout-Server fragt nach jedem Appupdate den URA-Server an und führt ggf. den BeOut-Algorithmus aus

2.) Für eine hohe Erkennungsgeschwindigkeit werden die Positionsdaten der App und des Busses miteinander verglichen, deren zeitliche Differenz am geringsten zueinander ist

3.) Die Updaterate der App kann an die Updaterate des Busses angepasst werden

6.2 V2: Unregelmäßige Busupdates

Während in V1 die Busupdates regelmäßig versendet worden sind, modelliert V2 den Fall, dass der Bus seine Daten nicht mehr regelmäßig updated. Auch wenn im Sinne der Funktionalität der App davon ausgegangen werden sollte, dass die Updaterate zwischen zwei Updates nicht allzu groß, z.B. mehr als eine Minute, sein sollte, so wird hier der allgemeine Fall modelliert. Die einzige Bedingung ist, dass nach endlicher Zeit ein Busupdate stattfindet.
Betrachtet man noch einmal Abbildung 6.4, so fällt gleich das Problem in V2 auf: Eine einfache Eintaktung der Updaterate der App ist nicht mehr möglich.

Ein naiver Ansatz wäre, einen Wert, z.B. 10 Sekunden nach dem letzten Busupdate, zu wählen, ab dem die Updaterate der App erhöht wird, weil vermutlich bald ein Busupdate stattfinden müsste. Die Gefahr in diesem Konzept besteht aber darin, dass das Busupdate deutlich später als die gesetzte Zeit stattfindet und somit sehr viele unnötige Client-Server-Kommunikationen stattfinden, wie in Abbildung 6.5 zu sehen ist.

Eine Verbesserung dieser Problematik ergibt sich durch Ersetzen der Eintaktung der Appupdates durch ein *festes Updateintervall*. Dieses Updateintervall muss so gewählt

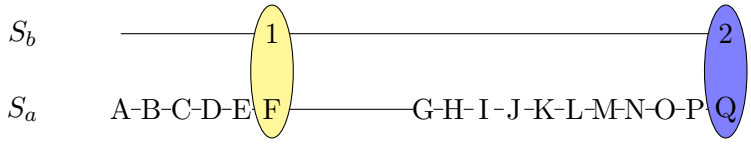

Abbildung 6.5: Naiver Ansatz der Eintaktung in Modell V2

werden, dass die *räumliche Distanz* zwischen der GPS-Position des Busses zum Zeitpunkt des Busupdates und der GPS-Position des Users zum Zeitpunkt des Appupdates nicht zu groß ist. Die räumliche Distanz der beiden Positionen ist dabei abhängig von der *zeitlichen Distanz* der beiden Updates. Dabei ist die räumliche Distanz in Relation zur zeitlichen Distanz von zwei Parametern abhängig:

1. Die Länge des Busses. Je weiter der User vom Busfahrer entfernt sitzt, desto höher ist die räumliche Distanz, sofern das Appupdate vor dem Busupdate stattfindet.

2. Die Geschwindigkeit, mit der der Bus zwischen dem Appupdate und dem Busupdate gefahren ist.

Man beachte, dass im Sinne der Robustheit zu 1. nur der genannte Fall wichtig ist, da dieser die räumliche Distanz zwischen der GPS-Position der App und der GPS-Position des Busses maximiert. Updated die App nach dem Bus, während sich der Bus fortbewegt, wird die räumliche Distanz geringer sein.
Folgendes Beispiel soll erläutert den Bezug zwischen räumlicher und zeitlicher Distanz:

Beispiel

Angenommen ein Bus fährt konstant mit 15 $^{km}/_{h}$ was etwa 4,2 $^{m}/_{s}$ entspricht. Der Bus hat eine Länge von 20 Metern, was einem Radius von 20 Metern entspricht, um den gesamten Bus abzudecken. Dann benötigt man für eine zeitliche Differenz von einer Sekunde und bei konstanter Busgeschwindigkeit einen Radius von 24,2 Metern, um den gesamten Raum, in dem sich der User zwischen einem Appupdate und einem Busupdate aufhalten konnte, abzudecken.

Wählt man eine Updaterate der App von 2 Sekunden, so würde immer maximal eine Sekunde zwischen einem App- und einem Busupdate liegen. Mit obiger Berechnung liegen also immer *maximal* 24,2 Meter zwischen den GPS-Daten der App und den GPS-Daten des Busses, nämlich genau dann, wenn der User sich in der hintersten Reihe des Busses sitzt und die App des Users eine Sekunde vor dem Bus updated.

Dieses Konzept wird in Abbildung 6.6 dargestellt.

Für die Berechnung des Radius' sind also zwei Parameter entscheidend. Die Länge des Busses und die gefahrene Strecke zwischen dem Appupdate und dem Busupdate. Diese Distanz ist von der Geschwindigkeit abhängig, mit der der Bus zwischen dem Appupdate und dem Busupdate gefahren ist.

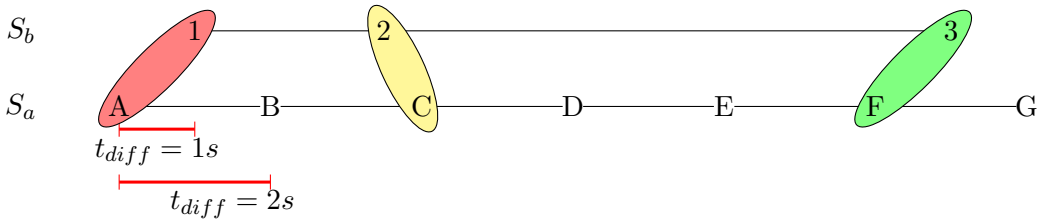

Abbildung 6.6: Konzept der festen Updatetaktung in V2

Während die Länge des Busses mit einem statischen Wert festgelegt werden kann (die längsten Busse sind 25 Meter lang), verhält sich das mit der Geschwindigkeit anders, da diese stetig während der Fahrt variiert. Die Wahl eines geeigneten Geschwindigkeitsparameters ist essentiell für die BeOut-Erkennung. Dafür bieten sich drei verschieden Möglichkeiten an:

1. Die maximale erlaubte Geschwindigkeit hält den Algorithmus robust, da dadurch garantiert ist, dass nie ein zu kleiner Radius für die BeOut-Erkennung verwendet werden würde (zumindest solange der Busfahrer nicht deutlich über der erlaubten Geschwindigkeit fährt). Allerdings hätte dies auch eine Einbuße in der Erkennungsgeschwindigkeit des BeOut-Agorithmus'.

2. Wähle für den Parameter die Durchschnittsgeschwindigkeit auf einzelnen Strecken. Dies würde die BeOut-Erkennung aufgrund eines kleineren Radius' schneller ermöglichen als Fall 1. . Allerdings hätte diese Wahl negativen Einfluss auf die Robustheit, da die Geschwindigkeit deutlich unter der gefahrenen Geschwindigkeit liegen könnte, mit der der Bus zwischen einem App- und einem Busupdate gefahren ist und somit zu kleine Radien für die BeOut-Erkennung verwendet werden würden. Insbesondere müssten viele Messungen der Durchschnittsgeschwindigkeit abhängig des Tages und der Tageszeit stattfinden, da der Verkehr davon abhängig ist.

3. Bestimme die Geschwindigkeit des Busses zwischen der letzten beiden Busupdates (oder auch zwischen dem App- und dem Busupdate) heuristisch.

Mittlerweile stellt sich heraus, dass es zwischen der Erkennungsgeschwindigkeit und der Robustheit des BeOut-Algorithmus einen Tradeoff gibt. Wird der Algorithmus robuster gehalten, leider darunter die Erkennungsgeschwindigkeit, und umgekehrt. Daher ist die heuristische Bestimmung der Geschwindigkeit die wohl interessanteste Möglichkeit, mit diesem Tradeoff umzugehen und die sich gegenüber stehenden Qualitätsanforderungen NF2 und NF3 gleichermaßen zu optimieren. Alledings wäre die Geschwindigkeit nur über die GPS-Daten der letzten zwei Updates zu bestimmen wie Fall 1. zu fehleranfällig. Man denke z.B. an Kurven, die der Bus während der beiden Updates gefahren ist. Doch ist es denkbar, dass gute heuristische Verfahren die Geschwindigkeit relativ genau bestimmen könnten. Im Rahmen dieser Arbeit konnte ein solches Verfahren nicht herausgearbeitet werden, darum wurde für die Berechnung

der Radiengröße die *maximal erlaubte Geschwindigkeit* = 18 m/s + eines geringen Offsets von 2 m/s gewählt, insgesamt also 20 m/s. Das Offset soll dem Busfahrer und der Bordtechnik einen gewissen Freiraum nach oben lassen. Für die Buslänge wurde die maximale Größe von 25 Metern gewählt, da es zur Zeit dieser Arbeit noch nicht möglich war, die Länge eines Busses durch den URA-Server zu erfahren.

Die Zeitdifferenz der App- und des Busupdates, die abhängig von der maximalen Geschwindigkeit und der maximalen Länge des Busses ist, definiert also die Größe des Radius'. Um die Zeitdifferenz nicht zu groß zu halten, muss dabei eine geeignete Update der App gewählt werden. Wählt man z.B. eine Updaterate von 4 Sekunden, liegen maximal 2 Sekunden zwischen einem App- und einem Busupdate. Mit den oben genannten Parametern ergibt sich damit eine Radiusgröße von $20 \cdot 2 + 20 = 60$ Metern. Bei einer Updaterate von 8 Sekunden, läge die Radiusgröße bereits bei $20 \cdot 4 + 20 = 100$ Metern.

Zusammenfassung

Unregelmäßige Busupdates führen dazu, dass die Eintaktung der Updaterate der App aus V1 nur noch ineffizient funktioniert. Darum wurde die Updaterate der App nun fest gewählt und der Radius, mit dem der BeOut-Erkennung die Anwesenheit des Users überprüft, wird nun dynamisch berechnet. Für die Berechnung des Radius' werden folgende Parameter verwendet:

1. Geschwindigkeit = maximale erlaubte Geschwindigkeit v
2. Länge des Busses l
3. Radiusgröße $r = l + s \cdot t_{diff}(u(A), u(B))$

6.3 V3: Latenzen der Busupdates

Latenzen der Busupdates haben keine großen Änderungen im Konzept, sondern werfen das Problem auf, dass Positionsdaten, anders als bisher, auch aus der Vergangenheit verglichen werden müssen.
Während bislang stets davon ausgegangen wurde, dass $t_u(B) = t_e(B)$ gilt, so gilt in V3 nun $t_u(B) \neq t_e(B)$. Für das Konzept bedeutete das Folgendes:
Bislang hat hat der BeOut-Algorithmus gemäß Konzeption in 6.2 beim Eintreffen eines Appupdates S_{a_i} überprüft, ob auch der Bus seine Daten geupdated hat. Wenn ja, also wenn das Update S_{b_j} empfangen worden ist, berechnete der BeOut-Algorithmus $t_{diff}(u(S_{a_i}), u(S_{b_j}))$ und $t_{diff}(u(S_{a_{i-1}}), u(S_{b_j}))$ und wählte die Updates des kleineren t_{diff} als passendes Sequenzenpaar.

Nun können folgende Fälle eintreffen:

1. Der Checkout-Server empfängt nach dem Appupdate S_{a_i} das passende Busupdate S_{b_j} mit kurzer Verzögerung. Dies entspricht exakt dem Konzept in V2.

2. Der Checkout-Server empfängt nach dem Appupdate S_{a_i} das Busupdate S_{b_j}, welches aber nicht zwischen S_{a_i} und $S_{a_{i-1}}$ gesendet worden war, sondern zwischen zwei früheren Appupdates.

3. Der Checkout-Server empfängt nach dem Appupdate S_{a_i} ein Busupdate S_{b_j}, obwohl das Busupdate $S_{b_{j-1}}$ noch nicht empfangen worden ist.

4. Der Checkout-Server empfängt nach dem Appupdate S_{a_i} zwei Busupdates, z.B. S_{b_j} und $S_{b_{j-1}}$.

Bis auf den 1. Fall, der keine Änderung zu V2 darstellt, erfordern die restlichen Szenarien eine gute Zusammenführung der richtigen Appdaten durch den BeOut-Algorithmus. Der BeOut-Algorithmus muss bei jedem Empfang eines Busupdates die beiden Appupdates finden, zwischen denen das Busupdate S_{b_i} stattgefunden hat. Dies entspricht also den Sequenznummern S_{a_j} und $S_{a_{j+1}}$, sodass gilt: $t_e(S_{a_j}) < t_e(S_{b_i}) \wedge t_e(S_{b_i}) < t_e(S_{a_{j+1}})$. Aus den beiden bildet er dann das passende Sequenzenpaar wie in 6.1.2 herausgearbeitet worden ist. Dies ist in Abbildung 6.7 visualisiert. Anstatt das einkommen Appupdate $S(D)$ mit $S(1)$ zu matchen, sucht der BeOut-Algorithmus das passende Appupdate $S(B)$ aus der Vergangenheit und matcht es mit $S(1)$.

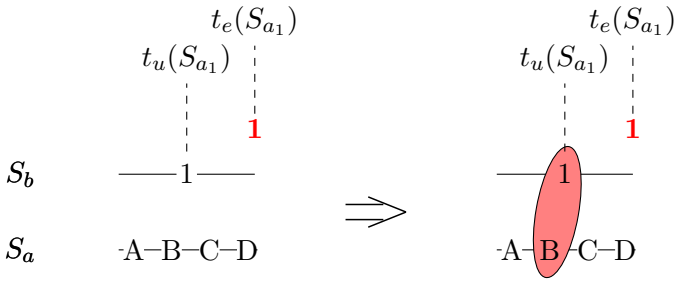

Abbildung 6.7: Finde passende Sequenzenpaare in der Vergangenheit nach V3

Zusammenfassung

Latenzen der Busupdates führen dazu, dass der BeOut-Algorithmus nun die Wahl der passenden Sequenzenpaare auch mit vergangenen App-Updates durchführen muss.

6.4 V4: Latenzen der Appupdates

Durch entschärfen der letzten zeitlichen Restriktion mit $t_u(A) \neq t_e(A)$ wird die Zuweisung der passenden Sequenzenpaare durch das Nicht-Wissen der gesendeten doch nicht empfangenen Appupdates zusätzlich erschwert. Die Problematik besteht darin, dass eine Sequenzummer S_{a_i} der App empfangen werden kann, die aber mit den bekannten Busupdates nur hohe zeitliche Differenzen hat.

Angenommen ein Bus updated alle 15 Sekunden und sein letztes Update mit Sequenznummer S_{b_j} wurde zum Zeitpunkt t gesendet und zum Zeitpunkt $t+1$ beim URA-Server empfangen. Zum Zeitpunkt $t+1$, $t+6$ und $t+11$ sendet auch die App ihre Updates mit den Sequenznummern $S_{a_i}, S_{a_{i+1}}$ und $S_{a_{i+2}}$. Für den Empfang der Updates gilt aber: $t_e(S_{a_{i+2}}) < t+15$ und $t_e(S_{a_{i+2}}) < t_e(S_{a_i}) < t_e(S_{a_{i+1}})$. Das letzte Appupdate wird also von allen Appupdates zuerst empfangen und auch ehe ein neues Busupdate stattgefunden hat.
Dann würde nach bisherigem Konzept das Update $S_{a_{i+2}}$ trotz fester Updatetaktung der App ein Sequenzenpaar mit dem Busupdate zum Zeitpunkt t bilden, obwohl für die zeitlich Differenz gilt: $t_{diff}(u(S_{a_{i+2}}), u(S_{b_j})) = 11$.

Betrachte dazu Abbildung 6.8. Beim Empfang von $S(C)$ kennt der Checkout-Server die Updates $S(A)$ und $S(B)$ noch nicht und berechnet dadurch für $t_{diff}(u(S(C)), u(S(1)))$ den minimalen Wert, obwohl t_{diff} sehr groß ist. Insbesondere bekäme er wenige Momente später das Update $S(A)$ und müsste das nun korrekt einordnen, obwohl $S(1)$ schon zu einem passenden Sequenzenpaar gehört.

Damit ergeben sich zwei Möglichkeiten, mit dieser Problematik umzugehen:

1.) Wähle auch bei großen Zeitdifferenzen $t_{diff}(u(S_{a_i}), u(S_{b_j}))$ des empfangenen App- und des dazu passenden Busupdates[3] das passende Sequenzenpaar mit dem erhaltenen Update und führe die BeOut-Überprüfung durch. Bei Erhalt eines vorherigen, besseren Updates, führt der BeOut-Algorithmus die BeOut-Erkennung erneut mit dem nun besser passenden Sequenzenpaar durch.

2.) Setze eine obere Schranke für $t_{diff}(u(S_{a_i}), u(S_{b_j}))$. Nach Erhalt eines Appupdates überprüft der BeOut-Algorithmus das t_{diff} des dazu passenden Busupdates. Ist t_{diff} zu groß, führt der BeOut-Algorithmus keine Überprüfung durch.

Auch wenn bei Möglichkeit 1. mehr Daten miteinander verglichen werden, wurde für das Konzept die 2. Möglichkeit gewählt. Der Grund dafür liegt in den potentiell sehr hohen zeitlichen Differenzen, die durch Latenzen entstehen können. In Abschnitt 6.2 wurde herausgearbeitet, dass eine hohe zeitliche Differenz ungünstig für die BeOut-Überprüfung ist, da die räumliche Differenz mit dieser variiert. Dies kann bei Möglichkeit 2. durch die obere Schranke nicht geschehen. Dadurch ist die Erkennungsgeschwindigkeit des Algorithmus zwar potentiell langsamer als nach Möglichkeit 1., dafür ist aber auch der Algorithmus auch robuster.

[3] Erinnerung: Ein Busupdate bildet mit dem Appupdate ein passendes Sequenzenpaar, dessen zeitliche Differenz zum Busupdate am geringste ist, vgl. 6.1.2

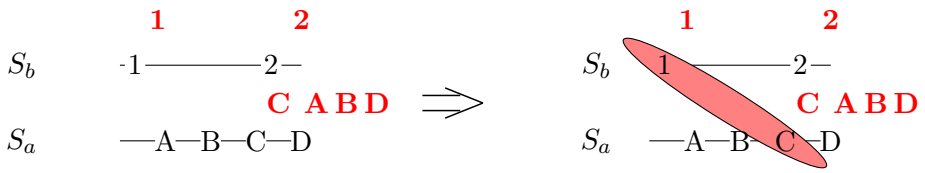

Abbildung 6.8: Falsche Wahl des Sequenzenpaares $(C, 1)$ zum Zeitpunkt $t_e(C)$

Zusammenfassung

Beim Auftreten von App- und Buslatenzen kann es trotz fester Updatetaktung der App zu hohen zeitlichen Differenzen von passenden Sequenzenpaaren kommen, wenn das Verfahren aus V4 verwendet wird. Darum wird eine obere Schranke für die zeitlich Differenz gesetzt, die zwei Sequenzen besitzen müssen, um zu einem passenden Sequenzenpaar gebildet zu werden.

6.5 V5: Fehlerbehaftung der GPS-Daten

Während die vorherigen Abschnitte V1-V4 den BeOut-Algorithmus auf Basis von zeitlichen Abhängigkeiten und Fehlerquellen konzipiert haben, folgt nun die Konzeption des Umgangs mit den Fehlern der GPS-Daten.

GPS in seinen Grundzügen zu erklären und zu verstehen, sprengt den Rahmen dieser Arbeit. Für die Arbeit entscheidend ist die Tatsache, dass ein GPS-Empfänger, im Fall dieser Arbeit also das Handy, beim Bestimmen bzw. Empfangen seiner GPS-Daten, einen Fehler der Daten abschätzen muss [Ken02]. Der zu bestimmende Fehler ist von verschiedenen Faktoren abhängig, u.a. den Satellitenuhren oder dem Phänomen, ein Radiosignal aus mehr als einer Richtung aus einem Pfad zu empfangen[4]. Der GPS-Empfänger bestimmt also seine GPS-Daten und schätzt dabei die Fehlerbehaftung der Daten durch differentiale Korrekturmethoden ab.

Dieser Fehler kann über die „Android location API“, über die die App im Prototyp die GPS-Daten ermittelt, mittels der Funktion *getAccuracy()* zurückgegeben werden [loc15]. Für den zurückgegebenen Float-Wert gilt Folgendes:
Sei x der Wert, den die Funktion *getAccuracy()* zurückgibt. Dann liegt die Wahrscheinlichkeit bei *68%*, dass sich die *tatsächliche Position* des Handys in einem Radius mit x Metern um die empfangene Position herum befindet [loc15].

Die große Problematik liegt darin, dass nicht nur das Handy, sondern auch der Bus fehlerbehaftete Daten hat. Während es möglich ist, einen ungefähren Fehler der Handy-GPS-Daten zu erfahren, ist dies beim Bus nicht der Fall ist. Auch wenn die Bus-GPS-Daten prinzipiell nicht so fehlerbehaftet sind wie die Busdaten, da der GPS-Sender auf dem Dach des Fahrerhauses befestigt ist und dadurch nicht durch die

[4] Dieses Phänomen ist auch bekannt als Multipath.

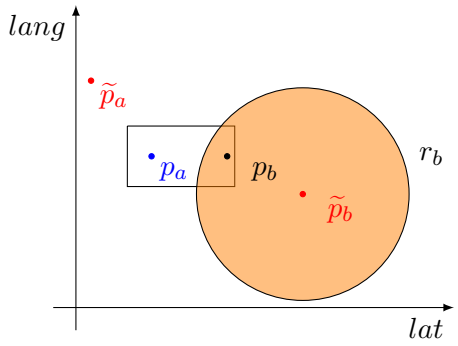

Abbildung 6.9: Bedeutung des GPS-Fehlers bei der BeOut-Erkennung

Fensterscheiben und Wände im Bus gedämpft wird, muss hier stets von einem potentiell hohen Fehler ausgegangen werden.

Was das für die BeOut-Erkennung bedeutet, ist in Abbildung 6.9 visualisiert.

Aufgrund des Fehlers in der GPS-Position des Busses, wäre selbst die tatsächliche Position des Users im Bus nicht im BeOut-Radius gewesen. Die GPS-Daten des Handys wurden sogar entgegengesetzt des Fehlers des Busses empfangen, wodurch die räumliche Distanz der auszuwertenden Daten noch weiter steigt.

Es folgt also unmittelbar, dass die *accuracy*, die im Folgenden stets GPS-accuracy genannt wird, Einfluss auf die BeOut-Erkennung nehmen muss und ein räumliches Offset für die Busposition gewählt werden muss.

Aus der Definition der GPS-accuracy folgt auch, dass durch Vergrößern des Wertes x die Wahrscheinlichkeit zunimmt, dass sich das Handy in dem Radius um die empfangene GPS-Position befunden hat. Ein Erhöhen des Wertes der GPS-Accuracy hätte also positiven Einfluss auf die Robustheit bei gleichzeitigem Verlust der Erkennungsgeschwindigkeit des BeOut-Algorithmus'.

Für das Konzept bedeutet das, dass diese Parameter in die Berechnung des BeOut-Radius' miteinfließen. In Abschnitt 6.2 wurde für die Radiusgröße die Funktion $r = l + s \cdot t_{diff}(u(A), u(B))$ definiert. Diese wird nun um die beiden Parameter erweitert. Dabei kann die GPS-Accuracy um den Faktor x im Sinne des Tradeoffs vergrößert oder verkleinert werden. Das Offset bestimmt die Toleranz der Daten. Will man auch hohe Fehler in den Bus-GPS-Daten in der BeOut-Erkennung abfangen, muss der Wert dementsprechend höher gesetzt werden, als wenn man von niedrigeren Fehlern in den Bus-GPS-Daten ausgeht. Insgesamt ergibt sich für den Radius:

$$r = l + (x \cdot accuracy) + (s \cdot t_{diff}(u(A), u(B)) + Offset$$

Zusammenfassung

Die GPS-Daten beider Seiten sind mit Fehlern behaftet. Die geschätzte Fehlerbehaftung der App-GPS-Daten kann die App dem Checkout-Server zusenden. Bei den Busdaten muss immer von einer unbekannten Fehlerbehaftung ausgegangen werden. Daraus ergibt sich die nun vollständige Funktion zur Bestimmung des BeOut-Radius'.

6.6 Das Gesamtkonzept für den Prototyp

Die Modelle V1 - V5 haben gezeigt, dass die Erkennung des Ausstiegs eines Fahrgastes mittels GPS-Daten und mobilem Internet eine sehr komplexe BeOut-Erkennung erfordert. Die einzelnen Konzepte zu den Modellen haben herausgearbeitet, dass der BeOut-Algorithmus einen dynamisch großen Radius für die BeOut-Erkennung berechnen muss und passende Sequenzenpaare abhängig von Latenzen und unterschiedlichen Updatezeitpunkten bestimmen können muss.

Das Gesamtkonzept für den BeOut-Algorithmus verwendet zwar nicht die exakt vorgestellten Methoden für die Findung der passenden Sequenzenpaare in der Vergangenheit, da auch mit der Entwicklung auf der Appseite des Prototyps zusammengearbeitet werden musste und die Zeit für die komplette Entwicklung aller Methodiken fehlte. Allerdings können diese relativ einfach ergänzt werden, da das Gesamtkonzept auf den Methoden aufbaut.

Im Gesamtkonzept sendet die App alle fünf Sekunden die GPS-Daten an den Checkout-Server. Dieser führt den BeOut-Algorithmus bei jedem Appupdate aus. Der BeOut-Algorithmus entscheidet anhand der zeitlichen Differenz des Appupdates und des aktuellsten Busupdates, ob er die BeOut-Überprüfung durchführt. Dafür dient eine obere Schranke für die maximale zeitliche Differenz, die zwei Updates haben dürfen. Ist die zeitliche Differenz zu hoch, aber fand das Appupdate *vor* dem Busupdate statt (d.h. ist eine hohe Latenz aufgetreten), überprüft der BeOut-Algorithmus die zeitliche Differenz des Appupdates und des vorherigen Busupdates und führt bei passender Differenz die BeOut-Überprüfung durch. Ansonsten führt er keine BeOut-Überprüfung durch und wartet auf das nächste Update.

Der Radius wird entsprechend der Funktion in 6.5 berechnet. Dabei wird die Funktion so gehalten werden, dass sie leicht verändert werden kann, um Änderungen der Parameter mit dem Verhalten der Qualitätsanforderungen testen und überprüfen zu können. Dies ermöglicht, dass der Tradeoff beliebig evaluiert und die Qualitätsanforderungen nach eigenem Ermessen für Testzwecke manipuliert werden können.

Das Konzept entspricht also einem Bestcase-Szenario, in welchem auf jeden Fall irgendwann passende Updates empfangen werden, mit denen die BeOut-Überprüfung durchgeführt werden kann. Auf Basis dieses Konzepts wird der neue BeOut-Algorithmus implementiert. Dazu folgt im folgenden Abschnitt der Softwareentwurf des Checkout-Servers.

7 Entwurf

> Design has become the cover for unnecessary consumption.
>
> Peter Saville

Contents

In diesem Kapitel wird der Entwurf der Softwarearchitektur des Checkout-Servers vorgestellt. Dabei folgt die Beschreibung einem Top-Down-Ansatz, wie er von Ludewig und Lichter vorgestellt wird [LL10]. Somit wird zunächst der Prototyp aus der Systemsicht visualisiert. Daraufhin folgt in Abschnitt 7.1.1 die Vorstellung der einzelnen Schichten, in die der Checkout-Server eingeteilt worden ist. In Abschnitt 7.1.2 werden die Schichten bildenden Komponenten analysiert.

Kapitel 7.2 stellt die Architektur dann aus der dynamischen Sicht vor. Hier werden die Kommunikation zwischen den Komponenten und Modulen vorgestellt. Das Kapitel schließt mit einem Aktivitätendiagramm ab.

7.1 Statische Sicht

7.1.1 Schichten

Zunächst soll ein Überblick über die Systemarchitektur geschaffen werden. Abbildung 7.1 zeigt die Architektur aus Systemsicht in Form ihrer 3-Schichten-Architektur. Man sieht, dass die Präsentationsschicht durch die App, die Anwendungsschicht durch den Checkout-Server und die Datenhaltungsschicht durch die Datenbank, die Backend-Komponente und den URA-Server gebildet werden. Der Checkout-Server stellt der Präsentationsschicht eine Schnittstelle zur Verfügung und kommuniziert selbst über Schnittstellen mit der Datenhaltungsschicht.

Der Checkout-Server wurde gemäß einer strengen Schichtenhierarchie entworfen, um in die bestehende 3-Schichten-Architektur des Systems leicht integriert werden zu können, wie Abbildung 7.2 zeigt. Dazu wurde im Entwurf der Checkout-Server so weit in einzelne Komponenten aufgeteilt, dass jede Schicht ihre eigene Zuständigkeit hat und nur auf Funktionen der Komponenten der unter ihr liegenden Schicht zugreifen kann [LL10]. Unter einer Komponente versteht man ein Bestandteil eines Systems, das ihrer Umgebung über wohldefinierte Schnittstellen eine Menge von Diensten zur

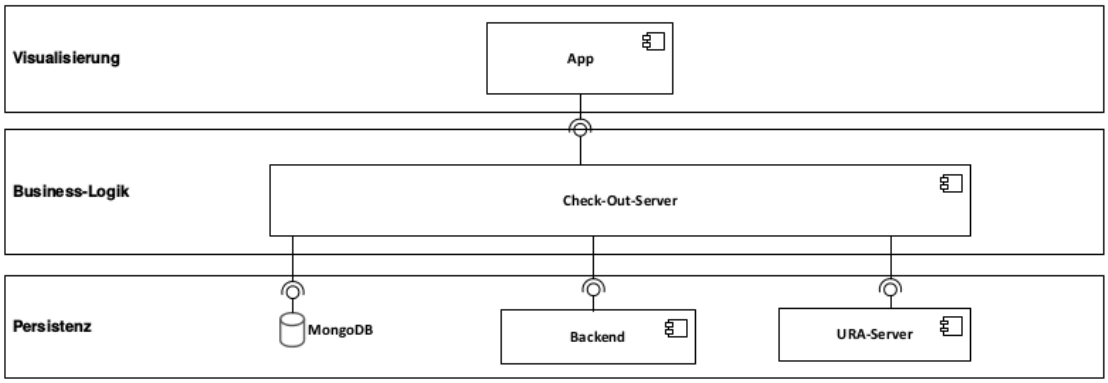

Abbildung 7.1: Schichtenmodell als Überblick

Verfügung stellt. Durch die Einhaltung einer Schichtenhierarchie wird durch die damit verbundene Funktionskapselung eine geringe Kopplung und ein hoher Zusammenhalt in der Architektur erreicht [LL10]. Diese Eigenschaften erhöhen die Wartung und das Verständnis der einzelnen Komponenten.

Neben dem Design-Prinzip der Schichtenhierarchie ist die Architektur des Checkout-Servers auch an das Architekturmuster des Model-View-Controllers (MVC) angelehnt. Ein Model-View-Controller unterteilt dabei die Architektur eines Systems in drei Komponenten: Ein Model, einer oder mehrere Views und einem Controller [LL10]. Die Model-Komponente implementiert die fachliche Funktionalität der Anwendung, eine View präsentiert mittels der Zugriffsfunktionen des Models dem Nutzer die Daten. Jeder View ist dabei ein Controller zugeordnet, der die Eingaben des Nutzers empfängt und abhängig des Nutzer-Aufrufs die View über die Änderungen informiert oder Anwendungsfunktionen des Models aufruft.

Der Entwurf der Architektur gemäß des MVC-Architekturmusters resultiert in einer klaren Trennung der Zuständigkeiten [LL10]. Das bedeutet, dass jede Komponente des Systems ihre eigene Zuständigkeit besitzt und eine klare Trennung von technischen und fachlichen Komponente, sowie Funktion unter Interaktion erreicht wird. Dies hat den Vorteil, dass die Wartung und Weiterentwicklung die einzelnen Komponenten des Systems schlank und unkompliziert gehalten werden und fördert ebenfalls dadurch die Wartung und Weiterentwicklung der einzelnen Komponenten und somit auch des Gesamtsystems [LL10].

Die Architektur wurde in 4 Schichten unterteilt und wird im Folgenden von der untersten Schicht zu der obersten hingehend beschrieben.

Datenverwaltung

Die Datenverwaltung bildet die unterste Schicht des Checkout-Servers. Gemäß des Prinzips der Trennung von Funktion und Interaktion ist diese Schicht für die Interaktion mit der Persistenzschicht zuständig [LL10]. Jegliche Zugriffe auf die

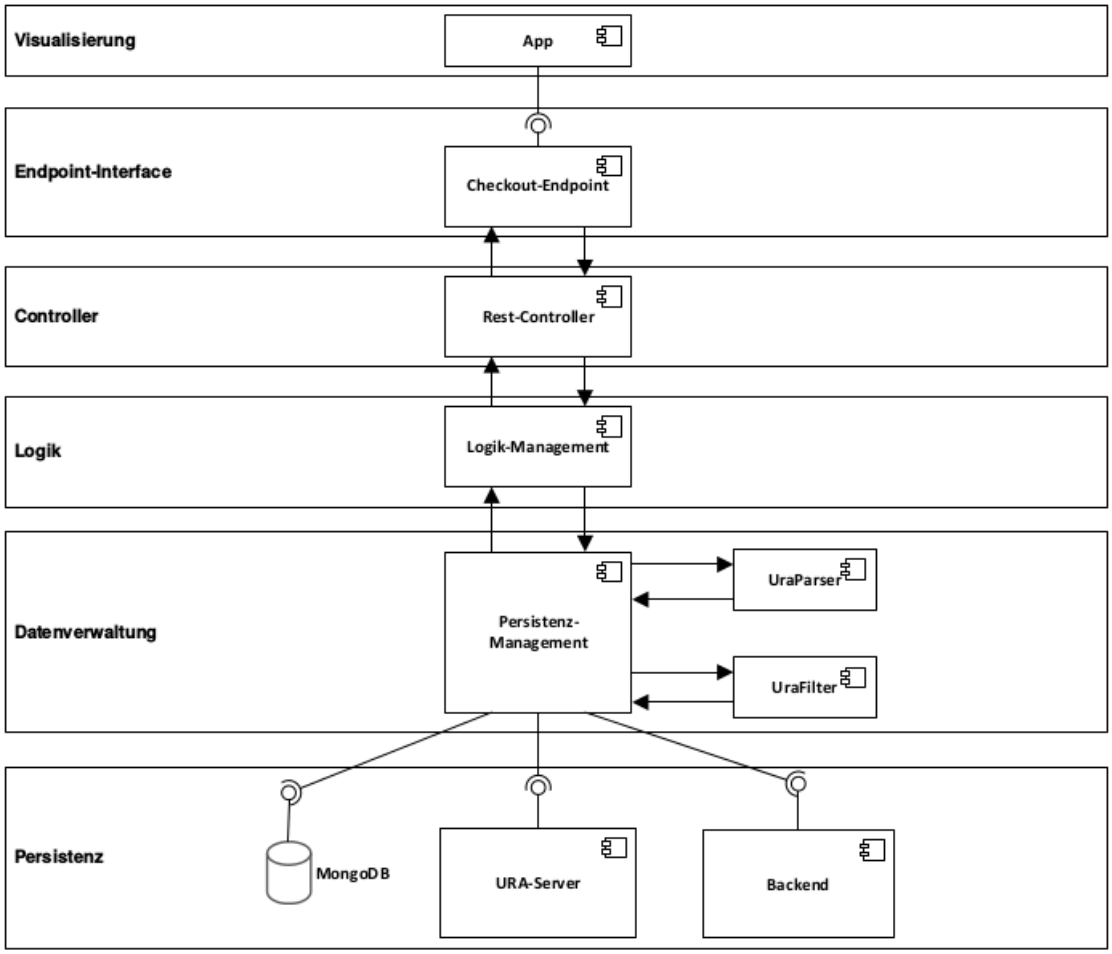

Abbildung 7.2: Schichtenmodell im Detail

Persistenzschicht finden nur über die Datenverwaltungsschicht statt. Dabei werden die Module und Komponenten, die diese Schicht bilden, nach dem Information Hiding Prinzip implementiert. Das Persistenzmanagement wird dazu durch Datenobjektmodule gebildet, die gemäß der Beschreibung von Ludewig und Lichter Art und Aufbau der Daten verstecken und über ihre Schnittstellen Operationen zur Verfügung stellen, um bestimmte Daten zu erhalten oder zu manipulieren [LL10]. Der UraParser und der UraFilter, die in der Abbildung zu sehen sind, bilden funktionale Module und werden für URA-Anfragen verwendet, um die korrekten angeforderten Daten der höheren Schicht zurückgeben zu können. Eine nähere Beschreibung erfolgt in Kapitel 8.

Logik

Die Logik-Schicht realisiert die funktionale Trennung der Zuständigkeiten innerhalb der Schichten. In der Logik-Schicht finden die Funktionen zum Erstellen eines neuen Trips eines Users und dem Update bereits existierender Trips statt. Auch die Erkennung des BeOut-Status des Users ist in dieser Schicht integriert. Für die Berechnungen fordert die Logik-Schicht die erforderlichen Daten von der Datenverwaltungsschicht an. Die Ergebnisse der Berechnungen gibt die Logik-Schicht an den nächst höher liegenden Controller weiter. Gemeinsam mit der Datenverwaltungsschicht stellen diese Schichten das Model des Model-View-Controllers dar.

Controller

Der Controller ist für die Verteilung der App-Anfragen an den Checkout-Server zuständig und bildet, wie der Name schon sagt, den Controller des Model-View-Controllers. Alle Anfragen der obersten Schicht des Checkout-Servers verteilt der Controller an die einzelnen Komponenten der Logik-Schicht. Der Controller erhält die Ergebnisse der Berechnungen aus der Logik-Schicht zurück und leitet diese an die oberste Schicht zurück.

Endpoint-Interface

Das Endpoint-Interface ist die oberste Schicht der Checkout-Servers und bildet die Schnittstelle zur Visualisierungsschicht. Jegliche Anfragen an das Endpoint-Interface werden ungeprüft und unbehandelt an die untere Schicht weitergegeben. Erst dort findet eine Validierung der Daten statt. Gleichzeitig werden die Antworten des Controllers vom Endpoint-Interface ungeprüft an die Komponente zurückgegeben, die Schnittstelle verwendet, da die Richtigkeit und Form der Antwort in den unteren Schichten gesetzt und überprüft worden ist. In Bezug auf das Modell-View-Controller-Paradigma kann das Endpoint-Interface auch als die Schnittstelle zur View betrachtet werden.

Durch die Einhaltung einer strikten Schichten-Architektur und dem MVC-Architekturmuster wurde eine Architektur erschaffen, die durch eine hohe Modularität gekennzeichnet ist, das heißt, dass das System so in Komponenten zerteilt worden ist, dass die einzelnen Komponenten möglichst unabhängig voneinander verändert und weiterentwickelt werden können [LL10]. Wie die hohe Modularität auf Komponentenebene realisiert wurde, beschreibt der folgende Abschnitt.

7.1.2 Komponenten und Schnittstellen

Die hohe Modularität auf Schichtenebene wurde auch auf Komponentenebene realisiert. Abbildung 7.3 zeigt den Checkout-Server auf Komponentenbasis. Die Logikschicht wird durch die Logik-Management-Komponente gebildet, die durch vier einzelne Komponenten gebildet wird. Auch das Persistenz-Management, das auf Schichtenebene die Datenverwaltungsschicht bildet, besteht aus drei verschiedenen Modulen, die

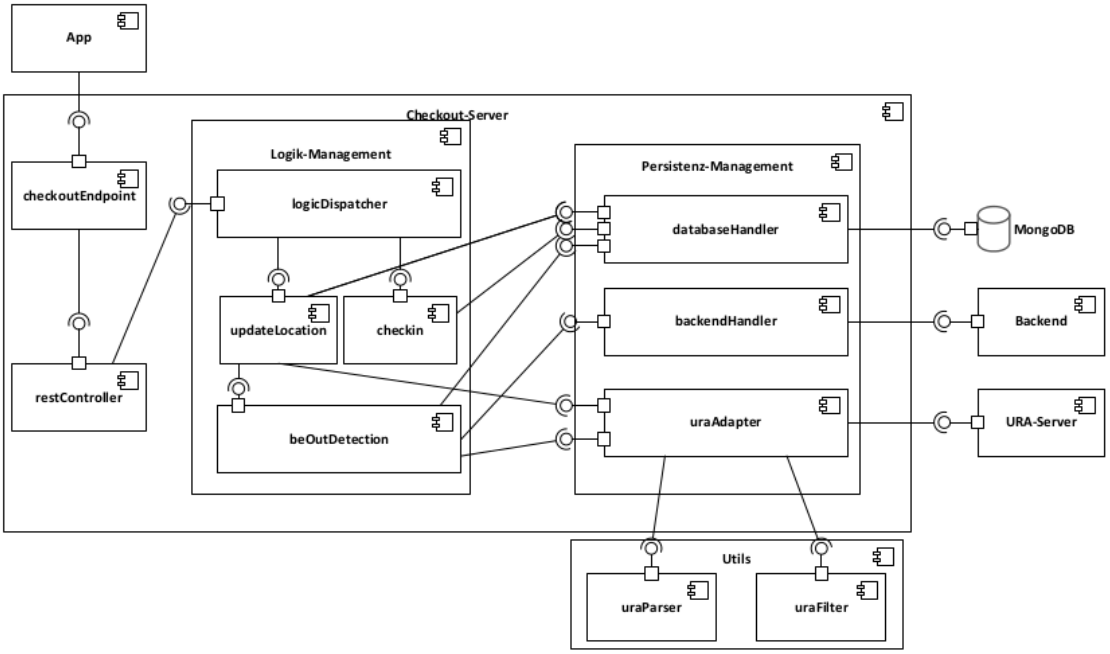

Abbildung 7.3: Übersicht der verschiedenen Komponenten im Komponentendiagramm

im Folgenden gemäß des Datenflusses von Schnittstellen-Komponente ausgehend beschrieben werden.

checkoutEndpoint

Das *checkoutEndpoint*-Modul implementiert die Schnittstelle zur App in Form einer REST-Schnittstelle. Die Anfragen der App werden an den *restController* weitergereicht. Die Antworten des *restControllers* gibt er dann mit einem passenden HTTP-Statuscode an die App zurück.

restController

Das *restController*-Modul validiert die Anfrage der App. Dabei formt es die Anfrage-Parameter so um, dass die Module des Logik-Managements diese ohne weitere Umformungen für die Berechnungen verwenden können. Die umgeformten Parameter gibt der *restController* an den *logicHandler* der Logik-Management-Komponente weiter. War die Anfrage der App ungültig, wirft der *restController* die Anfrage mit entsprechenden Fehlermeldung an den *checkoutEndpoint* zurück.

Logik-Management

Das Logik-Management wird durch vier Module gebildet. Der *logicDispatcher* wurde nach dem Design-Pattern des „Dispatchers“ entwickelt, d.h. dass der Dispatcher die Anfragen an die Logik-Handler-Komponenten empfängt und anhand der Anfrage die Anfrage an die entsprechende Logik-Handler-Komponente weitergibt [HW03]. Der logicDispatcher empfängt die Anfrage vom restController und ruft abhängig von den Parametern das *checkin*- oder das *updateLocation*-Modul auf.

Das *checkin*-Modul implementiert die Funktionalität einer Check-In-Anfrage der App. Hier wird eine neue Fahrt für einen User erzeugt und in der Datenbank abgespeichert, um bei einem App-Update die nötigen Daten aktualisieren zu können.

Das *updateLocation*-Modul implementiert die Funktionalität eines App-Updates mit neuen GPS-Daten. Dieses aktualisiert die Fahrtdaten, die durch das *checkin*-Modul erzeugt worden sind, und speichert die aktualisierte Fahrt erneut in der Datenbank ab.

Das *beOutDetection*-Modul wird wie im Konzept in Abschnitt 6.6 beschrieben ist, bei jedem Appupdate vom *locationUpdate-Modul* ausgeführt. Bei der Erkennung des BeOut-Status verwendet das beOutDetection-Modul das backendHandler-Modul aus der Datenverwaltungs-Komponente, um die Fahrt abzuschließen. Die Trennung der Funktionalitäten wird durch die klare logische Trennung eines Updates und einer BeOut-Überprüfung erreicht. Auch wenn bei einem Appupdate immer der BeOut-Algorithmus durchgeführt wird, entscheidet erst der BeOut-Algorithmus, ob das Appupdate für eine BeOut-Überprüfung verwendet werden kann. Somit kann die gesamte Logik zur Aktualisierung der Fahrtdaten auf das updateLocation-Modul und die gesamte Logik des BeOut-Algorithmus auf das beOutDetection-Modul ausgelagert werden.

Das checkin-, updateLocation- und beOutDetection-Modul verwenden für die Datenbankzugriffe und die Anfragen an den URA-Server die entsprechenden Module aus der Persistenz-Management-Komponente, wie in Kapitel 7.1.1 beschrieben.

Persistenz-Management

Das Persistenz-Management wird durch drei Module gebildet: Dem *databaseHandler*, dem *backendHandler* und dem *uraAdapter*. Jedes der drei Module kommuniziert über jeweils eine Schnittstelle mit der entsprechenden Komponente aus der Persistenzschicht aus Abbildung 7.2.

Das *databaseHandler*-Modul dient zum einen der Speicherung und Änderung von Daten in der MongoDB. Zum anderen liefert es den Modulen, die die Logik-Schicht bilden, angefragte Daten aus der Datenbank zurück.

Das *backendHandler*-Modul ist für das Senden von Daten an die Business-Backend-Komponente zuständig. Dazu filtert es die entsprechenden Daten für das Business-Backend aus den gespeicherten Fahrt-Daten heraus und sendet dem Backend die Daten in Form eines Objektes über die zur Verfügung gestellte REST-Schnittstelle.

Das *uraAdapter*-Modul implementiert die Anfragen an den URA-Server. Dazu nutzt es die REST-Schnittstelle des URA-Servers. Zum Formatieren der Antwortdaten verwendet der uraAdapter zusätzlich die Module uraParser und uraFilter aus der Utils-Komponente, die zum Parsen und Filtern der Antwort des URA-Servers fungieren. Die Auslagerung der beiden Komponenten in eine Utils-Komponente wurde aus Gründen der Wiederverwendung gewählt [LL10]. Im Rahmen dieser Arbeit wurde zusätzlich ein Hilfstool entwickelt, das in Kapitel 8 vorgestellt wird. Dieses verwendet ebenfalls die beiden Komponente uraFilter und uraParser.

Nachdem die Schichten und Komponenten vorgestellt worden sind, müssen nun noch die Entitäten vorgestellt werden.

7.1.3 Entitäten

Trip

Eine Trip-Entität repräsentiert die aktive Fahrt eines Users in einem Bus. Sie besteht aus allen relevanten Daten zur Identifizierung der Fahrtdaten, die durch ein Appupdate geändert werden müssen. Das sind ein Identifikator des Users und ein Identifikator der Trip-Entität selbst. Diesen Identifikator muss das checkin-Modul beim Erstellen einer neuen Trip-Entität erzeugen. Mit jedem Update kann die App dann den Trip-Identifikator als Parameter mitversenden, damit das updateLocation-Modul die zu aktualisierenden Daten aus der Datenbank anfordern kann.

Neben diesen Daten werden speichert die Trip-Entität auch alle Daten ab, die für den BeOut-Algorithmus nötig sind, um die BeOut-Erkennung gemäß das entworfenen Konzeptes aus Abschnitt 6.6 durchführen zu können, also sämtliche GPS-Koordinaten der empfangenen Appupdates und der Busdaten sowie die Zeitstempel der jeweiligen Updates.

URA Response

Die URA Response speichert die Daten aller Entitäten, die ihre Daten beim URA-Server abspeichern. Dazu gehören die Daten der Busse und der Busstationen. Somit entspricht eine Bus-Entität einer Teilmenge der URA Response.

Nachdem der Checkout-Server aus der statischen Sicht vorgestellt worden ist, folgt nun die dynamische Sicht auf die Architektur.

7.2 Dynamische Sicht

Die dynamische Sicht betrachtet den Checkout-Server und ihre Komponenten zur Laufzeit, d.h. wie die Komponenten zur Laufzeit zusammenarbeiten [LL10]. Dazu wird im Abschnitt 7.2.1 das Systemverhalten anhand eines Aktivitätendiagramms dargestellt.

In Abschnitt 7.2.2 wird die Kommunikation der einzelnen Komponenten zur Laufzeit mit Hilfe von Sequenzdiagrammen visualisiert.

7.2.1 Aktivitäten

Das Verhalten des Checkout-Servers zur Laufzeit teilt sich in drei Aktivitäten auf. Der Bearbeitung einer Anfrage durch den checkoutEndpoint und den restController, die

In Abbildung 7.4 ist das entworfene Aktivitätendiagramm des Checkout-Servers abgebildet. Die Aktivitäten unterteilen sich dabei in vier Aktivitäten: Anfrage bearbeiten, Check-In, Update und BeOut-Test.

Anfrage bearbeiten

Erhält der Checkout-Server eine Anfrage, prüft er zunächst die Validität der Anfrageparameter, abhängig davon, ob es sich um einen Check-In oder ein Update handelt. Sind die Daten fehlerhaft, d.h. fehlten bestimmte Anfrageparameter oder waren nicht in der angeforderten Form, wird die Anfrage verworfen. Andernfalls beginnt die Check-In- oder die Update-Aktivität.

Check-In

Im Check-In sendet der Checkout-Server für die der Anfrage entsprechenden Bus-Entität den URA-Server an und wartet auf die URA Response. Nachdem der Checkout-Server die URA-Response erhält, erstellt es die Trip-Entität und den Identifikator für die Trip-Entität und speichert die neue Trip-Entität in der Datenbank ab. Danach sendet er ein Antwort-Objekt an die App zurück und validiert die Anfrage.

Update

Im Update fordert der Checkout-Server zunächst die zu den Anfrageparametern zugehörige Trip-Entität von der Datenbank und die Bus-Entität vom URA-Server an. Sobald der Checkout-Server beide die Trip-Entität und die Bus-Entität über die URA Response erhalten hat, überprüft er über die beiden Entitäten, ob sich die Bus-Entität seit der letzten Update-Aktivität zu dieser Trip-Entität geändert hat. Dazu überprüft er die Speicherung der alten Bus-Entität in der Trip-Entität. Ist die Bus-Entität gleich der letzten Bus-Entiät, ändert der Checkout-Server die Trip-Entiät entsprechend der neuen App-Daten und speichert sie in der Datenbank ab. Andernfalls ändert sie die Trip-Entität entsprechend der neuen App- und der neuen Busdaten und speichert sie in der Datenbank ab. In beiden Fällen formatiert der Checkout-Server die Daten für den BeOut-Algorithmus entsprechend des Falles um und beginnt die BeOut-Aktivität.

BeOut-Test

Der BeOut-Algorithmus überprüft zunächst über die Parameter, ob die zeitliche Differenz der erhaltenen Parameter für eine BeOut-Überprüfung in Frage kommen. Ist

die zeitliche Differenz sowohl der neuesten als auch der älteren Appdaten zu hoch, sendet der Checkout-Server ein Antwort-Objekt an die App und validiert die Anfrage.

Ansonsten führt der Checkout-Server mit den passenden App- und Busdaten die BeOut-Überprüfung durch. Wird der BeOut-Status nicht erkannt, sendet der Checkout-Server auch hier ein Antwort-Objekt an die App und validiert die Anfrage. Wird der BeOut-Status erkannt, filtert der Checkout-Server die Daten für die Business-Backend-Komponente aus den Parametern heraus und sendet sie über die Schnittstelle des Business-Backends an das Business-Backend. Dann sendet der Checkout-Server ein Antwort-Objekt an die App und validiert die Anfrage.

Nachdem die verschiedenen Aktivitäten des Systems dargestellt worden sind, werden im folgenden Abschnitt die Kommunikation zwischen den einzelnen Komponenten bei den verschiedenen Aktivitäten beschrieben.

7.2.2 Kommunikation

Auch die Darstellung der Kommunikation zwischen den einzelnen Komponenten wurde der Übersicht halber in die Modellierung eines Check-Ins und die Modellierung eines Updates unterteilt. Die Unterteilung orientiert sich an einem kompletten Fahrtszenario. Dementsprechend beginnt die Beschreibung im Check-In-Szenario eines Users.

Um die Anzahl an Kommunikationen zwischen dem Checkout-Server und dem URA-Server so gering wie möglich zu halten, findet dabei pro Szenario jeweils nur eine Kommunikation mit dem URA-Server statt. Die Antwort des URA-Servers wird im Checkout-Server gecachet. Auch die Kommunikation mit der Datenbank wird gering gehalten, indem der Checkout-Server auch das Trip-Objekt für die Berechnungen während der Szenarien zwischenspeichert.

Dadurch wurde erreicht, dass pro Szenario nur eine URA-Anfrage und maximal zwei Datenbankabfragen stattfinden, eine zum Erhalt des Trip-Objekts und eine zum Abspeichern des aktualisierten Trip-Objekts.

Check-In

Ein Check-In ist als UML Sequenzdiagramm in Abbildung 7.5 visualisiert.

Die Kommunikation beim Check-In beginnt mit einem Anpingen der App an den Server, um den Ping der Kommunikation zu berechnen. Dann sendet die App eine Check-In-Anfrage an den Checkout-Server. Dieser fordert vom URA-Server die entsprechende Bus-Entität an. Nach dem Erhalt der Entität erstellt der Checkout-Server eine neue Trip-Entität. Dabei verwendet es den Ping aus der ersten App-Server-Kommunikation, um die unterschiedlichen Systemzeiten zwischen App und URA-Server zu bestimmen. Die Trip-Entität speichert der Checkout-Server in der Datenbank ab. Daraufhin sendet er den erzeugten Trip-Identifikator an die App zurück.

7.2.3 Update der Appdaten

Nach dem Check-In erfolgt nach einer bestimmten Zeit ein Update der App an den Checkout-Server. Bei diesem Szenario finden zwei verschiedenen Kommunikationsszenarien statt, die im Folgenden einzeln beschrieben werden.

BeOut-Status wird nicht erkannt

In Abbildung 7.6 ist der Kommunikationsverlauf zu sehen, wenn ein Appupdate stattfindet, aber der BeOut-Status nicht erkannt wird.

Nachdem der Checkout-Server eine Update-Anfrage der App erhalten hat, fordert er die Bus-Entität vom URA-Server an. Währenddessen fordert er von der Datenbank die entsprechende Trip-Entität an. Nach Erhalt der beiden Entitäten aktualisiert der Checkout-Server die Trip-Entität und speichert in der Datenbank ab. Daraufhin führt der Checkout-Server die BeOut-Erkennung. Nachdem der BeOut-Status nicht erkannt worden ist, sendet er eine Meldung an die App zurück, dass das Update erfolgreich war.

BeOut-Status wird erkannt

Dieses Teilszenario unterscheidet sich durch eine zusätzliche Kommunikation mit der Backend-Komponente vom obigen Szenario. Abbildung 7.7 stellt dieses Szenario dar.

Die Kommunikation findet bis zur BeOut-Erkennung wie im obigen Szenario statt. Nachdem der Checkout-Server nun den BeOut-Status erkannt hat, filtert es die Daten aus der Trip-Entität heraus, die der Checkout-Server zum Business-Backend sendet. Nachdem der Checkout-Server eine Validierungsantwort des Business-Backends erhält, sendet er eine Validierungsnachricht mit den gleichen Daten für die Sendung an das Business-Backend an die App zurück.

7.3 Zusammenfassung

In diesem Kapitel wurde die entworfene Softwarearchitektur aus statischer und dynamischer Sicht beschrieben.
Aus statischer Sicht wurde der Checkout-Server durch eine strikte Schichtenhierarchie und angelehnt an das MVC-Architekturmuster entworfen. Dies ermöglicht die einfache Integration des Checkout-Servers in den bestehenden Prototypen und führt zu einer geringen Kopplung, einem hohen Zusammenhalt und einer klaren Trennung der Zuständigkeiten innerhalb der einzelnen Komponenten des Checkout-Servers.

Die Komponenten wurden ebenfalls strikt modular gehalten, um auch innerhalb der Schichten-bildenden Komponenten eine geringe Kopplung und einen starken Zusammenhalt zu erhalten.

Die Anzahl an Kommunikationen zwischen dem Checkout-Server und dem URA-Server bzw. dem Checkout-Server und der Datenbank werden durch Caching der Entitäten so gering wie möglich gehalten.

Die Speicherung der Fahrtdaten wird über eine Trip-Entität realisiert. Diese enthält alle relevanten Daten für die Identifizierung der Entität, der Durchführung der BeOut-Erkennung und dem Extrahieren der vom Business-Backend benötigten Daten.

Im folgenden Kapitel wird beschrieben, wie der Software-Entwurf umgesetzt worden ist.

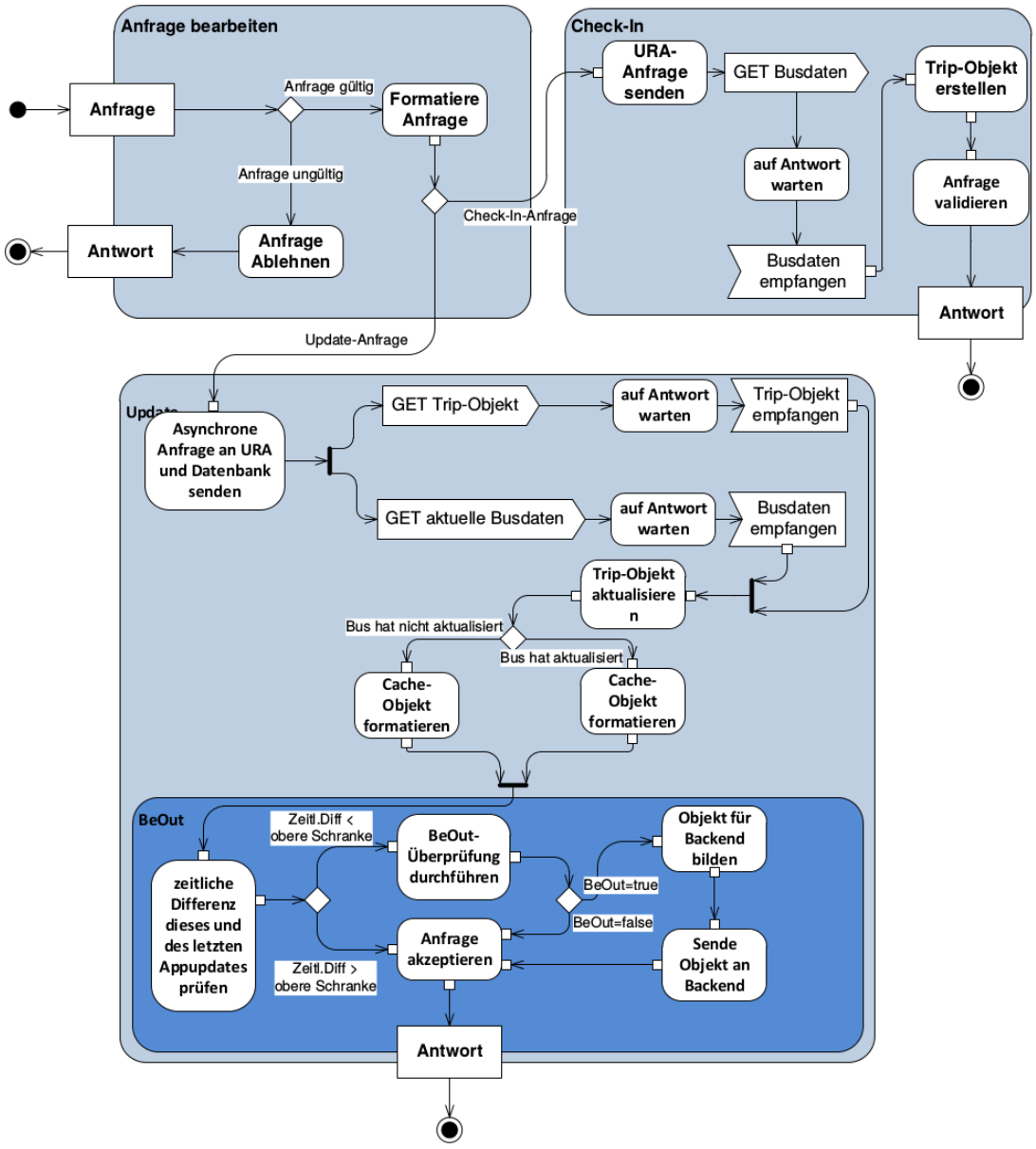

Abbildung 7.4: Übersicht der verschiedenen Komponenten im Komponentendiagramm

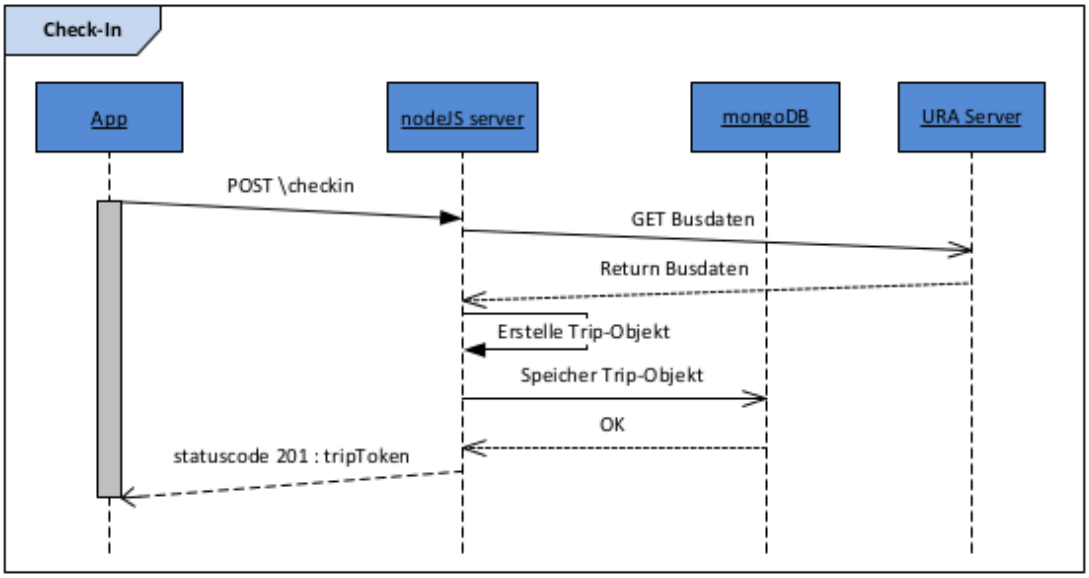

Abbildung 7.5: Kommunikation während eines Check-Ins im Sequenzendiagramm

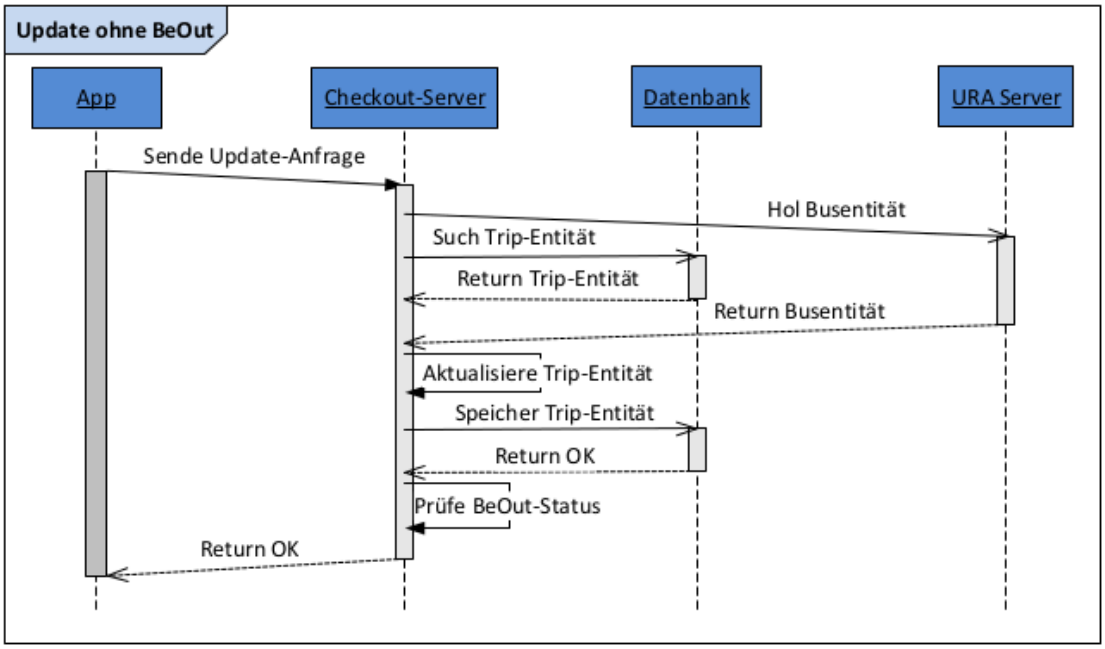

Abbildung 7.6: Kommunikation während eines App-Updates ohne Erkennung des BeOut-Status im Sequenzendiagramm

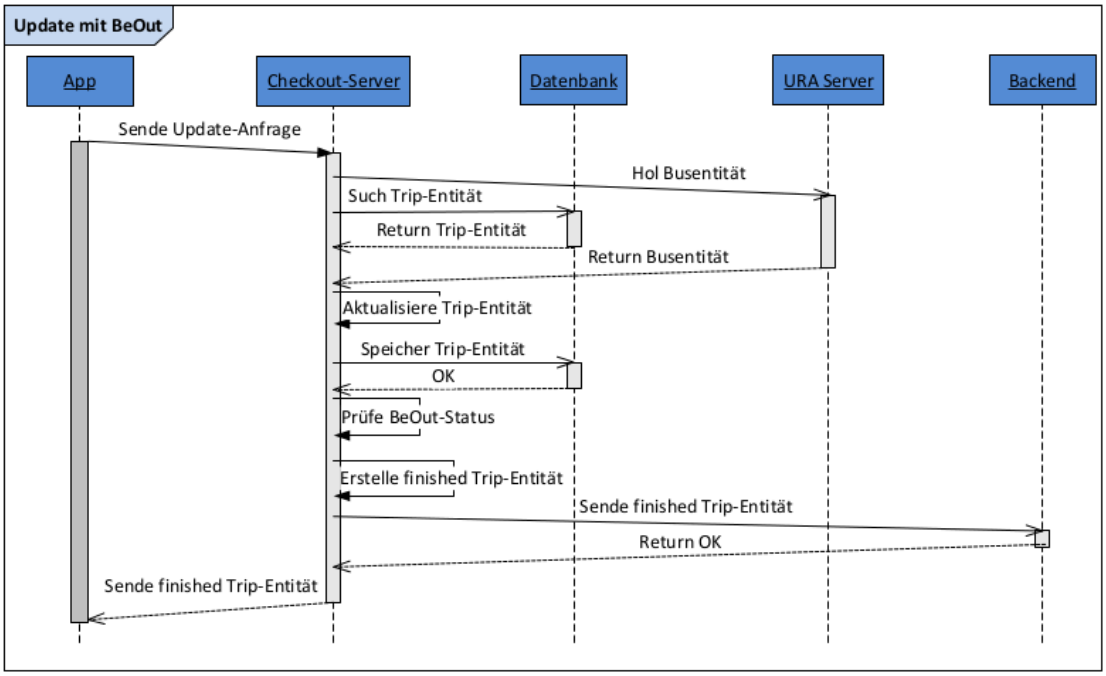

Abbildung 7.7: Kommunikation während eines App-Updates mit Erkennung des BeOut-Status im Sequenzendiagramm

8 Umsetzung

Contents

Die im Entwurf entworfene Schichtenarchitektur (siehe Abbildung 7.2 wurde durch eine klare Trennung der verschiedenen Komponenten erreicht, indem die Komponenten sehr modular gehalten worden sind. In diesem Kapitel wird beschrieben, wie die entworfene Softwarearchitektur und die hohe Modularität implementiert worden sind.

8.1 Technische Grundlagen

Bevor die genaue Umsetzung vorgestellt wird, werden ein paar technische Grundlagen geschaffen, auf denen die Umsetzung aufbaut.

8.1.1 JSON

Folgende Erläuterung ist aus dem Buch von Philipp Rieber übernommen [Rie09].
„[...] im Jahr 2005 etablierte sich aus praktischen Gründen neben Klartext und XML quasi ein weiteres Übertragungsformat für die Kommunikation via AJAX[1]: JSON, die JavaScript Object Notation. Mit Hilfe von JSON können wie bei XML strukturierte Daten übertragen werden, allerdings mit einem absoluten Minimum an Daten-Overhead für die Beschreibung der Datenstruktur. Hierfür sind statt langer Tag-Namen nur eckige und geschweifte Klammern nötig.“

Für die Umsetzung wurde das JSON-Datenformat für die Speicherung einer Trip-Entität in Form eines JSON-Trip-Objektes gewählt. Außerdem bietet sich JSON besonders für Kommunikation über REST-Schnittstellen an.

[1] „AJAX steht für *Asynchronous JavaScript And XML* und beschreibt das Zusammenspiel aus bestehenden client- und serverseitigen Webtechnologien.“ [Rie09]

8.1.2 REST

Representational State Transfer, oder kurz REST, ist ein architektonisches Paradigma, nach dem Web Services implementiert werden können. Web Services, die nach dem REST-Ansatz implementiert werden, werden auch RESTful Web Services genannt [Dai12].
Representational State Transfer wird durch mehrere Prinzipien definiert. Stefan Tilkov reduziert diese auf die fünf Kernprinzipien [Til09]

- Ressourcen mit eindeutiger Identifikation: Das entspricht der Vergabe von URIs bzw. URLs als eindeutige ID einer Ressource. Eine Ressource meint dabei ein Objekt oder eine Information, die vom Web Service nach außen hin freigegeben wird. In der URL *http://checkOut.com/trip/1234* ist „trip“ die Ressource, die durch „/trip/1234“ eindeutig identifiziert wird.

- Verknüpfung/Hypermedia: Dies wird erreicht, indem verschiedene Ressourcen des Web Services über Links miteinander verknüpft werden.

- Standardmethoden: Jede Ressource des Web Services wird durch einen Standardsatz von Methoden implementiert. Die HTTP-Spezifikation sieht dabei die Methoden GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE, CONNECT und PATCH vor. Grob gesagt unterscheiden sich die Methoden darin, inwieweit und ob eine Ressource verändert oder erzeugt bzw. gelöscht wird. Für die genaue Bedeutung der Methoden sei hier auf die offizielle HTTP-Spezifikation verwiesen [Gro99].

- Unterschiedliche Repräsentationen: Dies meint, dass der Web Service für verschiedene Anforderungen eines Clients auch dementsprechend verschiedene Repräsentationen zur Verfügung stellt. Fordert der Client eine Antwort in JSON-Format an, so soll die Ressource nicht in XML[2]-Format repräsentiert werden, und umgekehrt.

- Statuslose Kommunikation: Bei der Anfrage eines Clients zum Server soll entweder der Zustand der angefragten Ressource auf dem Client abgespeichert bzw. gecached werden, oder der Zustand der angefragten Ressource vom Server in einen neuen Ressourcenstatus umgewandelt werden.

Der Webservice, der in dieser Arbeit entwickelt worden ist, stellt eine REST-Schnittstelle für die App zur Verfügung. Das bedeutet, dass bei der Implementierung des Webservices die REST-Prinzipien eingehalten worden sind. Die REST-Schnittstelle bietet der App somit die Möglichkeit gemäß der REST-Prinzipien mit dem Webservice zu kommunizieren. Außerdem verwendet der Checkout-Server REST-Schnittstellen für die Kommunikation mit dem URA-Server und dem Business Backend.

[2]Extensible Markup Language, siehe auch http://www.w3.org/TR/REC-xml/

8.1.3 Node.js

Der Checkout-Server wurde in der JavaScript-Plattform Node.js entwickelt. Node.js ist eine serverseitige JavaScript-Plattform und wird insbesondere im Bereich der Entwicklung dynamischer Webapplikation in Verbund mit anderen Technologien, die von Datenbanken bis hin zu großen Java-Enterprise-Applikationen reichen, eingesetzt [Spr13]. Auch die Kommunikation mit anderen Webservices gestaltet sich Node.js als sehr einfach. Daneben basieren viele Konzepte von Node.js auf Asynchronität [Spr13]. Für die asynchrone Programmierung kann dabei nicht nur auf die Node.js-internen Funktionen und Module zurückgegriffen werden. Über den Node Package Manager können schnell und einfach weitere Module z.B. aus Open Source Projekten in Node.js eingebunden werden. In der Umsetzung wurden mehrere solcher Module verwendet. Eines der Module wird im Folgenden knapp beschrieben, da es das gesamte Projekt durchgängig verwendet worden ist und damit eine Art Basis bildet.

Q

Die von Kris Kowal entwickelte Bibliothek „Q“ vereinfacht die asynchrone Programmierung in Node.js, indem Objekte, sogenannte Promises, für callback-Funktionen eingesetzt werden [Spr13]. Die Verwendung von promises erhöht nicht nur die Lesbarkeit des asynchronen Codes, sie bieten insbesondere den Vorteil, einfacher getestet und gedebuggt werden zu können. Dies liegt darin begründet, dass im Gegensatz zu callbacks der Callback durch ein Objekt, nämlich das promise, repräsentiert wird.

Alle asynchronen Funktionen und Kommunikationen des Checkout-Servers verwenden diese Bibliothek für die asynchrone Implementierung.

8.1.4 MongoDB

Für die Wahl der Datenbank für die Speicherung von Trip-Objekten wurde die MongoDB ausgewählt. Dies ermöglicht eine einfache Kommunikation zwischen dem Checkout-Server und der Datenbank zur Speicherung der Trip-Objekte.

MongoDB ist die führende NoSQL Datenbank [mon15]. In MongoDB findet die Speicherung von Daten in Form von Dokumenten statt, die im BSON-Format vorliegen und gehört dadurch zu den dokumentenorientierten Datenbanken [Spr13]. BSON steht dabei für Binary JSON und ist ein binär kodiertes Serialisierung von JSON-ähnlichen Dokumenten [bso]. Das macht die MongoDB besonders bei der JavaScript-Entwicklung attraktiv. Dadurch können Daten einfach in Form von JSON-Objekten in der Datenbank abgespeichert werden und bei Datenbankanfragen zurückerhalten werden. Eine weitere attraktive Eigenschaft der MongoDB ist der einfache Umstieg von SQL zu NoSQL-Datenbanken, da viele Querys und eine große Menge relationaler Denkweisen aus SQL entnommen werden können [Wil12].

8.2 Fachliche Grundlagen

An dieser Stelle werden zwei fachliche Grundlagen geklärt, um die Umsetzung noch besser nachvollziehen zu können.

8.2.1 Die URA-Response

Eine HTTP-GET-Anfrage an den URA-Server wird durch Parameter definiert, die mittels zusätzlicher Parameter gefiltert werden können. So können zum Beispiel können die der Zeitstempel des letzten Updates, die Längengrad- und die Breitengrad-Koordinaten des Busses mit ID x angefragt werden.

Das Datenformat der URA-Response ist ein String, wobei jede Zeile des Strings einem Array-Element gleicht. Jedes Array-Element beginnt mit einer Ziffer $1-9$, die definiert, um welche Informationen es sich in dieser Zeile handelt, z.B. ob es Informationen zu den Busstationen oder zu den fahrenden Bussen sind. Aufgrund dessen wurden in der Arbeit zwei Module implementiert, die die Antwort des URA-Servers in ein JSON-Objekt parsen und die für das Trip-Update relevanten Daten aus dem geparsten JSON-Objekt filtern. Dies sind die Module *uraParser* und *uraFilter* aus Abbildung 7.3.

Eine solche URA-Response ist in Abbildung 8.1 zu sehen, die unter anderem die BusID, den Zeitpunkt ihres letzten Updates, ihre Positionsdaten und die ID ihrer nächsten Haltestelle anzeigt.

```
[4,"2.0",1428478133923]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[1,"616","98000079010001"]
[8,"616"]
[9,"616","98000079010001",1428478123768,6.072375,50.78094,"73",23,"100641","100641"]
```

Abbildung 8.1: Beispiel eines Antwort-Strings der URA-Response im Client

8.2.2 Cache-Objekte

Wie in Kapitel 7.2.2 beschrieben worden ist, sollten die Anzahl an Kommunikationen zwischen Checkout-Server und dem URA-Server und der MongoDB durch Zwischenspeicherung der Daten so gering wie möglich gehalten werden.

In der Umsetzung wurde dies durch die Kapselung aller, für die in der Logik-Schicht stattfindenden Berechnungen, relevanten Daten in JSON-Objekte umgesetzt. Neben

der Zwischenspeicherung der Informationen bietet die Kapselung den weiteren Vorteil, dass gemäß der Regeln der Codierung nur sehr wenige Parameter für Funktionsaufrufe verwendet werden [LL10]. Ein solches Caching Objekt ist in Listing 8.1 dargestellt..

```
1
2  var cacheObject = {
3    "trip": trip,
4    "currentStopCode": uraResponse['VehicleCurrentStopCode'],
5    "observationTime": uraResponse['ObservationTime'],
6    "newUserTrackingList": newUserTrackingList,
7    "newVehicleTrackingList": null,
8    "vehicleUpdated" : false,
9    "newStop": false
10 };
```

Quelltext 8.1: Caching Objekt im updateLocation-Modul

8.2.3 Das JSON-Trip-Objekt

Auch wenn dieser Abschnitt logisch eher in die Persistenzbeschreibung passt, wird die Umsetzung der Trip-Entität aus Kapitel 7 bereits hier beschrieben, da die folgenden Abschnitte Kenntnis darüber voraussetzen.

Die Trip-Entität aus Kapitel 7 wurde in Form eines JSON-Objektes (siehte 8.1.1) umgesetzt. Dies bietet insbesondere den Vorteil, dass die Daten im JSON-Format in der verwendeten Datenbank, beschrieben in Abschnitt 8.1.4, abgespeichert werden können.

Gemäß der Backend-API in 8.6 und des Konzepts aus **??** entwickelte sich das Trip-Objekt etwas komplexer als noch im ersten Prototypen. Um die BeOut-Erkennung über ältere App- und Busupdates durchführen zu können, war es nötig, zwei verschiedene Listen im Objekt anzulegen, die neben den GPS-Koordinaten eines Updates auch den Zeitstempel des Sendens des Updates beinhalten. Da der Radius auch von der GPS-Accuracy eines Appupdates abhängig ist, musste auch dieser in der dazugehörigen Liste abgespeichert werden.

In 8.2 ist ein Beispiel-Trip visualisiert. Die Speicherung der für den BeOut-Algorithmus relevanten Daten findet in der *tracking*-List statt. Pro Bushaltestelle wird in der *tracking*-Liste ein Objekt erzeugt, welches die Haltestellen-ID der Haltestelle enthält, die der Bus zum Zeitpunkt des Updates angefahren hat, identifiziert durch *vehicleCurrentStopCode*. Zu jedem *vehicleCurrentStopCode* wird eine Liste für die Busupdates und eine Liste für die Appupdates erzeugt. In diese Listen werden die Informationen aller Updates hinzugefügt, die gesendet worden sind, während der Bus diese Haltestelle angefahren hat. In der der *timeDifference* wird der Unterschied der Systemzeiten in Sekunden abgespeichert.

```
{
  "_id" : ObjectId("5524dff60391603f105e557b"),
  "timeDifference" : 1374,
  "lineId" : 98000009010001,
  "vehicleId" : 255,
  "tripToken" : "64b2ae07411e64491d2c121b7e3a4fe9",
  "userId" : "alex",
  "visitedStops" : [
  {
      "stopId" : "100025",
    "timestamp" : 1428479990383,
    "_id" : ObjectId("5524dff60391603f105e557f")
  }
  ],
  "tracking" : [
  {
      "vehicleCurrentStopCode" : "100025",
    "_id" : ObjectId("5524dff60391603f105e557c"),
    "userTrackingList" : [
    {
      "userLatitude" : 50.770728,
      "userLongitude" : 6.0752266,
      "userUpdateTimestamp" : 1428479990383,
      "gpsAccuracy" : 0,
      "_id" : ObjectId("5524dff60391603f105e557d")
    },
    {
      "userLatitude" : 50.987,
      "userLongitude" : 6.123,
      "userUpdateTimestamp" : 1428479992791,
      "gpsAccuracy" : 1,
      "_id" : ObjectId("5524dff70391603f105e5580")
    }
    ],
    "vehicleTrackingList" : [
    {
      "vehicleLatitude" : 50.772175,
      "vehicleLongitude" : 6.0769667,
      "vehicleUpdateTimestamp" : 1428479983756,
      "_id" : ObjectId("5524dff60391603f105e557e")
    }
    ]
  }
  ],
  "__v" : 1
}
```

Quelltext 8.2: JSON-Objekt des Trips

Nachdem die technischen und fachlichen Voraussetzungen beschrieben worden sind, folgt nun die Beschreibung der Umsetzung des Entwurfs. Dabei wird nach dem Bottom-Up-Verfahren vorgegangen, das heißt die Beschreibung beginnt bei den Komponenten der obersten Schicht [LL10].

8.3 Endpoint-Interface

Das Endpoint-Interface wird durch das Modul *checkoutEndpoint.js* implementiert. Dieses ist zum einen das ausführbare Modul des Servers zum anderen implementiert es, wie auch im Schichtenentwurf modelliert, die REST-Schnittstelle zum Checkout-Server. Diese ist für einen HTTP-POST zur URI /checkin in Tabelle A.1 dargestellt. In Tabelle A.2 ist die Checkout-API für einen HTTP-POST zur URI /updateLocation dargestellt.

Name	**Type**	**Description**
userId	String	Identifier of the user
vehicleId	String	Identifier of the bus in which the user checks in
timestamp	Number	Current systemtime of the device in Unix epoch time millis. This will be,used to calculate the latencies of the updates since the mobiledevice may not have the same systime as the checkout-server
ping	Number	Ping resulted in the pingtest as an Integer. ping=1000 corresponds to 1 second

Tabelle 8.1: Checkout-Server-API von POST /checkin

Name	**Type**	**Description**
userId	String	Identifier of the user
tripToken	String	Identifier of the session of the user
geocode	Number	GPS data of the new location; Format: latitude, longitude. E.g.: geocode=50.13212124,6.98345377
accuracy	Number	GPS accuracy of this update rounded up as an Integer. accuracy=null if accuracy was not given by the device
timestamp	Number	Current systemtime of the device in Unix epoch time millis

Tabelle 8.2: Checkout-Server-API von POST /updateLocation

Beide APIs implementieren nur einen HTTP-POST an die jeweilige URI. Für alle anderen HTTP-Methoden schickt der Checkout-Server eine Antwort mit HTTP-Statuscode 405, das für „unerlaubte Methode“ steht. Die Realisierung dazu ist in Listing 8.3 zu sehen.

```
router.route('/checkin')
  .post(function (req, res) {
    restController.postCheckin(req)
      .then(function (response) {
        res.status(201).send(response);
      })
      .catch(function (error) {
        res.status(400).send(error);
      });
  })
  .get(function (req, res) {
    res.status(405).send({"405": "Method not allowed"});
  })
  .put(function (req, res) {
    res.status(405).send({"405": "Method not allowed"});
  })
  .patch(function (req, res){
    res.status(405).send({"405": "Method not allowed"});
  })
  .delete(function (req, res) {
    res.status(405).send({"405": "Method not allowed"});
  });

  router.route('/updateLocation')
  .post(function (req, res) {
    restController.postUpdate(req)
      .then(function (response) {
        if (response === false) {
          res.status(204).send();
        }
        else {
          res.status(200).send(response);
        }
      })
      .catch(function (error) {
        console.log(error);
        res.status(400).send(error);
      });
  })
  .get(function (req, res) {
    res.status(405).send({msg: "Method not allowed"});
  })
  .put(function (req, res) {
    res.status(405).send({msg: "Method not allowed"});
  })
  .patch(function (req, res){
    res.status(405).send({msg: "Method not allowed"});
  })
  .delete(function (req, res) {
    res.status(405).send({msg: "Method not allowed"});
  });
```

Quelltext 8.3: checkoutEndpoint.js

8.4 restController

Wie in Kapitel 7 entworfen wurde, wurde der restController zur Überprüfung der Anfragedaten und Formatierung der richtigen Anfragedaten implementiert. Nach der Formatierung reicht der Controller die formatierten Daten an den logicDispatcher weiter. Findet der restController unkorrekte oder fehlende Anfrage-Parameter, erzeugt er mit dem *error.js*-Modul einen neuen Fehler, der den gefundenen Fehler beschreibt, und gibt den Fehler an den checkoutEndpoint zurück. Ein Ausschnitt der Erzeugung der Fehlermeldungen ist in Listing 8.4 zu sehen. Mittels eines switch-Statements wird ein JSON-Objekt der Form *{error: «error message>"}* erzeugt und an den Controller zurückgegeben.

```
1  var createError = function (error) {
2    var newError = {};
3    switch (error) {
4      case ('missingVehicleId') :
5        newError = {error: "missing parameter vehicleId"};
6        break;
7      case ('missingUserId') :
8        newError = {error: "missing parameter userId"};
9        break;
10       ...
11     case ('invalidTripToken') :
12       newError = {error: "tripToken was not a String"};
13       break;
14   }
15   return newError;
16   };
17 });
```

Quelltext 8.4: error.js

8.5 Businesslogik

Die Businesslogik, die das Logik-Management aus dem Entwurf entspricht, teilt sich in 4 Module auf: Der *logicDispatcher.js*, sowie die drei Module *checkin.js*, *locationUpdate.js* und *beOutDetection.js*.

logicDispatcher.js

Das Modul *logicDispatcher.js* wurde entsprechend des Entwurfsmusters eines Dispatchers implementiert. Der Dispatcher fürht dazu lediglich abhängig von den Parametern, die ihm übergeben werden, das entsprechende Logik-Handler-Modul für den Check-In oder das Update auf.

8.5.1 checkin.js

Das *checkin.js*-Modul kapselt die Funktionalität der Check-In-Komponente aus Kapitel 7. Gemäß des Entwurfs erzeugt das Modul ein neues JSON-Trip-Objekt und speichert es mit dem *databaseHandler*-Modul aus der Persistenzschicht in der MongoDB ab. Dabei berechnet es mit der im Parameter *ping* übergebenen Zeit die zeitliche Differenz zwischen der Systemzeit des Handys und der Systemzeit des URA-Servers. Der Trip-Identifikator wird mit dem MD5-Algorithmus erzeugt. Siehe für nähere Erklärungen zum MD5-Algorithmus auch [md592].

8.5.2 locationUpdate.js

Das *locationUpdate.js*-Modul kapselt die Funktionalität der Update-Komponente aus dem Entwurf. Es aktualisiert das zu einer userId und einem tripToken gehörige Trip-Objekt mit dem *database*-Modul aus der Persistenzschicht. Dazu erfragt es zunächst mittels des *uraAdapter*-Moduls den URA-Server nach den aktuellen Busdaten und holt sich mittels des *database*-Moduls die zuletzt abgespeicherten Busdaten des Trip-Objektes. Dann verwendet es das *database*-Modul, um abhängig eines stattgefunden Busupdates, nur in die *userTrackingList* um ein Element mit den gesendeten Appdaten einzufügen oder auch die *vehicleTrackingList* dementsprechend zu erweitern.
Dann speichert das *locationUpdate*-Modul alle für die BeOut-Erkennung benötigten Daten in einem Cache-Objekt ab und führt die BeOut-Erkennung aus.

8.5.3 beOutDetection.js

Das *beOutDetection.js*-Modul kapselt die Funktionalität der BeOut-Erkennung und ist das komplexeste Modul der Businesslogik. Hier findet der GPS-Datenabgleich mittels der Node.js-Bibliothek „geolib“ zwischen den passenden Updatepaaren statt.

An dieser Stelle sind auch die Parameter der Radien-Berechnung als solche gekennzeichnet und mit ausreichend Dokumentation versehen worden, um die Funktionsweise und den Einfluss der einzelnen Parameter auf die Qualitätsanforderungen für die IVU zu erleichtern. Die vollständige Doku ist im Anhang zu finden, an dieser Stelle ist der Übersicht halber der Auszug einer Dokumentation und die verschiedenen Parameter in 8.5 dargestellt.

Das *beOutDetection*-Modul wurde streng nach dem herausgearbeiteten Konzept implementiert. Über eine Fallunterscheidung entscheidet das Modul, ob das gesendete App-Update für die BeOut-Erkennung verwendet wird, oder ob ein älteres Update für die BeOut-Überprüfung besser in Frage kommt. Dabei berechnet es die Zeitdifferenz zwischen der neuesten App- und Busupdates und überprüft, ob das Appupdate vor oder nach diesem Busupdate stattgefunden hat. Fand das Appupdate zu spät nach dem Busupdate statt, holt es sich mittels des *database*-Moduls das zuvor abgespeicherte, also vorletzte, Element der *userTrackingList* und überprüft, ob die zeitliche Differenz zwischen App- und Bus bei diesen Appdaten passen.

```
1
2  /* CHANGING THESE PARAMETERS INFLUENCES THE QUALITY
3   * OF THE BEOUT-DETECTION!!
4   *
5   * FACTOR_OF_GPS_ERROR:
6   * Defines the factor, with which gpsAccuracy will be multiplied.
7   * In the Android-location API it is explained that if the
8   * gps-accuracy is x the probabilaty is 68% that the real location
9   * was inside a radius of x meters with the recieved
10  * (lat,lang)-pair as the center of the radius.
11  *
12  * The higher this number the more robust but less efficient
13  * the algorithm becomes
14  *
15  * Default value: 2
16  *
17  */
18
19  var TIME_DIFF = 5000;
20  var BUS_LENGTH = 30;
21  var FACTOR_OF_GPS_ERROR = 2;
22  var BUS_SPEED = 8.5;
```

Quelltext 8.5: Definition der Stellschrauben im beOutDetection.js Modul

8.6 Persistenzverwaltung

8.6.1 database.js

Im Modul *databas.js* wurden jegliche Datenbankfunktionalitäten implementiert. Dazu gehören das Abspeichern eines neuen Trips und das Aktualisieren eines bestehenden Trips. Neben diesen rein funktionalen Funktionen wurden hier auch viele Querys definiert, die die Arbeit an einem Trip-Objekt erleichtern. So gibt es nicht nur eine Query, mit der ein bestimmter Trip von der MongoDB zurückgegeben wird, sondern z.B. auch Querys, mit denen das letzte Element der letzten *vehicleTrackingList* zurückgegeben wird. Diese Querys werden an vielen Stellen in der Businesslogik-Schicht verwendet.

8.6.2 uraAdapter.js

Der uraAdapter implementiert eine HTTP-GET Anfrage an den URA-Server und filtert mit Hilfe der utils-Module *parser.js* und *queryHandler.js* die erforderlichen Informationen heraus. Die Antwort des URA-Servers ist ein String bestehend aus verschiedenen Zeilen, wobei jede Zeile einem Array-Element gleicht.
Abhängig von den Parametern, die in der Anfrage verwendet werden, sendet der URA-Server verschiedenste Zeilen zurück. Die nötigen Informationen für einen Check-In oder ein Update befinden sich irgendwo in diesem String und müssen herausgefiltert

werden. Für diese Filterung verwendet der uraAdapter die utils-Module *parser.js* und *queryHandler.js*. Der Parser übersetzt den String in ein deutlich einfacher handhabbares JSON-Objekt. Mit Hilfe des *queryHandler*-Moduls filtert der uraAdapter dann die nötigen Informationen aus dem geparsten JSON-Objekt heraus und gibt sie an das ausführende Modul zurück.

Da sowohl der uraAdapter als auch der Tool-Server, beschrieben in **??**, die Module *parser* und *queryHandler* verwenden, wurden diese Module in eine utils-Komponente gepackt.

8.6.3 backendHandler.js

Das *backendHandler*-Modul erzeugt nach Aufruf ein JSON-Objekt wie in Listing 8.6 zu sehen und sendet dieses über einen HTTP-POST Request an die Backend-API.

```
1 {
2   "startTimestamp": "2014-12-14 12:33:56",
3   "stopTimestamp": "2014-12-14 12:36:12",
4   "userCiboId": "user",
5   "startStopId": 1,
6   "endStopId": 2,
7   "busId": 1
8 }
```

Quelltext 8.6: JSON-Objekt von finishedTrip für den POST zum Backend

8.7 Tests

Die Testung teilt sich in zwei Arten von Tests auf: Ein Sysmtetest in Form eines Unit-Test und ein fachlicher Test.
Der Systemtest *server.test.js* wurde mit dem Test-Framework mocha erstellt. Hier wird jede Anfrage an jede URI des Checkout-Server, inklusive eines korrekten Check-Ins und eines korrekten Updates auf Korrektheit überprüft. In diesem Test werden auch die HTTP-Anfragen an den URA-Server nicht simuliert, sodass bei diesem Test echte Checkin- und Update-Anfragen simuliert und verarbeitet werden.

Neben dem Unit-Test wurde auch ein fachlicher Test durchgeführt. Dieser wird in Kapitel 9 erläutert.

8.8 Bus Tracking Tool

Im Rahmen der Bachelorarbeit wurde auch ein Hilfstool zur Unterstützung der Visualisierung und Konzeption des Algorithmus implementiert - Das Bus Tracking Tool. Neben der Konzeption findet das Bus Tracking Tool auch in der Testung des BeOut-Algorithmus Gebrauch, da die Visualisierung dem Verständnis der Test-Ergebnis

hilft.
Das Bus Tracking Tool ermöglicht das gezielte Tracken eines fahrenden Busses durch ein Server-Polling vom Tool-Server an den URA-Server. Die Visualisierung des Trips erfolgt über eine sich automatisch aktualisierende Google Map, die mit der Googe Maps API implementiert worden ist. Hierbei sendet der Tool-Server alle 5 Sekunden eine HTTP-GET Anfrage an den URA-Server. Hat der getrackte Bus seine Daten geupdated, wird sein neuer Standort mit einem Marker auf der Google Map gekennzeichnet und ein Radius um den Marker gesetzt, der den Busradius aus dem Konzept repräsentiert.
Auch App-Daten können über einen Pfad in die Datenbank übernommen werden. Beim nächsten Start des Tools werden die User-Positionen neben den Busmarkern als blaue Punkte auf der Google Map markiert. Wie die Ergebnisse im Bus Tracking Tool aussehen und wie es für diese Arbeit geholfen hat, wird im nachfolgenden Kapitel ersichtlich.

8.9 Zusammenfassung

Die in Kapitel 7 entworfene modulare Architektur wurde in der Umsetzung durch die Kapselung der Funktionalität der Business-Logik in einzelne Komponenten umgesetzt. Die Speicherung aller für den BeOut-Algorithmus und das Business-Backend relevanten Daten wird durch ein JSON-Objekt realisiert. Zur Unterstützung der Konzeption und Testung wurde das Bus Tracking Tool entwickelt, welches das Tracken eines Busses über die googleMaps-API in Echtzeit ermöglicht.

Dieses Tool wird im nachfolgenden Kapitel verwendet, um den Algorithmus gegen die Qualitätsanforderungen zu testen.

9 Evaluation

> If you don't like unit testing your product, most likely your customers won't like to test it either.
>
> ANONYMOUS

Contents

In der Evaluation des BeOut-Algorithmus muss der Algorithmus hinsichtlich der Qualitätsanforderungen NF2 und NF3 an den Algorithmus gemessen werden.
Dieses Kapitel beginnt gemäß des Goal-Question-Metric-Ansatzes mit der Identifizierung geeigneter Metriken, um die Qualitätsanforderungen messbar zu machen [LL10]. Der darauffolgende Abschnitt befasst sich damit, wie die Qualitätsmerkmale gemessen worden sind und wie die Evaluationsergebnisse ausgefallen sind. Da das Ziel der Bachelorarbeit die Entwicklung eines verbesserten BeOut-Algorithmus war, wird dann die Erfüllung der nicht-funktionalen Anforderungen mit denen des ersten Prototypens verglichen werden.

9.1 Die Qualitätsmerkmale

Im Kapitel 3.2 wurden die nicht-funktionalen Anforderungen an den Algorithmus beschrieben; NF2 eine hohe Erkennungsgeschwindigkeit, NF3 eine hohe Robustheit und NF4 eine hohe Testbarkeit des Algorithmus.
In NF4 wurde gefordert, dass der BeOut-Algorithmus isoliert und automatisch getestet werden kann und dabei die Qualitätsanforderungen gemessen werden sollen. Durch die entworfene Architektur des Checkout-Servers, insbesondere der Trennung der Zuständigkeiten konnte im Rahmen der Arbeit die Qualität des BeOut-Algorithmus entsprechend der Qualitätsanforderungen automatisch und mittels realer Daten gemessen werden. Dies wird in Abschnitt 9.2 erklärt. Dadurch kann die geforderte hohe Testbarkeit gemäß NF4 als erfüllt betrachtet werden.

Für die Evaluation der Erkennungsgeschwindigkeit und Robustheit werden folgende

Qualitätsmerkmale definiert:

> Für die Überprüfung der NF2 Erkennungsgeschwindigkeit wird die vergangene Zeit nach einem Ausstieg des Users bis zur BeOut-Erkennung gemessen.
>
> Für die Überprüfung der NF3 Robustheit wird gemessen, wie oft der Algorithmus während eines Trips den BeOut-Status fälschlicherweise erkannt hat. Bei der Bewertung dieses Qualitätsmerkmales muss auch die GPS-Accuracy und Update-Paare mit hohen zeitlichen Differenzen miteinbezogen werden.

Im folgenden Abschnitt wird nun vorgestellt, wie die definierten Qualitätsmerkmale gemessen worden sind.

9.2 Messung der Qualitätsmerkmale

Um die Qualitätsmerkmale für die Erkennungsgeschwindigkeit und Robustheit zu messen, wurde im Rahmen der Arbeit ein Feldtest durchgeführt. Dabei wurden die Busdaten eines Busses mittels des Bus Tracking Tools in der MongoDB geloggt. Während der Server den Bus getrackt hat, fuhr eine Testperson im getrackten Bus mit und nutzte die Android-App „GPS Logger for Android“, um gleichzeitig seine App-Daten während der Fahrt zu loggen. Nach der Fahrt wurden dann die App-Daten gemäß der Definition der *userTrackingList* eines Trip-Objektes gefiltert und umbenannt und in die MongoDB-Datenbank des Bus Tracking Tools überführt. Somit konnten die Daten auch veranschaulicht und manuell überprüft werden.

Die Testfahrt wurde mit der Linie 3B durchgeführt. Die Testperson stieg an der Haltestelle „Aachen - Misereor“ ein und an der Haltestelle „Aachen - Schanz“ aus, was einer Fahrt über 2 Haltestellen entspricht. Der GPS-Logger wurde so eingestellt, dass es gemäß des entworfenen Konzeptes alle 5 Sekunden die Handydaten aktualisiert. Nach dem Ausstieg aus der Bushaltestelle ging die Testperson wenige Meter entgegen der Fahrtrichtung zurück und blieb dann für eine Minute stehen.
Um die Daten automatisch überprüfen und evaluieren zu können, wurde mit dem Test-Framework mocha ein zweiter Automatiktest *short_trip.test.js* geschrieben.

In diesem Test wird die hohe Testbarkeit des Projektes ausgenutzt, um die realen Daten einer Fahrt im Nachhinein simulieren und auswerten zu können. Dazu werden die checkin- und die update-Komponente gemockt, um nicht die Live-Daten der Busse sondern die gesammelten Busdaten der Testfahrt im Antwort-String der URA-Response zu erhalten. Der modulare Aufbau und die Trennung der Zuständigkeiten der Architektur ermöglicht die Zuständigkeit der URA-Anfrage innerhalb der beiden Komponenten zu mocken, ohne andere Stellen des Codes anfassen zu müssen. Das heißt, dass in diesem Test das *checkin*- und *locationUpdate*-Modul regulär wie bei einer echten Fahrt verwendet werden, und diese lediglich ein manipuliertes *uraAdapter*-Modul für die HTTP-GET Anfrage an den URA-Server verwenden, welches eine HTTP-Anfrage

vortäuscht.
Um die HTTP-Anfrage vorzutäuschen, wird mit der NodeJS-Bibliothek proxyquire das *uraAdapter*-Modul, welches im *checkin*- und *updateLocation*-Modul durch ein „require“ verwendet wird, innerhalb des Testes durch eine manipulierte Funktion überschrieben. In der manipulierten Funktion wird die HTTP-Anfrage an den URA-Server mittels der NodeJS-Bibliothek nock vorgetäuscht. Dabei wird die Antwort auf die angebliche HTTP-Anfrage so abgeändert, dass es bei jedem App-Update die dazu passenden geloggten Busdaten des Feldtests als String zurückgibt.
Ein positiver Nebeneffekt der Vortäuschung der HTTP-Anfrage ist, dass dieser Test auch offline ohne eine Internetverbindung und einen laufenden Checkout-Server durchgeführt werden kann.

In Listing 9.1 ist dies am Beispiel des ersten Appupdates illustriert. Die Variable *responseOfUraRequest* wird bei jedem simulierten Appupdate lokal überschrieben und an die Funktion *sendMockRequest()* übergeben. Diese Funktion täuscht dann eine Anfrage an die URA-URL vor und gibt dabei vor, dass bei einer GET-Anfrage an den URA-Server die Antwort des URA-Servers den HTTP-Statuscode 200 hat und aus dem String *responseOfUraRequest* besteht.

```
1
2  it('mocks the 1. app update', function(done){
3
4    responseOfUraRequest = '[4,"2.0",1425998237000]\r\n[9,'+
5    (now+2*1000)+',6.0835433,50.7686867,"100010"]';
6
7      var uraStub = sendMockRequest(paramListOfAppUpdate, queryList);
8      var update =
9        proxyquire('../src/logic/logicHandlerModules/locationUpdate', {
10       '../../persistence/ura' : uraStub
11     });
12
13     ...
14   }
15
16 var sendMockRequest = function (paramList, queryList) {
17
18   ...
19
20     nock(options.hostname)
21     .persist()
22     .get(options.path)
23     .reply(200, responseOfUraRequest);
24
25   ...
26 }
```

Quelltext 9.1: Manipulation einer URA-Anfrage in short_trip.test.js

Um die Werte der Parameter der Radienberechnung nachzuvollziehen, die im Test verwendet worden sind, wird an dieser Stelle der Übersicht halber noch einmal die Funktion zur Radienberechnung dargestellt:

$$r = l + (x \cdot accuracy) + (s \cdot t_{diff}(u(A), u(B)) + Offset$$

Parameter	**Wert im Test**	**Beschreibung**
l	15	Länge des Busses in m
x	2	Faktor der GPS-Accuracy
s	8.5	Geschwindigkeit des Busses in m/s
t_{diff}	5000	ob. Schranke der zeitl. Differenz der Updates in s
Offset	15	Offset für verbesserte Robustheit

Für diese Werte wird im folgenden Abschnitt das Ergebnis des Feldtests evaluiert und analysiert, ob und inwieweit die Qualitätsanforderungen erfüllt worden sind.

9.3 Ergebnisse des Feldtests

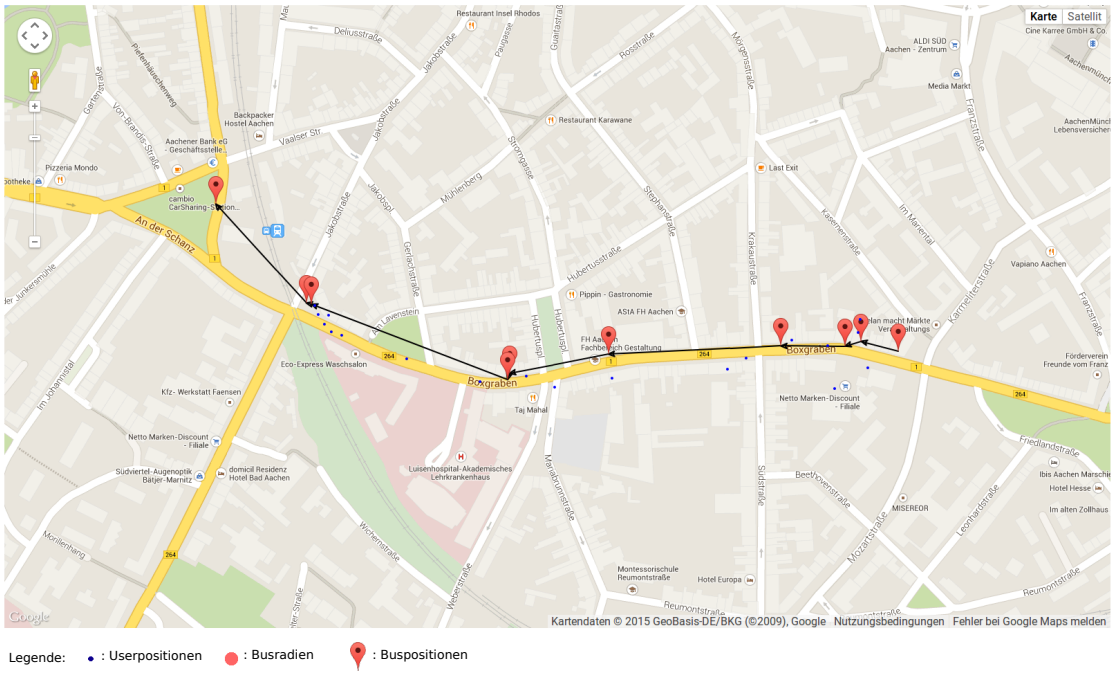

Abbildung 9.1: Feldtest im Bus Tracking Tool

In diesem Abschnitt werden die Ergebnisse des Feldtests vorgestellt und gemäß der Qualitätsanforderungen an den BeOut-Algorithmus analysiert und evaluiert. Der im obigen Abschnitt beschriebene Feldtest wird in Abbildung 9.1 im Bus Tracking Tool visualisiert.

Um die Erfüllung der Qualitätsanforderungen der NF2 Erkennungsgeschwindigkeit und der NF3 Robustheit des BeOut-Algorithmus zu analysieren, werden nun die interessanten Updatepaare (vgl. passende Sequenzenpaare aus 5.5) im Bus Tracking Tool visualisiert und miteinander verglichen.

9.3.1 Analyse der Erkennungsgeschwindigkeit

Für die Erfüllung der Erkennungsgeschwindigkeit müssen die Updates ab dem Zeitpunkt des Ausstiegs der Testperson überprüft werden. Betrachte dazu Abbildung 9.2, die die für die BeOut-Erkennung verwendeten Busradien in ihrer verwendeten Größe anzeigt.

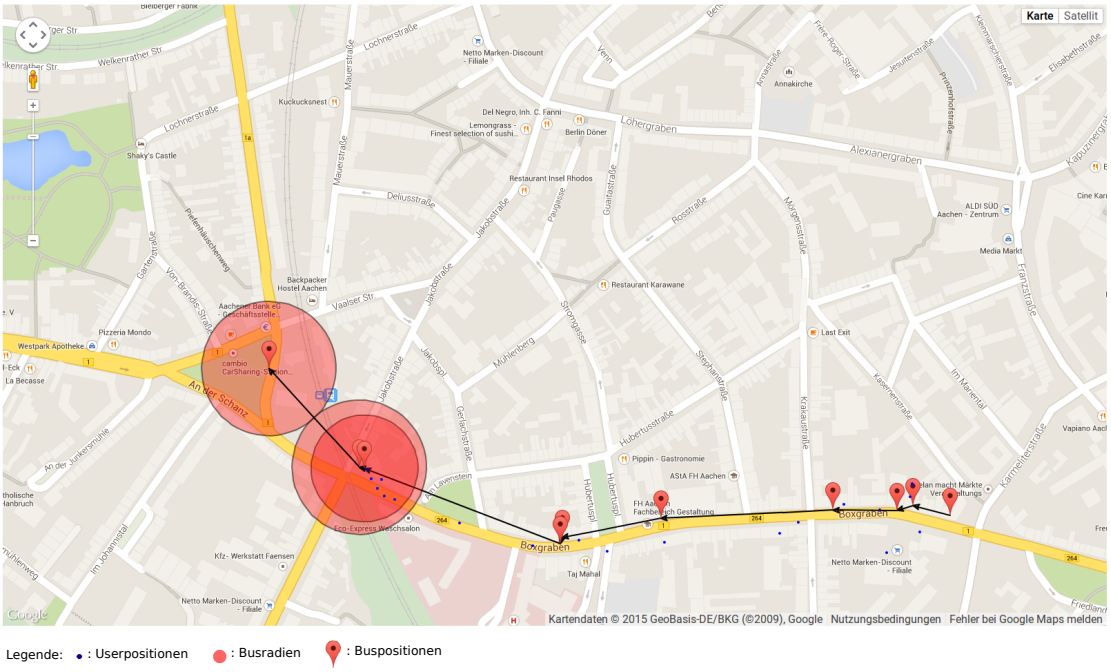

Abbildung 9.2: Analyse der Erkennungsgeschwindigkeit im Bus Tracking Tool

Der kleine Radius in der Mitte hat eine Größe von 58,5 Metern und ist der erste Radius, der nahe des Ausstiegs des Users stattgefunden hat. Der zweite Radius, der über dem kleinen Radius steht, hat eine Größe von 73.5 Metern und der letzte Radius, durch den der BeOut-Status erkannt worden ist, eine Größe von 72.5 Metern. Zu erkennen ist, dass der Radius, durch den der Ausstieg erkannt worden ist deutlich klein genug war, um keine Überschneidungen mit den Userpositionen zu haben. Während der Bus an der Haltestelle stand, lagen alle Userpositionen deutlich in den vom BeOut-Algorithmus verwendeten Radien. Davon war auch auszugehen, da die räumliche Distanz zwischen der Busposition und der Userposition sehr gering war und durch die Verwendung eines Radius' auch ein räumlicher Overhead entsteht, wie in Abbildung 5.1 in Kapitel 5 visualisiert worden ist.

Dementsprechend war der dritte Radius überhaupt der erste Radius, durch den der BeOut-Status hätte erkannt werden können, bedingt durch die geringe räumliche

Distanz an der Bushaltestelle. Der BeOut-Status wurde dabei nach etwa 40 Sekunden nach dem Ausstieg des Fahrgastes erkannt.

Inwieweit die Erkennungsgeschwindigkeit des neuen Algorithmus besser ist als im Algorithmus des ersten Prototyps wird in Abschnitt 9.4 verglichen. Zunächst folgt die Analyse der Robustheit. Dazu werden im folgenden Updates mit einander verglichen, die bezüglich der Problemfaktoren aus Kapitel 5.1 eine hohe Robustheit des BeOut-Algorithmus fordern, d.h. mit Appdaten mit hohen Fehler und einer hohen zeitliche Differenz zum Busupdate haben, mit dem sie verglichen werden.

9.3.2 Analyse der Robustheit

Der höchste Fehler, der im Feldtest aufgetreten ist, beträgt 45 Meter. Darum ist es naheliegend, sich das Verhalten des BeOut-Algorithmus bei diesem Update anzuschauen. In Abbildung 9.3 ist der „Schnappschuss“ der Fahrt im Bus Tracking Tool visualisiert.

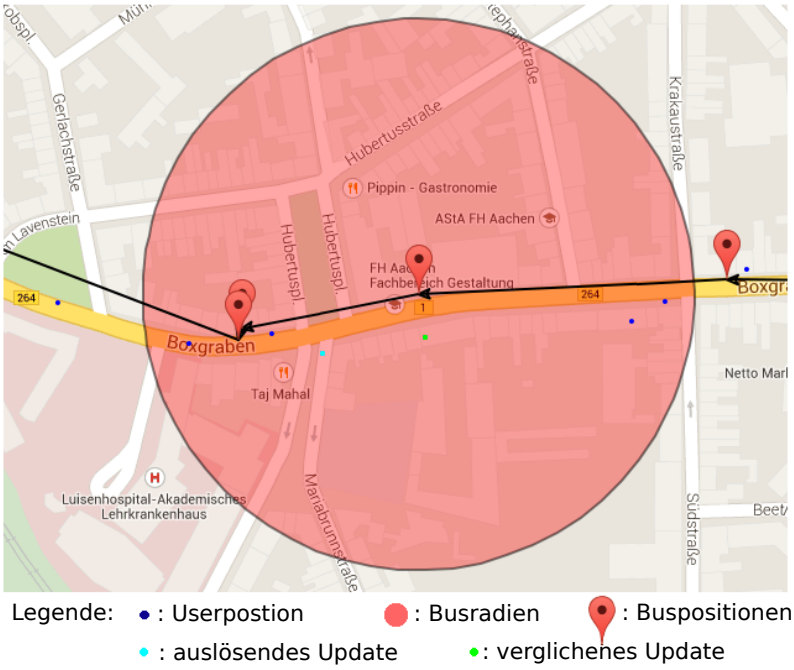

Abbildung 9.3: Radius bei hohem Fehler im Bus Tracking Tool

Der hellblaue Punkt stellt das Appupdate dar, welches den BeOut-Algorithmus ausgelöst hat. Der hellgrüne Punkt ist das Appupdate, mit welchem das Busupdate, um das der Radius liegt, verglichen worden ist, da die zeitliche Differenz zwischen dem hellblauen Appupdate und dem Busupdate zu hoch war, die zeitliche Differenz für das grüne Appupdate und dem Busupdate aber passend war.

Der Radius um die Busposition beträgt 168,5 Meter. Die Fehlerbehaftung des hellblauen Appupdates beträgt 46 Meter, die des grünen Appupdates 35 Meter. Auf den ersten Blick erscheint der Radius sehr groß, da ein großer Overhead vorherrscht. Was allerdings nicht deutlich wird und weswegen das Sequenzenmodell entwickelt und eingeführt worden ist, sind die zeitlichen Bezüge der Daten. Es scheint, als wäre das grüne Update etwa zur selben Zeit wie das Busupdate gesendet worden, mit welchem das Appupdate verglichen worden ist, da die räumliche Distanz auf der Breitengrad-Achse sehr gering ist. Tatsächlich lag die zeitliche Differenz dieser beiden Updates bei knapp 5 Sekunden, so dass die Updates noch gerade gemäß der gesetzten oberen Schranke, die auf 5 Sekunden gesetzt worden ist, überprüft werden konnten. Betrachtet man den Fall aus Sicht des Sequenzenmodells, wird diese Schwierigkeit im Umgang mit den Updates deutlicher. Der Vergleich zu dem Fall ist im Sequenzenmodell in Abbildung 9.4 visualisiert.

Aus einer geringen räumlichen Distanz der beiden Punkte kann also nicht trivial geschlossen werden, dass die Updates auch eine geringe zeitliche Distanz haben. Umgekehrt gilt der gleiche Fall. Darum muss der komplette Raum, in dem sich der User zwischen zwei Updates aufhalten konnte, durch den Radius der BeOut-Erkennung abgedeckt werden. Sollte der Bus zwischen den beiden Updates z.B. an einer Ampel stillgestanden haben, wäre der Overhead zwar sehr groß. Befand sich der Bus allerdings zwischen den beiden Updates in voller Fahrt, so deckt der Radius den Raum während der vergangenen Zeit inklusive des Fehlers der Appdaten ab.

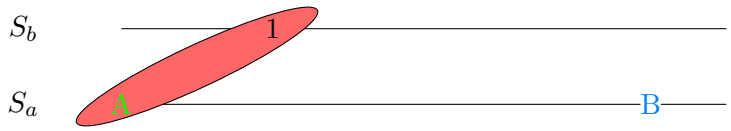

Abbildung 9.4: Zeitliche Relationen im Feldtest am Sequenzenmodell

Der Feldtest hat somit nicht nur gezeigt, dass der BeOut-Algorithmus eine hohe Erkennungsgeschwindigkeit hat (und hier kann das Wort „hoch“ durchaus als Adjektiv verwendet werden, da der BeOut-Status bereits mit dem ersten möglichen Update erkannt worden ist), sondern auch robust gegen Problemfaktoren der BeOut-Erkennung konstruiert worden ist.

Im nächsten Abschnitt wird der neue BeOut-Algorithmus mit dem alten Prototyp verglichen.

9.4 Vergleich zum ersten Prototyp

Der Vergleich mit dem BeOut-Algorithmus des ersten Prototyps wird mit den finalen Reports der drei Teams, Team Blau, Gelb und Grün, des Praktikums durchgeführt, die im Sommersemester 2014 an der Entwicklung des ersten Prototyps beteiligt waren [KLT+14], [BMFvD14], [MLR+14].

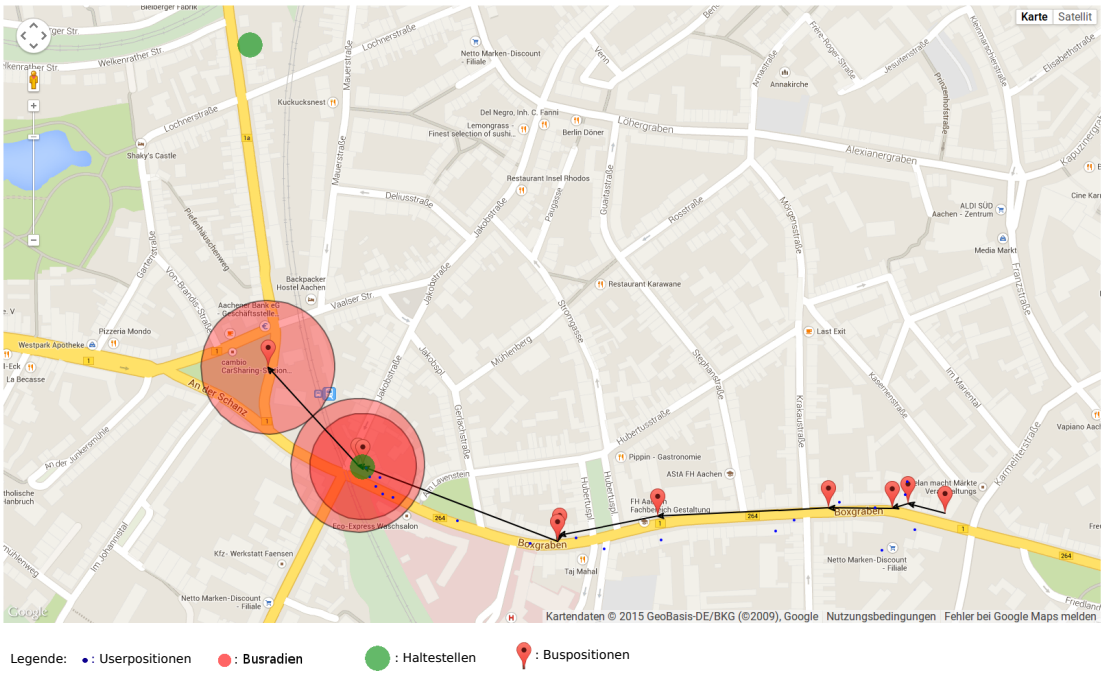

Abbildung 9.5: Vergleich des neuen zum alten BeOut-Algorithmus im Bus Tracking Tool

Da die Busse nicht durch eigene GPS-Empfänger getrackt worden sind, waren alle BeOut-Algorithmen zur Erkennung des BeOut-Status wie in Kapitel 3 beschrieben von der Ankunft der Busse bei ihrer nächsten Haltestelle abhängig. Befand sich der User im Radius' einer Haltestelle, wurde in Echtzeit überprüft, ob der User auch im Radius der nächsten Haltestelle seines Busses empfangen wird, sobald der Bus in diese Haltestelle einfährt. Die NF2 Erkennungsgeschwindigkeit des Ausstiegs war also durch die Wegstrecke des Busses von der Ausstiegshaltestelle zur nächsten Haltestelle des Busses nach unten beschränkt.

Mit Hilfe des neuen BeOut-Algorithmus können die Überprüfungen der GPS-Daten in Echtzeit geschehen. Was dies für die BeOut-Erkennung im Vergleich zum alten Prototyp bedeutet, ist in Abbildung 9.5 zu sehen, die den Feldtest mit Hilfe des BusTrackingTools visualisiert. Hier sind die vom BeOut-Algorithmus dynamisch berechneten Radien *nach* dem Ausstieg der Testperson aus dem Bus zu sehen. Die zwei grünen Punkte stellen den Ausstiegshaltepunkt bei den Busradien und die nächste Haltestelle des Busses dar.

Es ist ersichtlich, dass die Verwendung des neuen BeOut-Algorithmus die Erkennungsgeschwindigkeit deutlich verbessert hat. Im alten Prototyp hätte der Ausstieg frühestens bei der Einfahrt des Busses in die nächste Haltestelle erkannt werden können. Dies entspricht einem Unterschied von etwa ein bis zwei Minuten. Das Ziel der Arbeit, die Erkennungsgeschwindigkeit durch einen BeOut-Algorithmus zu verbessern, wurde also erfüllt.

Inwieweit die NF3 Robustheit des Algorithmus im Vergleich zum alten Prototyp zu- oder abgenommen hat, kann ohne nähere Tests mit dem alten Prototyp nicht verglichen werden, da diesbezüglich keine Auswertungen aus dem Praktikum vorliegen. Allerdings kann aus Abschnitt 9.3.2 festgehalten werden, dass die Robustheit des neuen BeOut-Algorithmus hoch gehalten worden ist.

9.5 Zusammenfassung

In diesem Kapitel wurde der neue BeOut-Algorithmus anhand eines Feldtests evaluiert. Dazu wurden die Qualitätsanforderungen gemäß gesetzter Qualitätsmerkmale gemessen und die Ergebnisse analysiert.

Aus dem Feldtest wurde deutlich, dass der neue BeOut-Algorithmus die Qualitätsanforderungen der NF2 Erkennungsgeschwindigkeit und der NF3 Robustheit einhalten kann. Die Erkennung des Ausstiegs des Fahrgastes wurde mit dem ersten Busupdate nach der Abfahrt des Busses aus der Bushaltestelle vollautomatisch erkannt und auch trotz App-Daten, die entsprechend der formulierten Problemfaktoren aus Kapitel 5.1 ungünstig für den BeOut-Algorithmus waren, fand keine fehlerhafte BeOut-Erkennung in diesem Feldtest statt.

Neben diesen positiven Ergebnissen des Feldtests wurde auch festgestellt, dass die Erkennungsgeschwindigkeit gegenüber dem BeOut-Algorithmus aus dem ersten Prototyp deutlich zugenommen hat. Somit wurden die Ziele an den BeOut-Algorithmus erfüllt.

Zwar handelt es sich bei dem durchgeführten Feldtest nur um eine kurze Fahrt, in der auch der Ausstieg der Testperson nicht entsprechend der Problemfaktoren ungünstig stattfand (die Testperson entfernte sich ein wenig von der Bushalte entgegen der Fahrtrichtung des Busses), und im zeitlichen Rahmen dieser Arbeit war es nicht mehr möglich weitere Feldtests zu machen, doch bildet dieser Test sowohl die Möglichkeit, erste Aussagen zum neuen BeOut-Algorithmus zu machen, als auch die Basis, weitere Tests zur Optimierung des in Kapitel 6.2 genannten Tradeoffs durchzuführen. Die Parameter, die die BeOut-Erkennung beeinflussen, können geändert und auf den gleichen historischen Daten erneut getestet werden. Dies ermöglicht die Optimierung des Tradeoffs zwischen Erkennungsgeschwindigkeit und Robustheit. Auch können dadurch spezifische Fahrten und Strecken getestet und analysiert werden, um z.B. die Robustheit auf Strecken mit erhöhten Problemfaktoren zu erhöhen.

Zum Ende der Arbeit folgt nun ein Resümee über die Arbeit sowie ein kurzer Ausblick auf mögliche Erweiterungen oder Verbesserungen des in dieser Arbeit entwickelten BeOut-Algorithmus'.

10 Zusammenfassung und Ausblick

Contents

In diesem Kapitel wird ein Resümee über die Arbeit und ihre Ergebnisse gezogen. Anschließend folgt die Vorstellung möglicher Erweiterungen und Verbesserungen für den neuen BeOut-Algorithmus.

10.1 Zusammenfassung der Arbeit

Mobile Ticketing-Systeme bieten eine attraktive Lösung für die Fahrgäste des öffentlichen Personennahverkehrs wie auch für ein Busunternehmen, den Kauf von Fahrtentickets und damit die Erlaubnis zur Fahrt mit den öffentlichen Transportmitteln zu vereinfachen. Der Fahrgast muss sich nicht mehr mit den Ticketpreisen und den verschiedenen Tarifen in einem Verkehrsbund auskennen, ehe er die öffentlichen Verkehrsmittel nutzen kann. Auch erhöht die Tatsache, dass vor Fahrtantritt kein Ticket mehr gekauft oder ausgewertet werden muss, die Attraktivität für die Nutzung des öffentlichen Personennahverkehrs bei Nutzern, die eher selten den ÖPNV verwenden.

Während es bereits mit der Oyster Card und Touch&Travel produktive mobile Ticketing Systeme gibt, die eine Fahrt mit öffentlichen Transportmitteln durch ein CICO-System ermöglichen, wurde in dieser Arbeit ein Schritt in Richtung eines CIBO-Systems gemacht. Durch ein inkrementelles und iteratives Vorgehensmodell konnte Algorithmus entworfen werden, der die Erkennung des Ausstiegs eines Fahrgastes durch den GPS-Datenabgleich zwischen seinem Smartphone und des Busses nicht nur ermöglicht, sondern auch zügig und robust realisiert.

Die Entwicklung am Algorithmus zeigte, dass ein BeOut-Algorithmus mittels der GPS- und mobilen Internet-Technologie einen Tradeoff zwischen der Erkennungsgeschwindigkeit und der Robustheit der automatischen Ausstiegserkennung zur Folge hat. Auf Basis dieser Erkenntnis konnte der Algorithmus so entworfen werden, dass der Tradeoff durch wenige Parameter geändert werden kann. Dabei konnte der Algorithmus und der Webservice, der den Algorithmus implementiert, durch eine strikte Schichtenhierarchie in das bestehende System integriert werden und ermöglichte mittels geeigneter Testvorbereitungen eine Test- und Messmethode, um einerseits die

Qualitätsanforderungen an den Algorithmus zu validieren und andererseits den Tradeoff messen zu können.

Der entwickelte BeOut-Algorithmus bildet nun die Basis für Erweiterungen und weitere Forschungen und Tests an diesem Ausstiegserkennungskonzept. Mögliche Erweiterungen bzw. Verbesserungen des Algorithmus werden im folgenden Abschnitt vorgestellt.

10.2 Ausblick

10.2.1 Implementierung des Sequenzenpaar-Konzepts

In Kapitel 6 wurde ein Konzept entwickelt, um mit hohen Latenzen aufgrund der mobilen Internetverbindung umgehen zu können, d.h. in der Vergangenheit passende App- und Busupdates auswählen und gemäß des BeOut-Erkennungs-Konzepts auf den BeOut-Status prüfen zu können. Aufgrund des zeitlichen Rahmens dieser Arbeit konnte das Konzept nicht komplett implementiert werden. Dieses Konzept kann und sollte bei der Weiterentwicklung des BeOut-Algorithmus implementiert werden, um die BeOut-Erkennung zu verfeinern und auch mit groben Latenzen und Fehlern robust umgehen zu können.

10.2.2 Implementierung des BeOut-Algorithmus für den URA-Stream

Im Laufe dieser Arbeit hat die IVU AG nach eigenen Angaben auch einen Stream zum URA-Server implementiert. Die Verwendung eines Streams hat den klaren Vorteil gegenüber des bisherigen Konzeptes, dass der Checkout-Server über Änderungen der Busdaten informiert wird. Damit bietet es sich an, das Konzept des neuen BeOut-Algorithmus auf den URA-Stream zu erweitern und implementieren.

10.2.3 Verwendung des Gyrosensors

Eine Möglichkeit, den BeOut-Algorithmus zu verfeinern, liegt in der zusätzlichen Verwendung des Gyrosensors. Der Gyrosensor reagiert auf Beschleunigungen, Drehbewegungen und Lageveränderungen [Bli15]. So könnte z.B. der Auslöser des BeOut-Algorithmus durch die Schritterkennung oder auch die Messung der Beschleunigung unterstützt werden. Davon könnte u.U. sowohl die Erkennungsgeschwindigkeit des BeOut-Algorithmus als auch die Robustheit des Algorithmus profitieren.

10.2.4 Unterstützung durch Buddy Tracking

Im Bereich von GPS und Handynetzwerken wird die Möglichkeit erforscht, die Position eines GPS-Empfängers durch die Position anderer GPS-Empfänger innerhalb von mobilen Netzwerken genauer zu orten [AEM+04]. Diese Technik könnte den

BeOut-Algorithmus unterstützen. So könnte man anhand der Positionen anderer Nutzer der IVU-App u.U. hohe GPS-Fehler bedingt durch die GPS-Accuracy abfangen und den Algorithmus damit robuster gestalten. Gerade mit der zusätzlichen Verwendung des Gyrosensors wäre es damit vielleicht möglich, die Erkennungsgeschwindigkeit weiter zu steigern, indem die GPS-Radien geringer gehalten werden können, da das Buddy Tracking und der Gyrosensor die Problemfaktoren, die große GPS-Radien bedingen, u.U. besser lösen könnten.

10.2.5 Verwendung der Delaunay-Triangulation

Im ersten Prototyp verwendete Team Gelb die Delaunay-Triangulation, um eine Hülle aus Dreicken mit historischen GPS-Daten als Knoten über einzelnen Strecken zu berechnen [BMFvD14]. Die Verwendung solcher Hüllen für den neuen BeOut-Algorithmus könnten u.U. hilfreich sein, um bestimmte Problemfaktoren wie hohe GPS-Fehler abzufangen. Da die Funktionalität schon einmal im IVU-Projekt implementiert worden ist, bietet sich die Erneute Verwendung der Triangulation für die Verbesserung des neuen BeOut-Algorithmus auch an.

A Sonstiger Anhang

```
/* CHANGING THESE PARAMETERS INFLUENCES THE QUALITY OF THE BEOUT-DETECTION!!

* TIME_DIFF:            Defines the biggest time-difference an app- and busupdate may have to be
*                       checked for the beOut-status.
*                       Increasing this parameter might result in higher efficiency but lower robustness
*
*                       Default: 5000
*
* BUS_LENGTH:           Defines the length of the bus. If URA implements the responseArray with
*                       also giving the bus length, this parameter can be set dynamically.
*                       Until then it should be the MAXIMUM POSSIBLE SIZE + a few meters of offset.
*                       Since Öcher Long Wajongg is 25m long, 30 should be an addequate number
*
* FACTOR_OF_GPS_ERROR:  Defines the factor, with which gpsAccuracy will be multiplied.
*                       In the Android-location API it is explained that if the gps-accuracy is x
*                       the probabilaty is 68% that the real location was inside a radius of x meters with
*                       the recieved (lat,lang)-pair as the center of the radius.
*
*                       The higher this number the more robust but less efficient the algorithm becomes
*
*                       Default: 2
*
* BUS_SPEED:            Defines the BUS_SPEED in meters per second. 8.5 corresponds to 30km/h
*                       If URA implements the responseArray with also giving the odometer,
*                       this parameter can be calculated dynamically. Until then it must be set as a constant.
*
*                       The higher this number the more efficient but less robust the algorithm becomes.
*
*                       Default: 19
*
* OFFSET:               Defines a possible offset for the radius for the use at for example difficult trips
*                       at which the gps-data become bad.
*                       Incrementing this Parameter results in a more robust but less efficient BeOut-detection.
* */

var TIME_DIFF = 5000;
var BUS_LENGTH = 25;
var FACTOR_OF_GPS_ERROR = 2;
var BUS_SPEED = 8.5;
var OFFSET = 5;
```

Abbildung A.1: Beschreibung der Parameter, die Einfluss auf die Qualitätsanforderungen an den Algorithmus haben

Name	Type	Description
userId	String	Identifier of the user
vehicleId	String	Identifier of the bus in which the user checks in
timestamp	Number	Current systemtime of the device in Unix epoch time millis. This will be,used to calculate the latencies of the updates since the mobiledevice may not have the same systime as the checkout-server
ping	Number	Ping resulted in the pingtest as an Integer. ping=1000 corresponds to 1 second

Tabelle A.1: Checkout-Server-API von POST /checkin

Name	Type	Description
userId	String	Identifier of the user
tripToken	String	Identifier of the session of the user
geocode	Number	GPS data of the new location; Format: latitude, longitude. E.g.: geocode=50.13212124,6.98345377
accuracy	Number	GPS accuracy of this update rounded up as an Integer. accuracy=null if accuracy was not given by the device
timestamp	Number	Current systemtime of the device in Unix epoch time millis

Tabelle A.2: Checkout-Server-API von POST /updateLocation

Glossary

ASEAG Kommunaler Busbetreiber des Verkehrsbereiches AVV

AVV Verkehrsbereich der Region Aachen

BeOut-Algorithmus Der in dieser Bachelorarbeit entwickelte Algorithmus, der den Check-Out des App-Users im CiBo-Projekt automatisch erkennt. Immer, wenn in dieser Arbeit von der *Erkennung des BeOut-Status* des Users die Rede ist, meint diese Formulierung, dass der BeOut-Algorithmus den Check-Out des Users als solchen erkennt oder erkannt hat.

BeOut-Status Fachliche Differenzierung zum Check-Out. Entspricht der Erkennung des Check-Outs eines Users als solchen durch den BeOut-Algorithmus. Wird der BeOut-Status vom BeOut-Algorithmus erkannt, gilt der Trip des User als beendet.

BIBO Be-In Be-Out

Check-Out Entspricht dem Ende einer Busfahrt für einen User durch seinen Ausstieg aus dem Bus. Ab dem Check-Out muss der BeOut-Status so bald es geht erkannt werden. Beachte, dass erst die Erkennung des BeOut-Status des Users die Beendigung des Trips bedeutet.

CheckOut-Server Der in dieser Bachelorarbeit in Node.JS entwickelte Server. Implementiert den Check-In einer Fahrt, die BeOut-Erkennung des Fahrgastes und die Abfertigung des Trips.

CIBO Check-In Be-Out

CiBo-Projekt Das CiBo-Projekt wird einem Fahrgast im öffentlichen Personennahverkehr die Möglichkeit geben, bis auf die Auswahl der Fahrt komplett automatisch mit einem Bus zu fahren. Ausstieg und Ticketpreisberechnung werden automatisch erkannt und berechnet.

CICO Check-In Check-Out

IVU IVU Traffic Technologies AG, Kooperationspartner dieser Arbeit.

NodeJS Eine auf Google Chromes JavaScript-Laufzeitumgebung V8 basierende, serverseitige Plattform, die insbesondere für die Entwicklung von Webservern verwendet wird.

URA-Server REST-Schnittstelle zu den Live-Daten der Busse und allen anderen Daten, die der IVU vorliegen, wie z.B. den GPS-Koordinaten der Bushaltestellen.

WIWO Walk-In Walk-Out

Literaturverzeichnis

[AEM+04] Arnon Amir, Alon Efrat, Jussi Myllymaki, Ihgcshwaran Palaniappan, and Kevin Wamplei. Buddy tracking - efficient proximity detection among mobile friends. *IEEE INFOCOM 2004*, pages 298 – 309, 2004.

[Bli15] Johannes Blischke. *Betriebssysteme für Smartphones: Android vs. iOS*. Bachelor + Master Publishing, 2015.

[BMFvD14] Jens Böttcher, Marcel Müller, Philipp Franke, and Marven von Domarus. SWP: BeOut the smooth way forward. Finaler Report, 2014.

[BMSW05] Dr. Andreas Böhm, Bernhard Murtz, Dr. Carsten Sommer, and Prof. Dr. Manfred Wermuth. Location-based ticketing in public transport. *Proceedings of the 8th International IEEE Conference on Intelligent Transportation Systems*, pages 837 – 840, 2005.

[bso] Homepage zur BSON-Spezifikation. `http://bsonspec.org/`.

[Dai12] Robert Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services* . Addison-Wesley, 2012.

[DGG07] Mark Düsener, Marcus Gemeinder, and Torben Greve. Touch & Travel: E-Ticketing per Handy. *Internationales Verkehrswesen*, 59:227–228, 2007.

[Eil14] Eilmes, Berit and Zietz, Axel and von Szombathely, Malte and Quast, Ferry and Blanckmeister, Corinna and Schmiede, Andreas. CICO-, CIBO und BIBO-basierte ÖPNV-Vertriebssysteme in Ballungsräumen weltweit. Technical report, Konsortium BLIC GmbH - KCW GmbH, März 2014.

[ELR09] Gerald Eichler, Karl-Heinz Lüke, and Britta Reufenheuser. Context information as enhancement for mobile solutions and services. *Intelligence in Next Generation Networks, 2009. ICIN 2009. 13th International Conference on. IEEE*, pages 1 – 5, 2009.

[Fin12] Klaus Finkenzeller. *RFID Handbuch*, volume 6. Hanser, 2012.

[Fra07] Fraunhofer Institut für Verkehrs- und Infrastruktursysteme. ALLFA-Ticket Elektronisches Fahrgeldmanagement für den öffentlichen Personennahverkehr. http://ivi.fraunhofer.de/content/dam/ivi/de/documents/PB_ALLFA-Ticket_deut.pdf, 2007.

[Gar12] Inc. Gartner. Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 per cent growth. http://www.gartner.com/it/page.jsp?id=2017015, 2012. Press Release.

[GD01] Thomas Gyger and Olivier Desjeux. EasyRide: active transponders for a fare collection system. *Micro, IEEE*, 21(6):36–42, 2001.

[Gro99] Network Working Group. Hypertext transfer protocol - http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html, 1999.

[Han08] Dominik Haneberg. Electronic ticketing: risks in e-commerce applications. In Dr. Paul J. J. Welfens and Dr. Ellen Walther-Klaus, editors, *Digital Excellence*, chapter 5, pages 55–66. Springer Berlin Heidelberg, 2008.

[HNS+02] Uwe Hansmann, Martin S. Nicklous, Thomas Schäck, Achim Schneider, and Frank Seliger. *Smart Card Application Development Using Java.*, volume 2. Springer, 2002.

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Ken02] Michael Kennedy. *The Global Positioning System and GIS - An Introduction*, volume 2. Taylor & Francis, 2002.

[KLT+14] Alexander Kölsch, Thomas Liepe, Jan-Hendrik Telke, Luis Rickert, Thomas Winkler, and Joel Hermanns. Finale Präsentation CheckInBeOut. Finaler Report, 2014.

[LL10] Jochen Ludewig and Horst Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse und Techniken.* dpunkt.verlag, 2010.

[LMET09] Karl-Heinz Lüke, Holger Mügge, Matthias Eisemann, and Anke Telschow. Integrated Solutions and Services in Public Transport on Mobile Devices. In Christian Erfurth, Gerald Eichler, and Volkmar Schau, editors, *9th International Conference on Innovative Internet Community Systems I2CS 2009*, pages 109–119, 2009.

[loc15] Android Location API. `http://developer.android.com/reference/android/location/Location.html`, 2015.

[Lor09] Lorenz, Helge. Be-In-Be-Out Payment Systems for Public Transport. Technical report, GWT, Juli 2009. Final Report.

[md592] Dokumentation des MD5-Algorithmus seines Entwicklers Ronald Rivest. `http://tools.ietf.org/pdf/rfc1321.pdf`, 1992.

[MLR+14] Andre Mann, Hongmei Liang, Robert Reineke, Shichen Nui, and Tibaud Kehler. Release Präsentation: CheckIn-Beout. Finaler Report, 2014.

[mon15] Homepage der MongoDB. `https://www.mongodb.org/`, 2015.

[Ord07] Christian Ordon. *Die Bedeutung des ÖPNV für die nachhaltige Sicherung der Mobilität in der Region Frankfurt Rhein-Main.* PhD thesis, Georg-August-Universität Göttingen, Göttingen, 2007.

[Oys] Transport for London. http://www.tfl.gov.uk/.

[Rie09] Philipp Rieber. *Dynamische Webseiten in der Praxis - Mit PHP 5, MySQL 5, XHTML, CSS, JavaScript und AJAX*, volume 2. mitp, 2009.

[Som07] Ian Sommerville. *Software Engineering*, volume 8. Pearson Studium, 2007.

[Spr13] Sebastian Springer. *Node.js - Das umfassende Handbuch*, volume 1. Galileo Computing, 2013.

[SV04] Jochen Schiller and Agnés Voisard. *Location-Based Services.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2004.

[Til09] Stefan Tilkov. *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien.* dpunkt.verlag, 2009.

[Wil12] Mike Wilson. *Buildung Node Applications with MongoDB and Backbone.* O'Reilly, 2012.

[Wir14] Matthias Wirtz. *Flexible Tarife in elektronischen Fahrgeldmanagementsystemen und ihre Wirkung auf das Mobilitätsverhalten*, volume 71. KIT Scientific Publishing, 2014.

[ZP08] Axel Zietz and Thomas Petersen. Die OysterCard – ein Erfolgsmodell?! Einführung des eTicketings in London als PPP-Projekt – auch ein Modell für Deutschland? *Verkehr und Technik*, 3:107–112, 2008.