Ein Demonstrator für ein Peer-To-Peer-basiertes Verkehrsinformationssystem

Bachelorarbeit

von

Norbert Goebel

aus

Neuss

vorgelegt am 14.11.2008

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

November 2008

Betreuer:
Jedrzej Rybicki, M.Sc.

Danksagung

Ich möchte mich bei all denjenigen bedanken, die mir bei der Erstellung meiner Bachelorarbeit geholfen haben.

Besonders bedanken möchte ich mich bei meiner Frau, die mir nicht nur als Lektorin beigestanden, sondern auch meine Launen ertragen und mir den Rücken frei gehalten hat.

Vielen Dank auch an Jedrzej Rybicki für die vielen Diskussionen, Informationen und Vorschläge, und an Michael Stini, den ich mehr als einmal mit Fragen zu JavaME löchern durfte.

Auch meiner Familie möchte ich für ihre ausdauernde Unterstützung danken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die Zahl der Kraftfahrzeuge steigt rapide an. Im Jahre 2007 gab es weltweit schätzungsweise 600 Millionen PKWs. In einer Studie des *Umwelt und Prognose-Instituts e.V.* [Umw] wird prognostiziert, dass die Anzahl der weltweiten PKWs bis zum Jahre 2030 auf ca. 2,3 Milliarden steigen wird. Diese Zahlen verdeutlichen eindrucksvoll, welchen Stellenwert Mobilität in der heutigen Gesellschaft einnimmt. Immer mehr Menschen sind auf ihre Kraftfahrzeuge angewiesen und wollen ohne Kartenstudium ihr Reiseziel erreichen. Der Markt für Navigationssysteme boomt.

Die erste Generation dieser Navigationssysteme leitet den Fahrer anhand ihres „eingebauten“ Kartenmaterials auf dem vermeintlich kürzesten oder schnellsten Weg zum gewünschten Zielort. Da diese Systeme aber über keinerlei Informationen über die aktuelle Verkehrslage verfügen, ist die von ihnen gefundene „schnellste“ Route oftmals suboptimal, da zum Beispiel Staus oder Baustellen nicht berücksichtigt werden.

Die Industrie entwickelte die Geräte weiter und stattete viele mit *TMC* (*Traffic Message Channel*) aus, um aktuelle Verkehrsinformationen zu erhalten. So können die Navigationssysteme auf Verkehrsbehinderungen, die sie per *TMC* gemeldet bekommen, reagieren und gegebenenfalls die Route ändern. Leider ist die Bandbreite von *TMC* sehr begrenzt und so können nur relativ wenige Meldungen pro Minute übertragen werden. Außerdem müssen die Behinderungen erst gemeldet und dann manuell an zentraler Stelle ins System eingebunden werden. Daher kommt es häufig vor, dass man trotz *TMC* in einem Stau steht oder dass man wegen *TMC* einen Stau umfährt, den es schon lange nicht mehr gibt.

Hier setzen nun proprietäre Weiterentwicklungen der Navigationssystemhersteller an, die „ihre“ Verkehrsinformationen nicht mehr nur durch Anrufe, Verkehrsüberwachung, Verkehrsleitsysteme oder Sensoren erhalten. Als weitere Quelle zur Analyse der aktuellen Verkehrslage nutzen sie anonymisierte Handybewegungen, die sie von ihren Kooperationspartnern, den Mobilfunkprovidern, erhalten. Die Übertragung dieser Verkehrsinformationen an die Navigationssysteme erfolgt in der Regel über Mobilfunk. Meist werden hierbei sowohl Gebühren für die Mobilfunknutzung als auch für die Nutzung der Verkehrsinformationen erhoben.

1.1 Motivation

Betrachtet man zum einen den rasanten Zuwachs an PKWs und zum anderen die immer länger werdenden Staunachrichten im Radio, so erkennt man, dass die Kenntnis der aktuellen Verkehrslage für Navigationssysteme immer wichtiger wird. So ist es nur verständlich, dass die Hersteller von Navigationssystemen einen weiteren Markt für sich entdeckt haben, die Vermarktung von aktuellen Verkehrsinformationen.

Diese Systeme haben jedoch einige Nachteile:

- Die meisten Systeme sind proprietär und daher nicht systemübergreifend beziehungsweise herstellerübergreifend nutzbar. Dadurch erhält der Nutzer bei weitem nicht alle Verkehrsinformationen, die zur Verfügung stehen.
- Die Mehrzahl der Systeme nutzt eine Client/Server Architektur und skaliert daher schlecht.
- Die auf Handybewegungsdaten basierenden Systeme nutzen jeweils nur die Daten eines Mobilfunkproviders – dies führt zu einer eingeschränkten „Sicht“ der Verkehrslage.
- Die Hersteller lassen sich den Mehrwert gut bezahlen (Abogebühren).

Im Rahmen dieser Arbeit werden wir zeigen, dass der Einsatz eines Peer-to-Peer Netzwerkes zum Verkehrsinformationsaustausch viele Vorteile für den Nutzer hat, und dass heutige Mobiltelefone und PDAs leistungsfähig genug sind ein solches Peer-to-Peer Netzwerk zusätzlich zu der Navigationsarbeit selbständig zu verwalten.

1.2 Aufbau der Arbeit

In Kapitel 2 dieser Arbeit werden zunächst die Funktionsweise des von uns verwendeten Peer-to-Peer Netzwerkes und dessen Erweiterungen dargelegt, sowie die Entscheidungen über die weiteren in dieser Arbeit verwendeten Grundlagen für die Implementierung des Demonstrators erläutert.

Kapitel 3 verdeutlicht die wichtigsten Aspekte der Implementierung der Programme und stellt ihre Funktionsweise vor.

In Kapitel 4 werden die Ergebnisse dieser Arbeit noch einmal zusammenfassend dargestellt.

Weitere Überlegungen und Ausblicke führt Kapitel 5 auf.

Kapitel 2

Design

Im Rahmen dieser Bachelorarbeit soll ein Demonstrator für ein Peer-to-Peer-basiertes Verkehrsinformationssystem programmiert werden. Dieser soll zeigen, ob ein auf *CAN* basierender Ansatz des Verkehrsinformationsaustausches mit aktuellen Mobiltelefonen und PDAs (im Weiteren zusammenfassend als „Mobilgeräte“ bezeichnet) möglich ist. Dabei soll untersucht werden, ob die heutigen Mobilgeräte leistungsfähig genug sind um neben der Navigation auch noch den anfallenden Datenverkehr und Verwaltungsoverhead des Peer-to-Peer Netzes zu bewältigen.

Um die gestellte Aufgabe zu lösen werden wir eine Navigationssystem-ähnliche Peer-to-Peer Anwendung für Mobilgeräte und einen Monitor, der auf einem PC die von allen Peers zusammengetragene und im Peer-to-Peer Netzwerk gespeicherte Verkehrslage auf einer Karte darstellt, implementieren.

Dafür müssen wir zunächst entscheiden, welche Programmiersprache wir zur Implementierung der beiden Programmteile nutzen wollen.

Danach ist zu klären, welches Kartenmaterial wir als Grundlage für die Navigation und den Verkehrsinformationsaustausch nutzen wollen. Es liegt nahe, dass alle Peers und auch der Monitor über das gleiche Kartenmaterial verfügen müssen um straßenbezogene Verkehrsinformationen effizient austauschen zu können.

Darüber hinaus müssen wir uns Gedanken über das zu verwendende Peer-to-Peer System machen.

2.1 Die Programmiersprachen JavaME und JavaSE

Zuerst musste entschieden werden, welche Programmiersprachen zur Umsetzung benutzt werden sollten. Da es für nahezu jede Programmiersprache auch Umsetzungen für PCs gibt, lag unser Hauptaugenmerk auf der Unterstützung der Mobilgeräte.

Aufgrund der weiten Verbreitung von *Java Micro Edition (JavaME)*-tauglichen Handys und PDAs entschieden wir uns für diese Programmiersprache. Fast jedes moderne Mobilgerät besitzt eine *KVM* (*Kilobyte Virtual Machine* - das *JavaME* Pendant zur *Java Virtual Machine*), die sich um die Hardwareansteuerung kümmert. So muss - zumindest in der Theorie - nicht für jedes Mobilgerät das Programm umgeschrieben beziehungsweise portiert werden. Außerdem bietet die Wahl von *JavaME* den Vorteil, dass zur Entwicklung der zum Projekt gehörenden PC Anwendungen (*OSM-Parser, JavaSE Peer, Monitor*, siehe Kapitel 3) große Teile des Sourcecodes direkt übernommen werden können, da auch *JavaSE* (*Java Platform Standard Edition*) bis auf wenige Ausnahmen *JavaME*-Code direkt nutzen kann. Darüber hinaus ist Java eine objektorientierte Programmiersprache, die typsicher ist und einen Garbage-Collector mitbringt. Dieser Garbage-Collector hat den Vorteil, dass wir uns nicht selbst um die Speicherallokation kümmern müssen. Damit ist eine Hauptfehlerquelle bei anderen Programmiersprachen (wie z.B. C) - falsche Speicherallokation - in Java gar nicht vorhanden.
Da die Entwicklungsumgebung, das *Wireless Toolkit* (*WTK* [wtk08]), zu *JavaME* einen Emulator bereitstellt, der ein virtuelles Handy zur Verfügung stellt, wurde vereinbart, dass der Demonstrator in der Emulation lauffähig sein muss.

2.1.1 JavaME

Bei *JavaME* handelt es sich um eine abgespeckte und auf geringen Speicherverbrauch und hohe Effizienz getrimmte Version der *JavaSE*. Ähnlich wie *JavaSE* hat sich auch *JavaME* über mehrere Generationen hinweg weiterentwickelt, und so gibt es heute zwei Unterkategorien, die die Fähigkeiten der KVM beschreiben:

CLDC (Connected Limited Device Configuration) Die CLDC definiert die kleinstmögliche Hardware Konfiguration, die von Java verwendet werden kann [Pfe07]. Mo-

mentan existieren die Versionen 1.0 und 1.1. Da aktuelle Mobilgeräte alle CLDC 1.1 unterstützen und CLDC 1.0 den entscheidenden Nachteil hat, dass es keine Gleitpunktzahlen unterstützt, verwendet dieses Projekt die CLDC 1.1.

MIDP (Mobile Information Device Profile) Das MIDP umfasst die Funktionen zur Ansteuerung und Abfrage der Tastaturen, der Bildschirme und des Speichers der Mobilgeräte [Pfe07]. Für dieses Projekt wurde die MIDP 2.0 als Mindestvoraussetzung gewählt.

2.1.2 Restriktionen der JavaME

Die Speicher- und Effizienzoptimierung führt im Vergleich zu J*avaSE* allerdings zu Einschränkungen. So stehen dem Programmierer viele Konstrukte, die neuere Versionen von *JavaSE* mitbringen, nicht zur Verfügung. In *JavaME* nicht verfügbar sind unter anderem:

- *Generics*
- Listen (weder *ArrayList*, noch *LinkedList*)
- *Maps*
- Serialisierung (kein *implements Serializable*)

Daher bleiben als schon in *JavaME* vorhandene Datenstrukturen zum Speichern von mehreren Werten nur *Arrays*, *Vectoren* und *Hashtables*. Um Daten über das Netzwerk schicken zu können müssen diese Daten erst serialisiert werden. Da *JavaME* keine Funktionalität dafür mitbringt, müssen wir alle Daten, die wir über das Netzwerk verschicken wollen, selbst serialisieren. Dazu erhält jedes zu serialisierende Objekt in der Implementation zwei Methoden, *fromDataInputStream()* und *toDataOutputStream()*, welche die wichtigen Objektvariablen über einen *DataInputStream* empfangen beziehungsweise über einen *DataOutputStream* verschicken.

In *JavaSE* würde in den meisten Fällen ein *implements Serializable* in der Klassendefinition reichen und wir könnten Objekte einfach über einen *ObjectOutputStream* (den

JavaME ebenfalls nicht kennt) verschicken bzw. über einen *ObjectInputStream* empfangen, wobei sich *JavaSE* selbständig um die Serialisierung kümmern würde.

2.2 OpenStreetMap (OSM)

Um eine Navigationslösung zu programmieren benötigt man natürlich Straßendaten. Da wir schon mit dem Peer-to-Peer Verkehrsinformationsaustausch einen Non-Profit-Weg einschlagen liegt es nahe, Kartendaten zu verwenden, die keine Lizenzkosten nach sich ziehen. Als Kandidat kristallisierte sich schnell *OpenStreetMap* [Ope08a] heraus. Bei *OSM* handelt es sich um eine freie Weltkarte, deren Aufbau jeder mitgestalten kann und darf. Da die Daten nur von *OpenStreetMap*-Mitgliedern mit Hilfe von GPS Loggern gesammelt werden, dürfen sie lizenzfrei verwendet werden. Auch wenn die Weltkarte noch längst nicht komplett ist und viele vor allem ländliche Gebiete noch fehlen, reicht die Datenfülle für unseren Demonstrator aus.

Die Daten erhält man entweder, indem man die Weltkarte komplett (*planet.osm* - Ende 2008 ca. 93GB als XML bzw. 4,3GB mit bzip2 komprimiertes XML) oder einen Teil davon direkt als XML File herunterlädt. Alternativ kann man auch einen Kartenausschnitt über das Webfrontend auswählen und dann als XML Datei exportieren [Ope08b]. Bei dem Export gibt es allerdings eine Größenbeschränkung um die Server nicht zu überlasten.

In *OSM* werden nicht nur Straßendaten gesammelt, sondern zum Beispiel auch Kartendaten zu Flüssen, Wäldern, Schienen, Orten von besonderem Interesse, Geschwindigkeitsbegrenzungen, Straßenbelag, Fahrrad- und Fußwegen. Da die Entwickler von *OSM* das Augenmerk nicht auf Navigation, sondern auf die Darstellbarkeit einer Weltkarte gelegt haben, ist die *OSM* Datenstruktur ungeeignet um mit ihr eine ressourcenschonende Navigationslösung zu erstellen. Daher haben wir einen Parser geschrieben (*OSMStAX-Parser*), der die *OSM*-Daten parsed und alle für den Demonstrator unnötigen Elemente entfernt.

2.3 Das Peer-to-Peer Netzwerk

Einen Überblick über verschiedene Peer-to-Peer Netzwerke bietet unter anderem [ATS04].

Nach der Auswertung der Ergebnisse der Arbeiten [RSK+07], [Koe08], [Mel07], [Kow07] und [Zok07], die verschiedene Peer-to-Peer Netzwerke verwenden, haben wir uns dazu entschieden, für diese Arbeit eine abgeänderte Version des in [Rat02] vorgestellten *Content-Addressable Network* (*CAN*) zu verwenden, da es sich hervorragend für unsere Zwecke modifizieren lässt und als einziges Peer-to-Peer Netzwerk eine 2-dimensionale Datenverteilung direkt unterstützt.

2.3.1 Content-Addressable Network (CAN)

CAN ist ein Peer-to-Peer Netzwerk, das eine Hashtabelle auf alle am System beteiligten Peers verteilt ablegt. Diese verteilte Hashtabelle (*Distributed Hashtable* (*DHT*)) speichert Schlüssel-und-Werte Paare (K_i, V_i) ab. Um diese Paare verteilt speichern zu können, spannt *CAN* ein virtuelles d-dimensionales kartesisches Koordinatensystem auf (üblicherweise ist $d \geq 2$).

CAN teilt diesen Raum dynamisch in Zonen auf (siehe Abbildung 2.1). Jeder Peer erhält seine eigene Zone und ist für die Verwaltung aller Schlüssel-Werte Paare zuständig, die die gleichförmige Hashfunktion angewandt auf den Schlüssel K auf Koordinaten P abbildet, die in seiner Zone liegen. Er speichert außerdem eine Liste mit seinen Nachbarn.

Zwei *CAN* Nodes sind genau dann Nachbarn, wenn sich ihr Koordinatenraum in $d - 1$ Dimensionen überschneidet und in einer Dimension aneinander angrenzt. Damit die „Randknoten“ auch Nachbarn in jede Richtung besitzen, wird der von *CAN* aufgespannte Raum als Torus betrachtet (siehe Abbildung 2.2) – verlässt man in einer Dimension den Wertebereich von *CAN* auf der einen Seite, so betritt man ihn wieder auf der anderen Seite (Abbildung 2.2) [1].

[1] Um die Darstellungen zu vereinfachen wird die Toruseigenschaft in einigen Abbildungen nicht dargestellt.

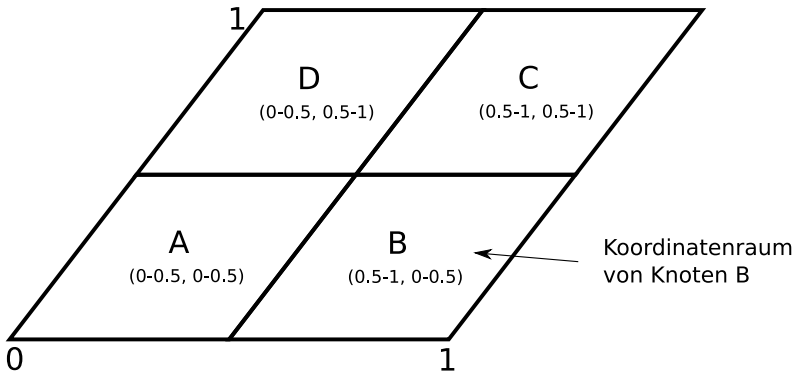

Abbildung 2.1: Beispiel eines zweidimensionalen $[0,1]x[0,1]$ *CAN* mit 4 Nodes

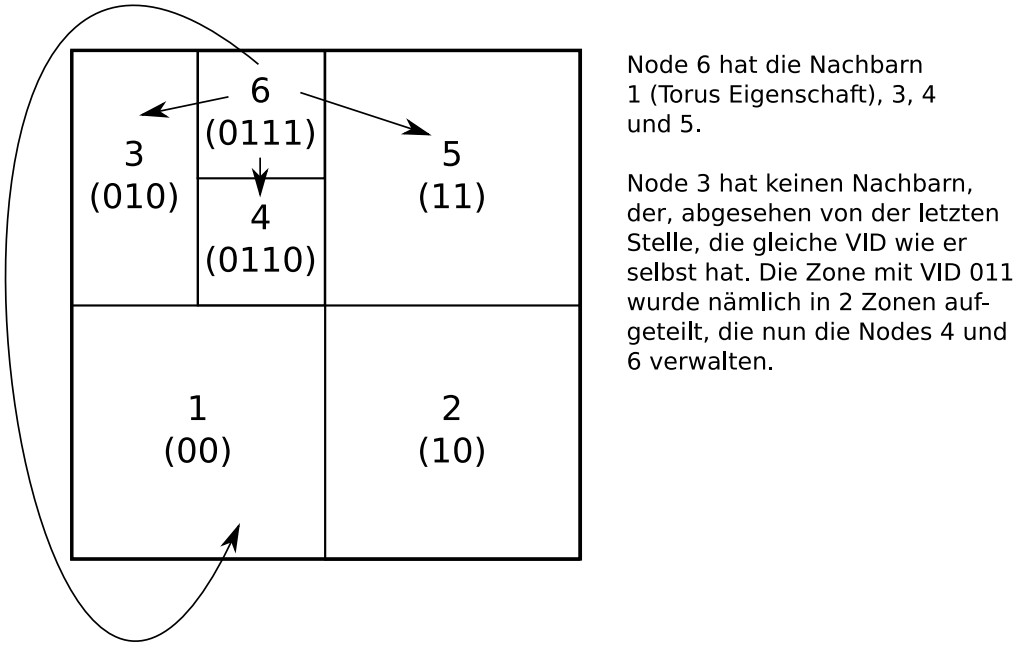

Abbildung 2.2: *CAN* Nachbarschaftsbeispiel

Will ein neuer Peer dem *CAN* beitreten, so wendet er sich an einen sich schon im *CAN* befindlichen Knoten. Dieser teilt seine Zone nun entlang einer Dimension in zwei Hälften und übergibt eine davon an den neuen Knoten (Abbildung 2.3). Die Dimension in welcher die Teilung vorgenommen wird ist nicht frei wählbar, sondern folgt einem festen Schema. Die erste Zone (die den gesamten „*CAN*-Raum“ umfasst) wird entlang der 1. Dimension geteilt. Eine daraus entstandene Zone wird dann entlang der 2. Dimension geteilt. Jede weitere Teilung erfolgt in der nächst höheren Dimension Modulo d.

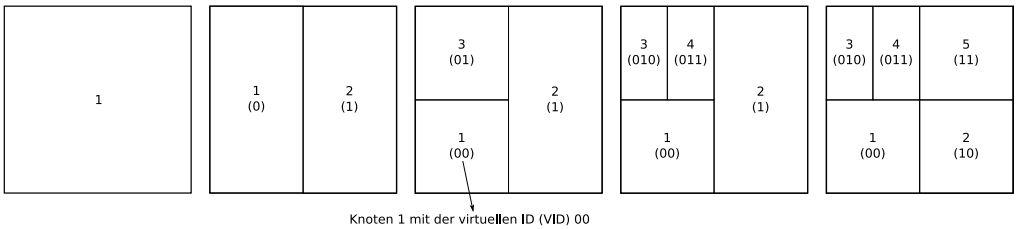

Abbildung 2.3: Beispiel einer Aufteilung von *CAN* nachdem nacheinander 5 Knoten beigetreten sind

Neben der Repräsentation einer Zone durch ihren Koordinatenraum gibt es die wesentlich effizientere Methode der Zuweisung einer eindeutigen *virtuellen ID* (*VID*). Man kann die Teilung der Zonen mit Hilfe eines binären Baumes veranschaulichen (Abbildung 2.4), wobei die Zonen die Blätter dieses Baumes sind. Wird eine Zone geteilt, so erhält der Knoten im Baum zwei Kinder - ihre zwei Teilzonen. Die Kanten von der Vaterzone zu den beiden Kindzonen beschriftet man derart, dass die Kante zu der Zone, die die „kleinere Hälfte“ bezüglich der Teilungsdimension erhalten hat, mit 0 beschriftet wird und die andere mit 1. Die VID einer Zone ist nun die Aneinanderreihung der Kantenbeschriftungen auf dem Weg von der Wurzel bis zu dem der Zone entsprechenden Blatt.

Verlässt ein Knoten das *CAN*-Netzwerk, so übergibt er seine Zone (und alle in ihr gespeicherten Daten) wenn möglich an denjenigen seiner Nachbarn, der mit Ausnahme der letzten Stelle die gleiche VID wie er selbst besitzt. Dieser kann dann beide Zonen wieder zu einer größeren verschmelzen. Sollte es diesen Nachbarn nicht geben (weil auch diese Zone aufgeteilt wurde, siehe Abbildung 2.2), so übergibt er die Zone an einen beliebigen anderen Knoten, der dann mehrere Zonen verwalten muss. Beim nächsten Join an seiner Adresse gibt dieser dann diese Zone wieder ab.

Will ein *CAN* Node auf einen Datensatz (K_i, V_i) zugreifen, so wendet er zunächst die Hashfunktion auf K_i an und erhält die Koordinaten P_i. Sollten diese Koordinaten nicht zu

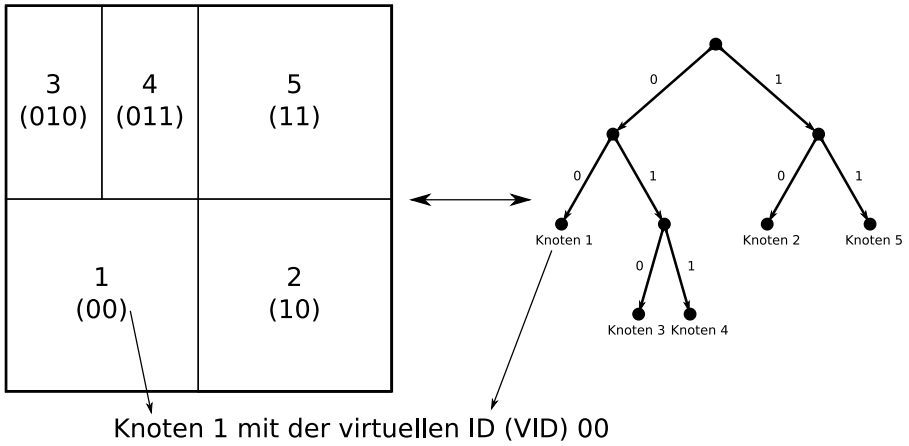

Abbildung 2.4: Gegenüberstellung von *CAN* Zonen und dem zugehörigen *VID*-Baum

seiner Zone gehören, so fragt er denjenigen seiner Nachbarn, dessen Zone den größten Fortschritt in Richtung der Zielkoordinaten verspricht. Fällt P_i nicht in die Zuständigkeit dieses Nachbarn, so sucht sich der Nachbar auf gleiche Weise aus seiner Nachbarschaftsliste wiederum den Nachbarn, der den größten Fortschritt verspricht und leitet die Anfrage an diesen weiter. Auf diese Weise wird verfahren bis die Nachricht den Verwalter von P_i erreicht. Der für die Daten zuständige *CAN* Knoten beantwortet die Anfrage dann auf direktem Wege. Somit braucht ein Peer also nur Informationen zu seinen direkten Nachbarn im virtuellen Raum um jeden Datensatz zu erreichen.

2.3.2 Veränderungen an CAN

Für ein Peer-to-Peer Netz, das dem Verwalten von Verkehrsinformationen dient, ist *CAN* eine gute Grundlage, in einigen Punkten lässt es sich allerdings für unseren Anwendungszweck verbessern.

2.3.2.1 Verzicht auf die gleichförmige Hashfunktion

Betrachten wir zunächst den Grund des geplanten Verkehrsinformationsaustausches - die Berechnung der aktuell schnellsten Route zu unserem Navigationsziel unter Berücksichtigung der aktuellen Verkehrslage. Diese Route ist örtlich zusammenhängend. Eine gleichförmige Hashfunktion, wie sie *CAN* verwendet, würde aber dafür sorgen, dass die

Verkehrsinformationen für die Route nahezu gleichmäßig im ganzen Netzwerk verteilt sind. Wir müssen also sehr viele *CAN* Knoten nach Verkehrsinformationen fragen um alle Informationen zu erhalten. Um den örtlichen Zusammenhang von Daten zu fördern verzichten wir daher auf eine gleichförmige Hashfunktion. Stattdessen legen wir unsere Straßenkarte über ein 2-dimensionales *CAN*. Jede Zone ist nun genau für die Daten zuständig, deren Längen- und Breitengrade sie umfasst. Dadurch liegen Verkehrsinformationen zu einer Route auf verhältnismäßig wenigen Knoten, die dabei häufig auch untereinander benachbart sind.

2.3.2.2 Aggregation von Anfragen und Antworten

Da die Verkehrsdaten zu einer Route durch den Verzicht auf die gleichförmige Hashfunktion auch auf einer Aneinanderreihung von *CAN*-Nachbarn liegen, liegt es nahe nicht jede Verkehrsinformation einzeln abzufragen, sondern die Anfragen zu bündeln. Wir schicken also wenn möglich nicht nur eine Anfrage an unseren Nachbarn, sondern direkt mehrere, für die er den größten Fortschritt in Richtung Ziel verspricht. Dieser Nachbar beantwortet dann alle Anfragen, die seine Zone betreffen (nach Möglichkeit ebenfalls in einer Nachricht) und leitet die restlichen Anfragen wieder aggregiert an seine Nachbarn weiter.

Durch diese Maßnahme lässt sich die Anzahl der Anfragen und Antworten, und damit der Overhead, den jedes Paket mitbringt, verkleinern.

2.3.2.3 Mehrere Datensätze pro Schlüssel

Da wir Verkehrsinformationen sammeln ist für uns nicht nur die letzte Information zu einer Kante im Routinggraphen interessant, sondern eine Aggregation aller Daten zu dieser Kante aus der nahen Vergangenheit (z.B. der letzten 30 Minuten). Daher speichern wir nicht nur einen Wert pro Koordinatenpaar, sondern wir sammeln dort die Verkehrsinformationen der letzten 30 Minuten.

Kapitel 3

Implementierung

Das Ziel dieser Bachelorarbeit ist die Erstellung eines Demonstrators für ein Peer-to-Peer-basiertes Verkehrsinformationssystem.

In Kapitel 2 wurde beschrieben, welche grundlegenden Techniken für die einzelnen Komponenten des Projektes benötigt werden. Nun geben wir einen Überblick über die wichtigsten Aspekte der Implementierung der zum Projekt gehörenden Programme.

Zuerst wird die Funktionsweise des *OSMStAXParsers* beschrieben, der die *OSM*-Daten in einen für uns nutzbaren Routinggraphen wandelt. Danach erklären wir, wie wir das Peer-to-Peer Problem des Bootstrappings lösen. Im Anschluss folgt die ausführliche Erläuterung der beiden Peerimplementierungen, dem *JavaMEPeer* und dem *JavaSEPeer*. Daran anschließend skizzieren wir die Funktionsweise des *JavaSEMonitors* und erklären abschließend die Programmfunktionen zur *Generierung zufälliger Verkehrsinformationen*.

Zusätzlich zu diesem Überblick befinden sich in den Quellcodes eine Fülle von Kommentaren, die jeden einzelnen Aspekt des Programmablaufes beschreiben.

3.1 Der OSMStAXParser

Das Ziel von *OpenStreetMap* ist die Erstellung einer freien Weltkarte – genauer die Sammlung von Daten, die es erlauben eine möglichst gute und vollständige Straßenkarte aus ihnen zu rendern. Die *OSM*-Datenstruktur ist also nicht für Routingzwecke optimiert.

Damit wir die Kartendaten von *OpenStreetMap* als Grundlage für unsere Navigationslösung nutzen können, müssen wir die Kartendaten also erst aus dem *OSM* eigenen Format in einen Routinggraphen überführen.

Die zwei für den Aufbau eines Routinggraphen grundlegenden Elemente in der Struktur des *OSM XML* Files sind *Node* und *Way*.

Zwei Beispiele für *OSM Node*-Elemente:

1. ```
   <node id="248095218" lat="51.1840087" lon="6.6652453"
    user="Ropino" visible="true"
    timestamp="2008-07-14T10:57:23+01:00">
     <tag k="created_by" v="JOSM"/>
     <tag k="ref" v="20"/>
     <tag k="highway" v="motorway_junction"/>
     <tag k="name" v="AK Neuss-West"/>
   </node>
   ```
2. ```
   <node id="247283908" lat="51.1836351" lon="6.664066"
    user="Corriere" visible="true"
    timestamp="2008-02-11T23:02:00+00:00"/>
   ```

Ein Beispiel für ein *OSM Way*-Element:

```
<way id="4394506" visible="true"
 timestamp="2008-07-21T10:59:21+01:00" user="Ropino">
  <nd ref="266220246"/>
  <nd ref="26854361"/>
  <nd ref="253869269"/>
```

```
  <nd ref="26854360"/>
  <nd ref="83763016"/>
  <nd ref="26854359"/>
  <tag k="oneway" v="yes"/>
  <tag k="lanes" v="3"/>
  <tag k="highway" v="motorway"/>
  <tag k="maxspeed" v="80"/>
  <tag k="name" v="Corneliusstrasse"/>
  <tag k="created_by" v="Potlatch alpha"/>
</way>
```

Die Bedeutungen der *OSM Node*-Tags lassen sich aus Tabelle 3.1 entnehmen, die des *OSM Way*-Elements aus Tabelle 3.2.

Tabelle 3.1: Bedeutung der für uns wichtigen Tags in OSM *Nodes*

TAG	BEDEUTUNG
id	eine eindeutige Node-ID im Wertebereich *long integer*
lat	Breitengrad auf dem der Node liegt
lon	Längengrad auf dem der Node liegt
user	OSM Benutzername desjenigen, der den Wegpunkt zur OSM Datenbank hinzugefügt hat
visible	Ist der Node sichtbar?
timestamp	Zeitpunkt der letzten Änderung an diesem Punkt

Tabelle 3.2: Bedeutung der für uns wichtigen Tags im OSM *Way*-Element

TAG	BEDEUTUNG
id	eine eindeutige Way-ID im Wertebereich *long integer*
nd ref	Referenz auf einen OSM *Node* über seine *Node-ID*
user	OSM Benutzername desjenigen, der den *Way* zur OSM Datenbank hinzugefügt hat
visible	Ist der *Way* sichtbar?
timestamp	Zeitpunkt der letzten Änderung an diesem Punkt
maxspeed	die Höchstgeschwindigkeit auf diesem *Way*
highway	der Straßentyp, zu dem dieser *Way* gehört
lanes	die Anzahl der Spuren der Straße
oneway	Ist der *Way* in beide Richtungen befahrbar?
name	der Straßenname des *Ways*

Ein *OSM Way* ist also eine Aneinanderreihung von *OSM Nodes*. Da das Ziel von *OpenStreetMap* das Rendern einer freien Weltkarte ist, wurde die *OSM*-Datenstruktur nicht für die Erstellung eines Routinggraphen entworfen. Dadurch befinden sich Straßenkreuzungen meist nicht am Anfang oder Ende eines *OSM Ways*, sondern in „inneren“ *Nodes*. Daher müssen wir für die Erstellung des Routinggraphen zunächst die Straßenkreuzungen in den *OSM*-Daten suchen und dabei oftmals einen *OSM Way* in mehrere Kanten für unseren Routinggraphen umwandeln (Abbildung 3.1).

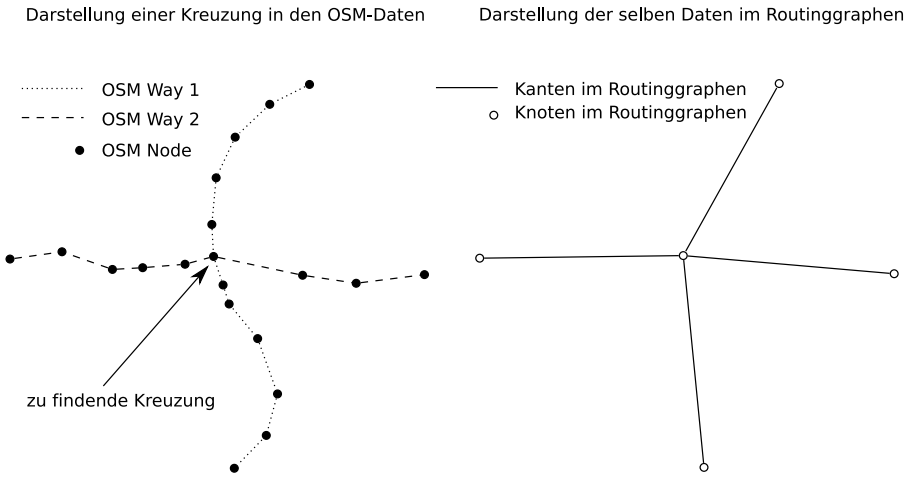

Abbildung 3.1: Finden von Kreuzungen in den *OSM*-Daten

3.1.1 Fehler in den OSM Kartendaten

Da die *OSM*-Daten nur auf das Rendern einer Karte ausgelegt sind, kann es vorkommen, dass eine Kreuzung (ein Punkt, von dem mindestens 3 Wege ausgehen) in *OSM* richtig gerendert wird, obwohl in Wirklichkeit die *OSM*-Daten fehlerhaft sind und die Erkennung der Kreuzung für uns unmöglich machen (Abbildung 3.2).

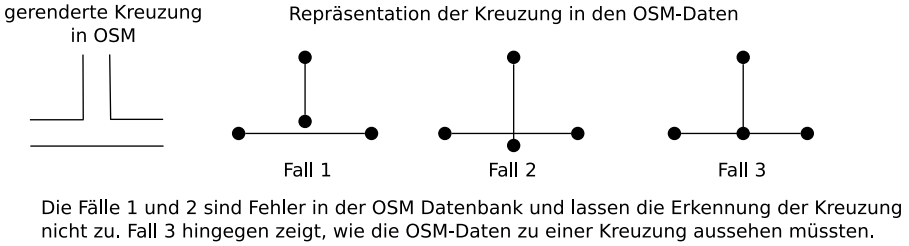

Abbildung 3.2: Fehler in den *OSM*-Daten

Wir haben uns dazu entschlossen, diese Fehler in den *OSM* Kartendaten zu ignorieren und in unseren Routinggraphen nur eindeutig erkennbare Kreuzungen zu übernehmen.

3.1.2 Berechnung der Durchfahrtszeiten für den Routinggraphen

Unsere Motivation für den Austausch von Verkehrsinformationen ist die Minimierung der Zeit die wir benötigen, um von unserem Navigationsstart zu unserem Navigationsziel zu gelangen. Daher ist es sinnvoll nicht die Höchst- beziehungsweise Durchschnittsgeschwindigkeit für jede Kante zu speichern, sondern die „Durchfahrtszeiten“. Dies minimiert vor allem den Rechenaufwand für den *Dijkstra*-Algorithmus (vergleiche Abschnitt 3.3.4), da wir hier die Durchfahrtszeiten der Kanten als Metrik verwenden.

Damit wir für den Fall, dass wir noch keine Verkehrsinformationen zu einer Kante besitzen, trotzdem eine plausible Durchfahrtszeit für sie nutzen können, speichern wir eine Durchfahrtszeit in jeder Kante unseres Routinggraphen. Da das *OSM XML*-File uns solche Daten nicht direkt zur Verfügung stellt, müssen wir sie aus den Größen „Länge der Kante“ (diese lässt sich aus den Koordinaten der *OSM Nodes* des *OSM Ways* berechnen) und „Höchstgeschwindigkeit“ mit Hilfe der Formel $\text{Zeit} = \frac{\text{Strecke}}{\text{Geschwindigkeit}}$ berechnen.

Leider ist auch die Angabe von Höchstgeschwindigkeiten für *OSM Ways* optional. Da wir diese Information aber benötigen, mussten wir uns überlegen wie wir die Höchstgeschwindigkeit vernünftig schätzen, wenn sie nicht in den *OSM*-Daten vorhanden ist. Unserer Meinung nach eignet sich dazu am besten die feine Aufschlüsselung der Straßenart (`<tag k="highway" v="motorway"/>`). Als Schätzwerte für die Höchstgeschwindigkeiten haben wir die in Tabelle 3.3 angegebenen Werte benutzt.

Tabelle 3.3: Bestimmung der Höchstgeschwindigkeit anhand der Straßenart

HIGHWAYTYP	ANGENOMMENE HÖCHSTGESCHWINDIGKEIT
motorway	130
motorway_link	80
motorway_junction	80
trunk	100
trunk_link	80
primary_link	70
primary	100
primary_trunk	70
secondary	70
secondary_link	60
tertiary	50
unclassified	50
unsurfaced	25
track	50
residential	30
living_street	10
service	10
bridleway	0
cycleway	5
footway	0
pedestrian	0
bus_guideway	5
steps	0

3.1.3 Java XML Parsing APIs

In *JavaSE* stehen grundsätzlich drei APIs zur Verfügung, die das XML-Parsing für uns vereinfachen können:

1. Das *Java Document Object Model (JDOM)*. Hierbei wird das gesamte *XML*-File eingelesen und in einer internen Datenstruktur zur weiteren Verarbeitung angeboten.

2. Die *Simple API for XML Parsing (SAX)*. SAX basiert auf einem Ereignismodell. Sie liest die *XML*-Datei wie einen Datenstrom ein und löst bei erkannten Elementen ein Ereignis aus. Da dieser Parser selbständig Ereignisse auslöst, auf die das aufrufende Programm reagieren muss, spricht man von einem Push-Parser.

3. Die *Streaming API for XML (StAX)*. *StAX* ist ein Pull-Parser, der ein *XML*-File sequentiell abarbeitet. Dabei bleibt es dem Programm überlassen, wann es die nächsten Daten abrufen möchte.

Weil die zu parsenden *OSM XML*-Files relativ groß sein können (das *OSM* planet *XML* mit den Daten für die gesamte Weltkarte umfasst aktuell 93GB), war schnell klar, dass das speicherhungrige *JDOM* nicht in Frage kam. Wir mussten uns also zwischen *SAX* und *StAX* entscheiden und haben mit *StAX* den pull-basierten Ansatz gewählt, da er leichter zu implementieren ist und *SAX* keine Vorteile für uns bringen würde. Da *StAX* erst ab *Java SE Development Kit 6 (JDK6)* verfügbar ist, braucht der *OSMStAXParser* auch zwingend die *Java SE Runtime Environment 6 (JRE6)*.

Der *OSMStAXParser* parsed also das *OSM XLM*-File und wandelt es in die von uns für den Routinggraphen gewählte Datenstruktur (siehe Paket *graph* in Abschnitt 3.3.1), und speichert sie in dem File routing.graph, das später im *JavaMEPeer*, *JavaSEPeer* und *JavaSEMonitor* verwendet wird.

Die Umwandlung des *OSM XLM*-Files in das File *routing.graph* bringt eine gewaltige Speicherplatzersparnis mit sich. So hat zum Beispiel das Neuss und Düsseldorf umfassende *XML*-File eine Größe von 6,7MB. Das daraus generierte routing.graph belegt hingegen nur 690KB Speicherplatz.

3.2 Der BootStrapServer (BSS)

Bei jedem Peer-to-Peer Netzwerk stellt sich die Frage, wie ein zukünftiger Teilnehmer (im Weiteren „Joiner“ genannt) Adressen von sich schon im Netzwerk befindlichen Peers findet, damit er dem Netzwerk beitreten kann. Die Lösung des Problems ist keineswegs trivial und es gibt verschiedenste Lösungsansätze.

Optimal wäre es natürlich, wenn das Programm eigenständig in der Lage wäre mindestens einen *CAN* Peer selbständig zu finden um dann über ihn dem *CAN* Netzwerk beitreten zu können. Da es sich bei unserem Projekt jedoch nur um eine Demonstration handelt, gibt es keine „stabilen“ Peers (Peers, die dafür bekannt sind, dass sie beinahe immer im Peer-to-Peer Netzwerk sind) mit festen IP-Adressen, die man dem Programm als Liste mitgeben könnte. Damit fällt ein Bootstrapping Ansatz, wie ihn zum Beispiel *Gnutella* nutzt, aus.

Daher haben wir uns dazu entschlossen, den Bootstrappingprozess zu vereinfachen und das Problem über einen *BootStrapServer* zu lösen. Dieser Server verhält sich ähnlich wie ein *DNS* Server. Ein Peer, der dem Netzwerk beitreten möchte, erfragt beim *BootStrapServer* eine Liste von sich im Netzwerk befindlichen Peers und deren IP-Adressen und für *CAN* benutzte UDP-Ports (je eine IP und ein UDP Port pro Peer). Der *BootStrapServer* antwortet mit einer Liste von maximal 5 zufällig ausgewählten Peers (er hält eine Liste von maximal 200 Peers vor um den Speicherbedarf zu begrenzen). Sollte der *BootStrapServer* keine Peers kennen, so teilt er das dem Joiner mit und dieser baut dann ein neues *CAN*-Netzwerk auf.

Zusätzlich zu der Liste übergibt der *BootStrapServer* im *BSSReply* auch noch die IP-Adresse und den UDP-Port des Joiners - diese müsste der Joiner zwar eigentlich kennen, aber wie sich bei der Implementation herausstellte schaffen es weder *JavaSE*, noch *JavaME* beim Erzeugen eines *DatagramSockets* bzw. einer *DatagramConnection* zuverlässig, die eigene (vom Internet aus erreichbare) IP korrekt zu ermitteln. Noch größere Probleme bereitet die Erkennung des eigenen UDP-Ports der *DatagramConnection* bei *JavaME*. Überlässt man *JavaME* die Wahl des Ports, so hat man keine Möglichkeit mit *JavaME*-Boardmitteln den UDP-Port zu identifizieren.

Damit der *BSS* immer weiß, welche Peers sich gerade im *CAN* Netzwerk aufhalten,

schickt ihm jeder Joiner nach erfolgreichem Join – und später periodisch – ein sogenanntes *BSSUpdate*, in dem er indirekt (über die Absendeadresse des Paketes) seine für *CAN* genutzte IP-Adresse und seinen UDP-Port mitteilt. Sollte der *BSS* mehrere dieser Updates nicht erhalten, so entfernt er den Peer aus seiner Liste. Damit Peers die das Netzwerk verlassen haben schneller entfernt werden können, wurde zusätzlich ein *BSS-Remove* Paket eingeführt, das den Server dazu veranlasst, den Peer umgehend aus seiner Liste zu löschen.

3.3 JavaMEPeer und JavaSEPeer

Durch die Wahl von *JavaME* und *JavaSE* als Programmiersprachen für die Implementierung der Peers (siehe Kapitel 2) und durch die Beschränkung auf Datenstrukturen, die in *JavaME* verfügbar sind, unterscheiden sich die beiden Peervarianten lediglich in zwei Aspekten:

- beim Umgang mit Netzwerkverbindungen
- bei der Grafik- und Debugausgabe

Daher haben wir uns entschieden, die Hauptfunktionalität der Peerimplementierungen (siehe Abbildung 3.3) in den folgenden vier Paketen (packages) zu kapseln:

1. dem Paket *graph*, das den Routinggraphen zur Verfügung stellt und die Verkehrsinformationen zu den einzelnen Kanten speichert,
2. dem Paket *can*, das sämtliche Verwaltungsfunktionen des *CAN*-Netzwerkes für den Peer übernimmt,
3. dem Paket *navigation*, das sowohl die Berechnung des schnellsten Weges, als auch die Simulation einer Bewegung des Peers beinhaltet, und
4. dem Paket *platform*, das als einziges jeweils in einer *JavaME* und einer *JavaSE* Variante vorliegt und die unterschiedliche Programmierweise der Netzwerkfunktionen der beiden Programmiersprachen für die anderen Pakete kapselt.

Durch diese Aufteilung müssen bei Änderungen an den drei erstgenannten Paketen keine Änderungen in zwei leicht unterschiedlichen Programmiersprachen durchgeführt und konsistent gehalten werden.

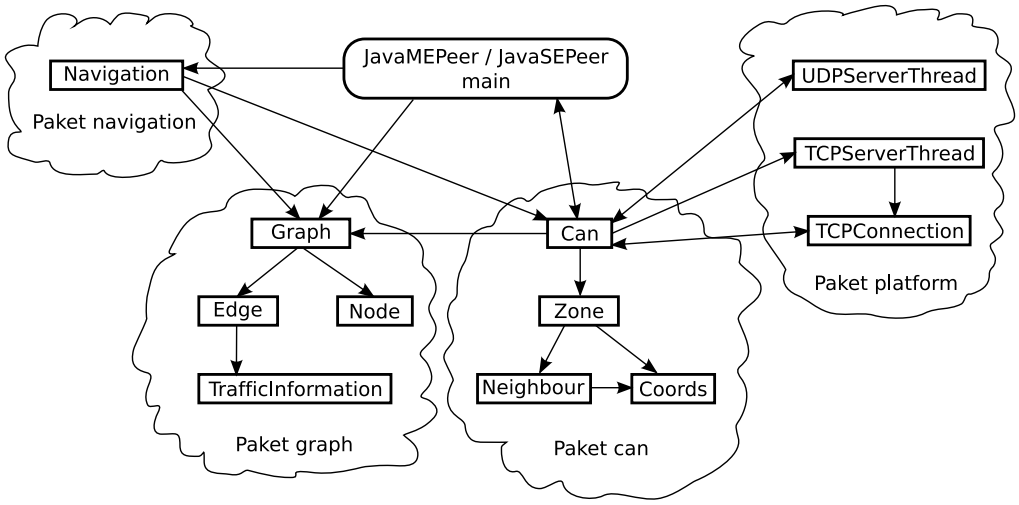

Abbildung 3.3: Vereinfachtes Klassendiagramm der Peer Implementierungen

Während der Programmierung und Tests auf *JavaME* stellten wir fest, dass moderne Mobilgeräte zwar häufig über 128MB oder mehr *RAM* verfügen, die *KVM* (siehe Kapitel 2) jedem Programm aber maximal 4MB (auf vielen Nokia Geräten sogar weniger als 2MB) zur Verfügung stellt. Außerdem stellte sich heraus, dass es zwar eine Erweiterung, das *FileConnection Optional Package (JSR 75)* [jsr] für *JavaME* gibt, die den Zugriff auf nichtflüchtigen Speicher (wie SD-Karten) der Mobilgeräte zulässt, die aber zwei gravierende Nachteile mit sich bringt:

1. Die Erweiterung ist optional und hardwareabhängig und auf den meisten Mobilgeräten nicht verfügbar.

2. Wenn die Erweiterung verfügbar ist, so fragt die *KVM* bei unsignierten Programmen bei jedem Zugriffsversuch den Nutzer um Erlaubnis für den Zugriff.

Bei der Implementierung haben wir daher auf das *JSR 75* verzichtet und stattdessen versucht, den Speicherbedarf möglichst gering zu halten. Außerdem haben wir darauf geachtet nur Objekte zu nutzen, die sowohl in *JavaME* als auch in *JavaSE* verfügbar sind.

3.3.1 Das Paket graph

Das Paket *graph* stellt sämtliche Datenstrukturen für unseren Routinggraphen bereit. Dies umfasst eine Knoten- und Kantenliste sowie die zu jeder Kante gehörenden Verkehrslageinformationen.

Entgegen dem üblichen Vorgehen haben wir den Graphen nicht pointer- beziehungsweise referenzbasiert implementiert, sondern die Knoten (Nodes) und Kanten (Edges) jeweils in einem Array gespeichert. Wir haben uns aus mehreren Gründen für diese relativ ungewöhnliche Methode entschieden:

1. Um Verkehrsinformationen eindeutig einer Kante zuzuordnen müssen alle Peers über den gleichen Graphen verfügen. Daher ist die Anzahl an Knoten und Kanten für jede Karte konstant, es wird also keine dynamische Datenstruktur benötigt.

2. Indem wir im *OSMStAXParser* ein eindeutiges Mapping von Node-IDs und Edge-IDs auf Indizes in den Knoten- bzw. Kantenlisten vornehmen, entfällt die Notwendigkeit diese IDs (beide *long integer*) zusätzlich in der Datenstruktur zu speichern. Wir können Kanten und Knoten nun eindeutig anhand ihres Arrayindexes zuordnen. Wir sparen also pro Kante und Knoten 8 Byte Speicherplatz.

3. Da wir zur Identifizierung einer Kante bzw. eines Knotens den Arrayindex benutzen ist ein Zugriff auf eine Kante in $O(1)$ ohne zusätzlichen Verwaltungsoverhead – wie zum Beispiel eine Hashtabelle – möglich.

Ein Knoten (Node) speichert seine Koordinaten (Längen- und Breitengrad) und ein Array von Kanten, die von ihm ausgehen. Die Kanten werden hierbei als Indizes des *Kantenarrays* des Graphen gespeichert.

Auch bei der Implementierung der Kanten wurde in Hinsicht auf Speichereffizienz und Laufzeitoptimierung des *Dijkstra*-Algorithmus (siehe Abschnitt 3.3.4) optimiert. So speichern wir keine Informationen darüber, in welchem Knoten eine Kante startet, sondern nur in welchem sie endet. Des Weiteren kann eine Kante ein *TrafficInformation*-Objekt mit Verkehrsinformationen speichern.

3.3.1.1 Die Klasse TrafficInformation (Abstract)

Von dieser abstrakten Klasse erben zwei Unterklassen, *TrafficInformationSmall* und *TrafficInformationLarge*. Da unser Anliegen die Minimierung der Fahrzeiten ist liegt es nahe, nicht die Durchschnittsgeschwindigkeiten für eine Kante zu erfassen, sondern die Durchschnittszeiten, die zum „Durchfahren“ der Kante benötigt werden.

Der Peer, der eine Kante im *Can*-Netzwerk verwaltet, hält in der Regel mehr Informationen (*TrafficInformationLarge*) vor, als ein Peer, der sich nur Informationen zu dieser Strecke aus dem *Can*-Netzwerk geholt hat (*TrafficInformationSmall*).

3.3.2 Das Paket can

Dieses Paket nutzt das Paket *graph* (siehe Abschnitt 3.3.1) um mit seiner Hilfe Navigationsdaten in einem *CAN*-ähnlichen Netzwerk zu verteilen beziehungsweise wiederzufinden. Es ist für die Verwaltung der Zonen und Nachbarn eines Peers ebenso zuständig wie für die Behandlung von eingehenden Paketen aus dem Netzwerk. Eine Zone enthält dabei die eigene virtuelle ID (*VID*) der *CAN*-Zone, eine Liste aller Nachbarn der Zone und eine Liste aller zu der Zone gehörenden Kanten. Eine Kante gehört genau dann zu einer Zone, wenn die Koordinaten des Zielknotens der Kante in den Koordinatenraum der Zone fallen.

Des Weiteren steuert das Paket das Beitreten (*JoinRequest*) und Verlassen (*LeaveRequest*) des Peer-to-Peer Netzwerkes, beantwortet *TrafficInformationRequests* und leitet gegebenenfalls Nachrichten im Netzwerk weiter. Außerdem verschickt es periodisch *NeighbourUpdates* und *BSSUpdates* und reagiert auf selbst empfangene *NeighbourUpdates* von anderen Peers.

Im Gegensatz zu UDP, das Pakete verbindungslos verschickt und sich nicht darum kümmert, ob ein Paket die Zieladresse auch wirklich erreicht, ist TCP verbindungsorientiert. Bevor Nutzdaten verschickt werden können, muss erst eine Verbindung aufgebaut werden (*TCP Handshake*). Außerdem muss jede per TCP verschickte Nachricht vom Kommunikationspartner bestätigt werden. Als Bonus stellt TCP aber sicher, dass Pakete in der Absendereihenfolge beim Empfänger ankommen. TCP verursacht also relativ hohe Kos-

ten in Form von Overhead, wohingegen UDP keine Sicherheit vor verlorenen Paketen bietet.

Immer dann, wenn Paketverlust kein großes Problem darstellt, findet die Kommunikation im *CAN*-Netzwerk daher per UDP statt. Nur in wenigen Ausnahmen (*Join* und *Leave*) wird auf TCP zurückgegriffen, um ein geregeltes Übergeben der Zonen an den Nachbarn sicherzustellen.

Tabelle 3.4 erläutert die verwendeten TCP Nachrichten, Tabelle 3.5 gibt einen Überblick über die verwendeten UDP Pakete.

Tabelle 3.4: Verwendete TCP Nachrichten

PAKETNAME	BEDEUTUNG
JoinRequest	Diese Nachricht baut eine TCP Verbindung von einem Joiner zu einem aktiven *CAN*-Peer auf, in der der kontaktierte Peer entweder eine seiner Zonen abgibt (falls er mehrere verwaltet) oder seine Zone splitted und eines der Splitteile abgibt. Tritt bei dieser Kommunikation kein Fehler auf, so wurde die Zone an den Joiner übergeben, der nun ein weiterer *CAN*-Peer ist. Ein TCP *JoinRequest* wird nicht abgelehnt.
LeaveRequest	Diese Nachricht baut eine TCP Verbindung von einem Peer, der das *CAN*-Netz verlassen möchte, zu einem anderen *CAN*-Peer auf. Über diese Verbindung wird dann die Zone des verlassenden Peers an den weiter aktiv bleibenden *CAN*-Peer übergeben.

Damit nicht sämtliche Funktionen des *CAN*-Netzwerkes jeweils in einer *JavaME* und *JavaSE* Version vorliegen müssen, wurden alle diesbezüglichen Eigenheiten der Programmiersprachen in Wrapperobjekten in dem Paket *platform* gekapselt (siehe Abschnitt 3.3.3).

Tabelle 3.5: Verwendete UDP Nachrichten

PAKETNAME	BEDEUTUNG
BSSRequest	fordert beim *BootStrapServer* eine Liste von aktiven *CAN*-Peers an
BSSReply	enthält eine Liste mit maximal 5 aktiven *CAN*-Peers, die vom *BSS* verschickt wird
BSSRootReply	Hiermit antwortet der *BSS*, wenn er keine aktiven *CAN*-Peers kennt und fordert den Peer dadurch dazu auf ein neues *CAN*-Netz aufzubauen.
BSSUpdate	teilt dem *BootStrapServer* mit, dass dieser Peer im *CAN*-Netz aktiv ist
JoinRequest	In dieser Nachricht verschickt ein Peer die Koordinaten, an denen er dem Netzwerk beitreten möchte an einen aktiven Peer und wartet danach auf ein *JoinReply*.
JoinRequestForwarded	Immer wenn ein aktiver Peer einen *JoinRequest* an einen anderen aktiven Peer weiterleitet, verschickt er an den Joiner dieses Paket, damit bei diesem kein Timeout für den Joinversuch auftritt.
JoinReply	ist eine Antwort auf einen JoinRequest und teilt dem Joiner mit, unter welchem TCP-Port er einen Join per TCP machen darf.
NeighbourUpdate	enthält die VID, VIDlength und den TCP-Port eines Nachbarn (IP und UDP-Port lassen sich aus den Absendedaten des Paketes ermitteln)
TrafficInformationRequest	hiermit fragt ein Peer einen anderen nach Verkehrsinformationen zu einer Liste von Kanten
TrafficInformationReply	Die Antwort auf einen TrafficInformation-Request enthält eine Liste mit Kanten und zu ihnen gehörigen Verkehrsinformationen (als serialisiertes *TrafficInformationSmall* Objekt).
TrafficUpdate	hiermit teilt ein Peer dem Verwalter einer Kante mit, wie lange er benötigt hat um diese Kante zu passieren

3.3.2.1 Ablauf des Beitritts eines neuen Peers (*GeoJoin*)

In der Standard *CAN*-Implementierung [Rat02] stellt ein beitretender Peer einen Joinrequest an einen beliebigen sich schon im *CAN*-Netz befindlichen *CAN-Node*, welcher daraufhin seine Zone teilt und eine Hälfte an den Joiner übergibt. Da hierdurch relativ schnell an einigen Stellen im Netz sehr kleine Zonen entstehen können, während an anderen noch sehr große Zonen vorhanden sind, kann es hier zu Fairnessproblemen bezüglich der gleichmäßigen Belastung der Peers des Netzes kommen. Um dieses Problem zu mindern, wurden diverse Vorschläge unterbreitet um ein *Loadbalancing* beim Joinvorgang vorzunehmen.

Wir haben uns ein eigenes, *GeoJoin* genanntes, Verfahren überlegt, das folgendermaßen funktioniert (siehe auch Abbildung 3.4):

1. Der joinende Peer sendet einen *UDP BSSRequest* an den *BootStrapServer*.
2. Dieser antwortet mit einem *UDP BSSReply*, das eine Liste mit Adressen von maximal 5 aktiven Peers enthält.
3. Der beitretende Peer sendet nun einen *UDP JoinRequest* mit seiner IP, seinem UDP-Port und den gewünschten Join-Koordinaten (der Startposition seiner Navigation) an einen der Peers aus der *BSSReply* Liste.
4. Der auf diese Weise kontaktierte Peer nimmt den *UDP JoinRequest* nun mit einer bestimmten Wahrscheinlichkeit an und sendet ein *UDP JoinReply*, oder er leitet ihn an denjenigen seiner Nachbarn weiter, dessen Zone den größten Fortschritt in Richtung der gewünschten Join-Koordinaten verspricht. Im Falle der Weiterleitung schickt der Weiterleitende außerdem ein *UDP JoinRequestForwarded*-Paket an den Joiner, damit dieser seinen Timer für JoinRequestTimeouts zurücksetzen kann.

   Die Wahrscheinlichkeit der Annahme eines *JoinRequests* hängt zum einen von der Anzahl der gerade vom befragten Peer verwalteten Zonen ab (je mehr Zonen, desto höher die Wahrscheinlichkeit), zum anderen steigt sie mit Abnahme der Distanz zu den gewünschten Join-Koordinaten. Ein Join wird grundsätzlich immer angenommen, wenn der Verwalter der Join-Koordinaten erreicht wird (damit ist auch sichergestellt, dass das Verfahren terminiert).

Für den Fall, dass wir mit keiner unserer Zonen die GeoJoin-Koordinaten verwalten, haben wir folgende Formel für die Joinwahrscheinlichkeit *P* benutzt:

$$P = 1.0 - \frac{2 \cdot \text{Minimalabstand der Zentren unserer Zonen zu den Koordinaten}}{\text{Anzahl der von uns verwalteten Zonen} \cdot \text{Kartendiagonalenlänge}}$$

5. Erhält der beitretende Peer ein *UDP JoinReply*-Paket, welches den TCP Port des den *JoinRequest* akzeptierenden *CAN*-Nodes enthält, so öffnet er eine TCP Verbindung zu ihm und signalisiert ihm einen *TCP JoinRequest*.
6. Diesen beantwortet der aktive Peer durch Übertragung einer seiner Zonen. Besitzt er nur eine, so führt er einen Split durch und überträgt eine der beiden neu entstandenen Zonen. Bei der Übertragung werden die Liste der zur Zone gehörenden Kanten, die zu den Kanten gehörenden Verkehrsinformationen und die Nachbarschaftsliste der Zone übermittelt. Wird die Übertragung fehlerfrei beendet, so gilt die Zone als übergeben.
7. Nun informieren sowohl der Joinende, als auch derjenige Peer, der eine Zone abgegeben hat, ihre Nachbarn über die Änderungen in ihrer Nachbarschaft per *UDP NeighbourUpdate*-Paket. (nicht in Abbildung 3.4 enthalten)
8. Zum Schluss der Joinprozedur meldet sich der neue aktive Peer noch per *BSSUpdate* beim *BootStrapServer* an.

Wir erhoffen uns durch dieses Verfahren, dass der beitretende Peer eine Zone zur Verwaltung erhält, die in der Nähe seines Navigationsstarts liegt. Damit verringert sich die Anzahl der *Hops* für das Routing im *Can*-Netzwerk für die Beschaffung der *TrafficInformationReplys*, die zur Berechnung des kürzesten Navigationsweges nötig sind. Sollten wir dieses Ziel nicht erreichen, so haben wir sehr wahrscheinlich wenigstens einem Peer, der mehrere Zonen verwaltet hat, eine dieser Zonen „abgenommen“ und so eine Art *Loadbalancing* erreicht.

3.3.2.2 Ablauf einer Verkehrsinformationsabfrage oder eines Verkehrsinformationsupdates

Der Grund für die Verwendung des Peer-to-Peer Netzes ist die gemeinsame Verwaltung der Verkehrsinformationen. Da das Versenden eines *TrafficUpdate*-Paketes auf die gleiche Weise durch das *CAN*-Netz geleitet wird wie die Abfrage von Verkehrsdaten (ein

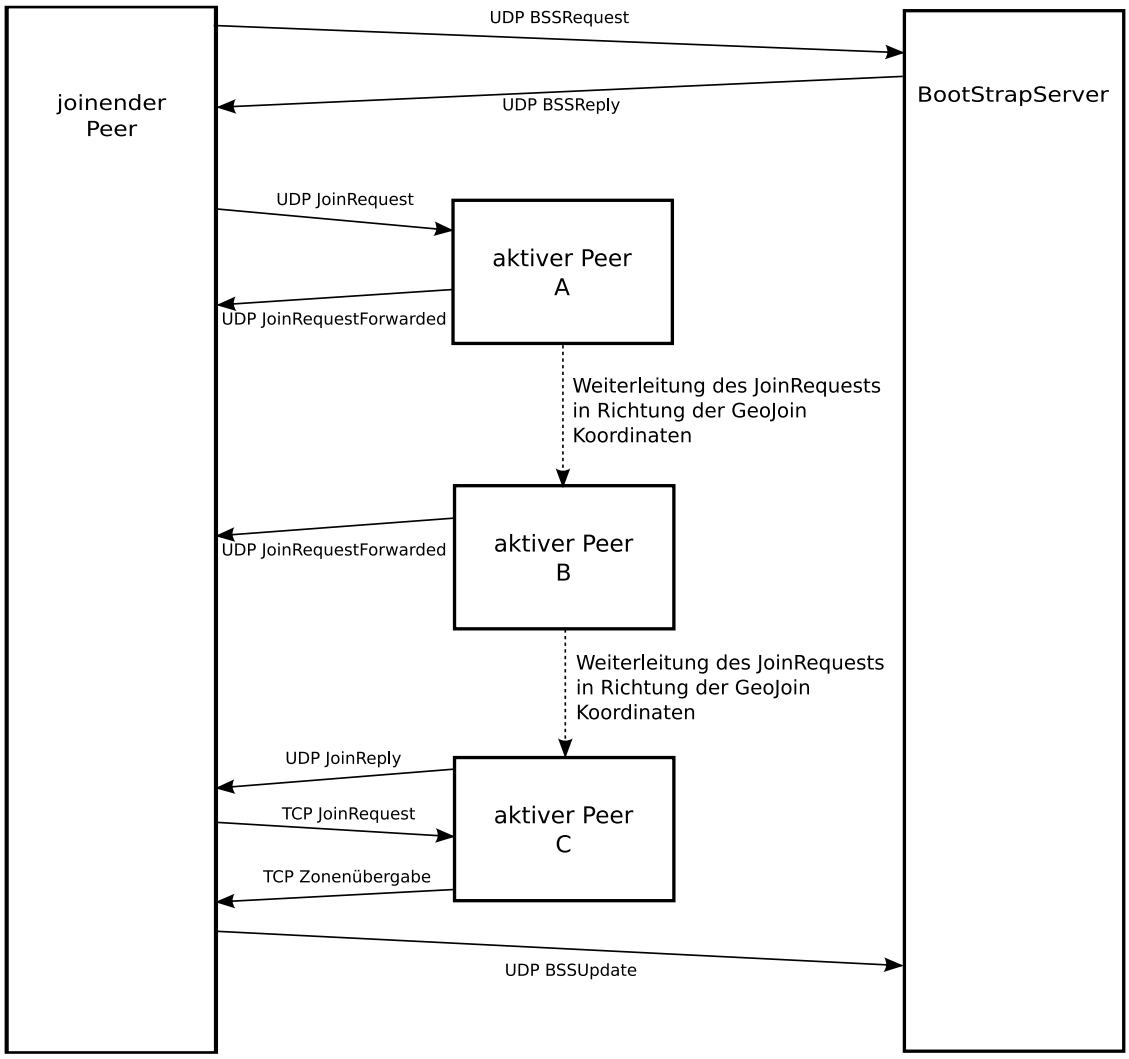

Abbildung 3.4: Beispielhafter Ablauf des Beitritts eines neuen Peers

TrafficInformationRequest), skizzieren wir hier nur kurz den Ablauf einer Verkehrsinformationsabfrage. Auf ein *TrafficInformationRequest* wird – anders als beim *TrafficUpdate* – zusätzlich auch über eine direkte Verbindung und ohne Routing durch das *CAN*-Netz geantwortet.

1. Der Peer, der Verkehrsinformationen zu einer Kante haben möchte, generiert ein *TrafficInformationRequest* Paket und leitet es an denjenigen seiner Nachbarn weiter, der den größten Fortschritt in Richtung Zielkoordinaten verspricht.

2. Erhält ein Peer einen *TrafficInformationRequest*, so prüft er, ob er Verwalter für diese Kante ist. Ist er es nicht, so leitet er das Paket an den Nachbarn mit dem größ-

ten Fortschritt weiter. Ist er der Verwalter, so entnimmt er dem Paket die Adresse des Anfragestellers und schickt ihm auf direktem Wege ein *TrafficInformationReply* bezüglich der angefragten Kante zurück.

Da wir, wie in Abschnitt 2.3.2.2 beschrieben, Nachrichtenaggregation betreiben, können natürlich Informationen zu mehreren Kanten mit einem *TrafficInformationRequest* abgefragt werden. Dabei muss dann in Punkt 2 jeder Peer, der ein solches Paket empfängt, das Paket gegebenenfalls aufsplitten und

1. für Kanten, für die er Verwalter ist, direkt eine Antwort schicken, und
2. für alle anderen Kanten jeweils den besten Nachbarn finden und die Teilanfrage(n) an seine Nachbarn weiterleiten.

3.3.3 Das Paket platform

Das Paket *platform* existiert sowohl in einer *JavaME* als auch in einer *JavaSE* Version. Es kapselt die Unterschiede der Netzwerkprogrammierung für die beiden Programmiersprachen. Dadurch wird das Paket *can* unabhängig von der verwendeten Java-Version nutzbar.

Um dies zu erreichen haben wir für beide Programmiersprachen jeweils die drei folgenden Klassen implementiert:

TCPConnection Dieses Objekt stellt sowohl in der *ME*, als auch in der *SE* Version alle nötigen Daten (*DataInputStream* und *DataOutputStream*) für die TCP Kommunikation bereit. Es bietet außerdem Methoden um eine TCP Verbindung aufzubauen bzw. sie zu beenden.

TCPServerThread Dieser Thread öffnet einen ServerSocket und wartet auf TCP Verbindungsversuche. Sollte ein solcher Versuch stattfinden, so wird die Verbindung als *TCPConnection*-Objekt an die Funktion Can.handleTCP übergeben, die dann das eigentliche (nun von der *KVM*/*JVM* abstrahierte) TCP Handling übernimmt.

UDPServerThread Dieser Thread wartet auf eingehende UDP Pakete (entweder auf einem fest vorgegebenen Port oder auf einem von der Laufzeitumgebung gewählten) und gibt diese an die Funktion Can.handleUDP weiter. Außerdem dient die Funktion sendUDPPacket dieser Klasse zum Verschicken aller *CAN* UDP-Pakete.

3.3.4 Das Paket navigation

Das Navigationspaket ist wieder so implementiert, dass eine gemeinsame Version für den *JavaMEPeer* und den *JavaSEPeer* existiert. Es kümmert sich um alle Belange der Navigation. Damit es andere Funktionen des Programms nicht lahm legen kann, läuft es als eigenständiger Thread. Diesem übergibt man beim Start einen Startknoten und einen Zielknoten aus dem Routinggraphen. Daraufhin sucht der Navigationsthread selbständig mit Hilfe des *Dijkstra*-Algorithmus den - nach aktuellem Informationsstand - zeitlich schnellsten Weg vom Startknoten zum Zielknoten. Sind Verkehrsinformationen zu Kanten verfügbar, so werden diese zur Berechnung herangezogen, wenn nicht, wird mit den vom Parser aus den *OSM*-Daten generierten Informationen gerechnet.

Enthält der gefundene schnellste Weg (repräsentiert durch ein Kantenindex-Array) Kanten, für die der Peer noch keine Verkehrsinformationsdaten besitzt, so werden diese angefordert. Damit für Kanten, zu denen im *CAN*-Netz keine Verkehrsinformationen existieren, nicht übermäßig viele Anfragen generiert werden, merkt sich das Navigationspaket in einer Hashtabelle zu jeder Kante, für die eine Nachfrage stattgefunden hat, wann die letzte Anfrage gestellt wurde. Mit Hilfe dieser Hashtabelle kann sichergestellt werden, dass *CAN*-Anfragen zu einer Kante mindestens 1 Minute auseinander liegen.

Damit auf aktuelle Verkehrsänderungen reagiert werden kann, überprüfen wir alle 10 Sekunden, ob sich die prognostizierte Fahrzeit für die Reststrecke unseres schnellsten Weges deutlich verschlechtert hat. Wenn dies der Fall ist, starten wir eine erneute *Dijkstra*-Suche.

Bei Tests im *JavaME*-Emulator (und auf einem PDA) zeigte sich, dass eine *Dijkstra*-Suche relativ viel Zeit kostet. Bei der zum Test benutzten Neuss und Düsseldorf umfassenden Karte, konnte die Berechnung des schnellsten Weges auf dem Testrechner knapp unter einer Sekunde dauern. Auf dem PDA brauchte die gleiche Berechnung ca. 25 Prozent weniger Zeit. Da diese Messungen ohne im Hintergrund laufende Ausgaben und

weitere Threads oder gar Netzwerkverbindungen erfolgten, haben wir uns bemüht, durch kleine Veränderungen an der Implementation die Geschwindigkeit zu erhöhen. Es zeigte sich, dass zwei relativ kleine Eingriffe in den Code einen Geschwindigkeitszuwachs von ca. 35 Prozent bringen:

1. Auf Variablendeklarationen innerhalb von Schleifen wird verzichtet, stattdessen werden die Variablen unmittelbar vor der Schleife deklariert, die Zuweisung der Werte kann dann weiterhin an der gleichen Stelle innerhalb der Schleife erfolgen. Ein Zeitgewinn von ca. 15 Prozent.

   Hintergrund: Eine Variablendeklaration kostet relativ viel Zeit (Reservierung von Speicherplatz auf dem *Stack*), eine Zuweisung eines neuen Wertes an eine schon existierende Variable ist dagegen vergleichsweise schnell.

2. Entgegen dem normalen Programmierstil, durch den Datenstrukturen durch *Getter*- und *Setter*-Methoden von anderen Teilen des Programms gekapselt werden, haben wir dem Navigationsthread direkten Zugriff auf die Kanten- und Knotenlisten der Klasse *graph* gegeben und fragen nicht jede einzelne Kante und jeden einzelnen Knoten den wir gerade benötigen über *getEdge* oder *getNode* der Klasse *graph* ab. Dadurch verlieren wir zwar die Abstraktion von der benutzten Datenstruktur, und gravierende Änderungen bezüglich der Repräsentation der *Kantenmenge* und *Knotenmenge* des Routinggraphen müssen nun auch im Navigationsthread vorgenommen werden, wir sparen jedoch ca. 20% der Zeit, die ein *Dijkstra*-Durchlauf benötigt, ein.

   Hintergrund: Jeder Aufruf von Funktionen anderer Klassen kostet viel Zeit, weil jedes Mal der *Kontext* der anderen Klasse hergestellt werden muss – und das in unserem Fall jeweils nur um einen direkt zu ermittelnden Rückgabewert zu liefern. Ein Verzicht auf den massenhaften Gebrauch von *Getter*- und *Setter*-Methoden führt also zu einer deutlichen Zeitersparnis.

3.3.5 Die grafische Ausgabe

Bei der Ausgabe der Kartendaten und Darstellung der gerade im Peer verfügbaren Verkehrsinformationen haben wir uns darauf beschränkt, eine Navigationssystem-ähnliche Anzeige der Straßenkarte darzustellen. Dabei geht es nicht darum, möglichst schön ge-

renderte Bilder zu zeigen, sondern darum die aktuelle Verkehrslage, die der Peer kennt, auf einer Karte darzustellen.

Dazu färben wir die Kanten wie folgt ein:

1. Eine Kante, über die wir keine Verkehrsinformationen haben zeichnen wir schwarz.
2. Eine Kante, für die die Verkehrsinformationen ähnliche Werte voraussagen wie die *OSM*-Daten zeichnen wir grün.
3. Eine Kante, für die die aktuellen Verkehrsinformationen gravierend schlechter (mind. 20% „langsamer“) sind als die *OSM*-Daten zeichnen wir rot.
4. Die Kanten der aktuell schnellsten Route markieren wir zusätzlich blau. Damit die Informationen aus 1. - 3. nicht verloren gehen, zeichnen wir den blauen Pfad leicht seitlich versetzt ein.

Die Abbildungen 3.5 bis 3.8 zeigen einige Screenshots des im Emulator laufenden *JavaMEPeers* in unterschiedlichen Situationen.

3.4 Der JavaSEMonitor

Der *JavaSEMonitor* stellt einen wichtigen Teil der Bachelorarbeit dar, da er die aktuell im Peer-to-Peer Netz verfügbaren Verkehrsinformationen visualisiert – er ist also quasi der Demonstrator.

Für den Betrieb des *CAN*-Netzes zum Verkehrsinformationsaustausch selbst ist er nicht nötig. Er wurde einzig zu Demonstrationszwecken entwickelt.

Anders als der *JavaSEPeer* und der *JavaMEPeer* soll er nicht nur die begrenzten Verkehrsinformationen, die ein Peer gerade zum Finden der für ihn schnellsten Route zum Ziel vorhält, darstellen, sondern die Verkehrslage nach aktueller Datenlage im Peer-to-Peer Netz visualisieren. Um an die aktuellen Verkehrsinformationen des gesamten *CAN*-Netzes zu kommen gibt es zwei Möglichkeiten:

1. Der *Monitor* fragt periodisch die Verkehrsinformationen zu allen im Netz befindlichen Kanten ab.
2. Die Peers schicken *TrafficUpdates* nicht nur an den für die Kante zuständigen Peer, sondern auch an den Monitor. Dieser behandelt die *TrafficUpdates* intern als wäre er Verwalter des gesamten *CAN*-Netzes.

Die erste Methode hat den Vorteil, dass kein Eingriff in die Peers nötig ist, hat aber gerade bei einer Demonstration mit einer geringen Anzahl von Peers (und damit der großen Anzahl an zu verwaltenden Kanten pro Peer) den Nachteil, dass das Abfragen aller Verkehrsinformationen, die ein Peer speichert, eine relativ hohe Last für den Peer erzeugt. Hingegen erhöht sich die Netzwerklast bei wenigen Peers die *TrafficUpdates* erzeugen nur unwesentlich durch das zweite Verfahren. Und auch der Eingriff in den Quellcode der Peers ist minimal, wir müssen nur jedes *TrafficUpdate*-Paket (welches wir immer dann erzeugen, wenn ein Peer eine Kante durchfahren hat) an zwei Adressen verschicken. Nach Abwägung der Vor- und Nachteile haben wir uns für das zweite Verfahren entschieden.

Die Ausgabe, die der *Monitor* erzeugt, ist der der beiden Peerimplementierungen sehr ähnlich. Auch er stellt den Verkehrszustand einer Kante durch unterschiedliche Einfärbung dar, zeigt dabei aber stets die gesamte Karte.

Abbildung 3.5: Screenshot des Statusbildschirms des *JavaMEPeers* bei laufender Navigation und Verbindung zum *CAN*-Netzwerk

Abbildung 3.6: Screenshot des kontextsensitiven Menüs des *JavaMEPeers*

Abbildung 3.7: Screenshot des Navigationsbildschirms des *JavaMEPeers* während eine neue schnellste Route berechnet wird

Abbildung 3.8: Screenshot des Navigationsbildschirms des *JavaMEPeers* während einer laufenden Navigation mit verfügbaren Verkehrsinformationen für einige Kanten

3.5 Generierung zufälliger Verkehrsinformationen

Zu Demonstrationszwecken verfügen sowohl die beiden Peer-Implementationen, als auch der Monitor über die Möglichkeit, zufällig erzeugte Verkehrsinformationen in das *CAN*-Netz einzuspeisen. Dabei verfügen die Peers über die Optionen 1. und 2., der Monitor nur über Option 1.:

1. Es wird für 500 zufällig aus der Kantenmenge ausgewählte Kanten jeweils ein *TrafficUpdate* mit einer zufällig gewählten Fahrzeit aus dem Intervall

   $$[0\ ,\ 2 \cdot \text{aktuelle Durchschnittszeit der Kante}]$$

   versendet.
2. Für alle noch zu durchfahrenden Kanten des aktuell schnellsten Weges werden *TrafficUpdates* versendet, die zufällige Fahrzeiten im Bereich

   $$[\text{aktuelle Durchschnittszeit der Kante}\ ,\ 2 \cdot \text{aktuelle Durchschnittszeit der Kante}]$$

   enthalten.

Dadurch ist es auf einfache Weise möglich zu demonstrieren, wie Verkehrsinformationen sich durch das *CAN*-Netz verbreiten. Man benötigt nur wenige Peers um den Effekt zu verdeutlichen.

Kapitel 4

Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurden Anwendungen zur Demonstration eines Peer-to-Peer-basierten Verkehrsinformationssystems erstellt und die zugrunde liegenden Verfahren erläutert.

Dabei wurden sowohl eine Peer-Software für PCs (*JavaSEPeer*), als auch eine Peer-Software für heutige Mobilgeräte (*JavaMEPeer*) entwickelt. Um zu demonstrieren welche Verkehrsdaten sich gerade im Peer-to-Peer Netzwerk befinden, wurde außerdem ein Monitor (*JavaSEMonitor*) entwickelt, der diese Informationen auf einer Straßenkarte darstellt.

Um Verkehrsinformationen auszutauschen ist es nahezu unerlässlich, dass alle Teilnehmer über die gleichen Kartendaten verfügen. Daher haben wir einen Parser für die XML Daten der lizenzfreien Weltkarte *OpenStreetMap* geschrieben und nutzen den daraus erstellten Routinggraphen als Grundlage für die Navigation und den Verkehrsinformationsaustausch.

Als Peer-to-Peer Netzwerk wird eine in einigen Punkten abgeänderte Version von *CAN* genutzt:

1. Wir verzichten auf eine gleichförmige Hashfunktion und speichern die Verkehrsdaten stattdessen „an“ ihrer geographischen Position.
2. Da durch 1. Informationen zu benachbarten Straßen nun im abgewandelten *CAN*-Netzwerk in benachbarten Peers oder sogar im gleichen Peer gespeichert sind,

benutzen wir Nachrichtenaggregation um den Overhead für Verkehrsinformationsanfragen und Updates zu verringern.

3. Da uns nicht nur die letzte Messung zu einer Straße interessiert, sondern vielmehr der Mittelwert aus allen Messungen der vergangenen Minuten, speichern wir nicht nur einen Wert pro Kante, sondern verwalten eine ganze Reihe von Daten als Aggregation von Messwerten.

4. Wir haben den von den *CAN*-Entwicklern vorgeschlagenen Vorgang des Joinens in ein neuartiges, von uns *GeoJoin* genanntes Verfahren geändert, um zum einen den Peer im *CAN*-Netzwerk „näher“ an die Peers zu bringen, die Informationen über die Straßen bereithalten, über die er navigieren, wird und zum anderen um ein *Loadbalancing* zu implementieren.

Durch diese Änderungen wird die Anzahl der durch das Peer-to-Peer Netzwerk zu routenden Verkehrsinformationsabfragen pro Navigationsroute deutlich reduziert.

Der Hauptgrund für den Verkehrsinformationsaustausch ist das Verringern der Fahrzeit, also die Findung des schnellsten Weges von einem Startpunkt zu einem Zielpunkt. Da daher während der Routenberechnung versucht wird die Fahrzeit zu minimieren, haben wir uns dazu entschieden keine Durchschnittsgeschwindigkeiten auszutauschen, sondern Durchschnittszeiten, die zum „Durchfahren“ einer Kante benötigt werden.

Während der Implementierung wurden einige Probleme und Schwierigkeiten deutlich, für die wir Lösungen entwickelt haben:

1. Es zeigte sich, dass *JavaME* keine Möglichkeit zur sicheren Erkennung von IP und Port einer selbst erstellten *DatagramConnection* bietet. Um dieses Problem zu umgehen nutzen wir die Antwort des *BootStrapServers*, der uns die Daten unserer eigenen Verbindungen mitteilt.

2. Auch *JavaSE* hat ein ähnliches Problem. Es ist zwar grundsätzlich dazu in der Lage IP und Port seines *DatagramSockets* festzustellen, verfügt der verwendete Rechner jedoch über mehr als ein Netzwerkinterface, so wird der *DatagramSocket* standardmäßig so geöffnet, dass er auf allen Interfaces horcht. Die Funktion zur Bestimmung der IP liefert dann 0.0.0.0. Diese Information hilft uns jedoch nicht weiter, wenn wir einem Kommunikationspartner mitteilen wollen wie er uns er-

reichen kann. Daher benutzen wir auch für den *JavaSEPeer* den Workaround über den *BootStrapServer*.

3. Die *KVM* stellt *JavaME*-Programmen nur wenig Arbeitsspeicher zur Verfügung. Dies führte dazu, dass wir ein Hauptaugenmerk bei der Implementierung auf niedrigen Speicherverbrauch gelegt haben.
4. Wir haben festgestellt, dass die Berechnung des schnellsten Weges in Graphen in *JavaME* durch zwei Optimierungen in der Programmiertechnik stark beschleunigt werden kann und haben die passenden Änderungen implementiert.
5. Die Analyse der *OSM*-Kartendaten ergab, dass die Daten in keinem für das Routing brauchbaren Format vorliegen. So entsprechen die *OSM Ways* selten Kanten, die wir im Routinggraphen direkt verwenden können, denn oftmals befinden sich Kreuzungen auf „inneren“ Wegpunkten von *OSM Ways*. Um die *OSM*-Daten nutzen zu können, müssen also erst einmal alle Kreuzungen gefunden werden. Außerdem zeigte sich, dass kleine Fehler in den *OSM*-Daten dazu führen, dass Kreuzungen nicht mehr erkannt werden können. Ferner haben wir erläutert, dass man den in *OSM Ways* hinterlegten Straßentyp dazu nutzen kann um eine plausible Annahme der für die jeweilige Straße beziehungsweise den Straßenabschnitt geltenden Höchstgeschwindigkeit zu finden.

Unsere Implementierung zeigt dennoch deutlich, dass heutige Mobilgeräte leistungsfähig genug sind um Navigationsaufgaben bewältigen und gleichzeitig Verkehrsinformationen über ein selbstorganisierendes Peer-to-Peer Netzwerk auszutauschen.

Kapitel 5

Ausblicke

Während der Erstellung des Demonstrators wurden weitere mögliche Probleme und interessante Aspekte deutlich, deren Untersuchung den zeitlichen Rahmen dieser Arbeit überschritten hätte, die aber Thema weiterer Arbeiten werden könnten.

5.1 Behandlung von Peerausfällen

Der *JavaMEPeer* und der *JavaSEPeer* sind zwar zu Demonstrationszwecken einsatzfähig, es wurde jedoch – wie vorher vereinbart – kein Verfahren implementiert, das dafür sorgen würde dass Zonen von Peers, die ohne ordentliche Zonenübergabe das Netz verlassen, von Nachbarn übernommen beziehungsweise neu gebildet werden. Während der Testphase der Peers zeigte sich zudem, dass ein solches Verfahren nicht nur bei dem Problem von droppenden Peers wichtig ist, sondern auch wenn das Netzwerk durch massenhafte *Joins* oder *Leaves* innerhalb kurzer Zeit gestresst wird, denn durch das gleichzeitige Splitten zweier benachbarter Zonen kann es auch zum Verlust von Nachbarschaftsinformationen kommen. Sollte einer der Peers weiterentwickelt werden, so wäre die Behandlung von Knotenausfällen ein wichtiger Aspekt. Lösungen dafür wurden unter anderem in [Rat02] vorgestellt.

5.2 Redundanz

Je nach Größe der verwendeten Straßenkarte kommt es bei steigender Peeranzahl früher oder später dazu, dass Peers Zonen verwalten, die keine Kantenziele (und damit Informationen) enthalten. Das hat nicht nur den Nachteil, dass es gegenüber anderen Peers, die mehrere Straßen verwalten, unfair ist. Es führt zudem dazu, dass sich das Routing im Peer-to-Peer Netz durch die längeren Wege verschlechtert. Man könnte die „überzähligen“ Peers jedoch dazu nutzen um Redundanzen zu schaffen. So wäre es denkbar, ein weiteres, paralleles *CAN*-Netz aufzubauen und jeden Peer in einem der Netze einen aktiven, zonenverwaltenden Part übernehmen zu lassen, und einen passiven (nur abfragenden) in dem anderen. Natürlich sollten dabei die *TrafficUpdates* in beide Netze geschickt werden. Dadurch wäre ein Verfahren zur Behandlung von Peerausfällen in den meisten Fällen sogar in der Lage, die Daten einer Zone wiederherzustellen.

5.3 NAT in UMTS/GPRS

Obwohl mit der Verbreitung von UMTS eigentlich auch IPv6 großflächig Einzug in die Mobilfunknetze nehmen sollte, setzen die Mobilfunkprovider auch heute noch auf IPv4 in ihren *UMTS* und *GPRS* Netzen. Durch die Knappheit der IPv4 Adressen und vielleicht auch zur Erschwerung der Nutzung von Peer-to-Peer Netzen, wird hier auch heute noch *Network Address Translation (NAT)* eingesetzt. Vor einer breiten Nutzung des *JavaME-Peers* in Mobilfunknetzen muss also geprüft werden, welche Änderungen gegebenenfalls nötig sind um trotz *NAT* in einem *CAN*-Netz zu arbeiten.

5.4 Sicherheit

Der Aspekt der Sicherheit und des Schutzes der Privatsphäre sollte vor dem produktiven Einsatz des hier vorgestellten Peer-to-Peer Systems untersucht werden. So ist es in der jetzigen Implementation zum Beispiel möglich, falsche Verkehrsinformationen zu versenden um eine Strecke für andere Teilnehmer uninteressant zu machen und sich so selbst eine relativ freie Fahrt zu verschaffen.

5.5 Bootstrapping

Der in dieser Arbeit vorgestellte *BootStrapServer* ist zwar ausreichend für einen Demonstrator, für ein produktives System wäre es aber wünschenswert, wenn dieser *Single Point of Failure* durch ein robusteres Bootstrapping ersetzt würde. Mögliche Bootstrapping-Verfahren für Peer-to-Peer Netze werden unter anderem in [GG08] und [KAD+04] vorgestellt und analysiert.

5.6 Tests auf realen Geräten

Der *JavaMEPeer* wurde während der Entwicklung hauptsächlich im Emulator des *Sun Wireless Toolkits* getestet. Es fanden lediglich einige kurze Tests auf einem *JavaME*-tauglichen PDA statt.

Bevor das System zu einem vollwertigen Navigationssystem mit „dynamischem Routing“ und Verkehrsinformationsaustausch wird, müssen Tests auf realen Geräten folgen. Denn obwohl *JavaME* eigentlich standardisiert ist, benötigt jedes Gerät eine gerätspezifische *KVM* und es zeigt sich, dass sich nicht alle *KVM*s gleich verhalten.

5.7 GPS Anbindung

Momentan berechnen die Peer-Implementationen ihre aktuelle Position aus ihren simulierten Bewegungsdaten. Für ein fertiges Navigationssystem müssten diese Daten natürlich – zum Beispiel durch die Auswertung der Daten eines GPS-Empfängers – ermittelt werden. Für *JavaME* existiert bereits eine Open Source Bibliothek *J4ME* [SO], welche die Einbindung eines GPS-Empfängers bei unterstützten Geräten erleichtern kann.

Literaturverzeichnis

[ATS04] ANDROUTSELLIS-THEOTOKIS, Stephanos; SPINELLIS, Diomidis: A survey of peer-to-peer content distribution technologies. In: *ACM Computing Surveys (CSUR)* 36 (2004), December, Nr. 4, 335–371. http://dx.doi.org/http://dx.doi.org/10.1145/1041680.1041681. DOI http://dx.doi.org/10.1145/1041680.1041681. ISSN 0360–0300.

[GG08] GAUTHIERDICKEY, Chris; GROTHOFF, Christian: Bootstrapping of Peer-to-Peer Networks. In: *Proceedings of DAS-P2P*. Turku, Finland : IEEE, August 2008.

[jsr] *JSR 75: PDA Optional Packages for the J2ME Platform*. http://jcp.org/en/jsr/detail?id=75.

[KAD+04] KARBHARI, Pradnya; AMMAR, Mostafa H.; DHAMDHERE, Amogh; RAJ, Himanshu; RILEY, George F.; ZEGURA, Ellen W.: Bootstrapping in Gnutella: A Measurement Study. In: *PAM*, 2004, S. 22–32.

[Koe08] KOEGEL, Markus: *A Peer-to-Peer System for Dynamic Location-Related Data*, Department of Computer Science, Heinrich Heine University Düsseldorf, Diplomarbeit, April 2008

[Kow07] KOWALSKI, Martin: *A Decentralized System for Sharing Traffic Information*, Department of Computer Science, Heinrich Heine University Duesseldorf, Diplomarbeit, November 2007

[Mel07] MELLENTHIN, Joachim: *Dezentraler Informationsaustausch zwischen Fahrzeugen über Infrastrukturnetzwerke*. Bachelor's thesis, März 2007.

[Ope08a] OPENSTREETMAP: *OpenStreetMap*. http://www.openstreetmap.com/. Version: 2008.

[Ope08b] OPENSTREETMAP: *OSM Karte Neuss-Düsseldorf*. http://www.openstreetmap.org/export/?lat=51.2&lon=6.75&zoom=12&layers=B000FTFT. Version: 2008.

[Pfe07] PFEIFFER, Michael: *Java Micro Edition - Mobile Anwendungen mit der MIDP 2.0 entwickeln*. Galileo Press, 2007

[Rat02] RATNASAMY, Sylvia P.: *A scalable content-addressable network*, Diss., 2002. Chair-Scott Shenker and Chair-Ion Stoica

[RSK+07] RYBICKI, Jedrzej; SCHEUERMANN, Björn; KIESS, Wolfgang; LOCHERT, Christian; FALLAHI, Pezhman; MAUVE, Martin: Challenge: Peers on Wheels - A Road to New Traffic Information Systems. In: *MobiCom 2007: Proceedings of the 13th Annual International Conference on Mobile Computing and Networking*, Montreal, Quebec, Canada, 2007, S. 215–221.

[SO] SCORE OUT, Inc.: *J4ME*. http://code.google.com/p/j4me/.

[Umw] UMWELT- UND PROGNOSE-INSTITUT E.V.: *Folgen einer globalen Massenmotorisierung*. www.upi-institut.de/upi35.htm.

[wtk08] *Sun Java Wireless Toolkit for CLDC*. http://java.sun.com/products/sjwtoolkit/. Version: 2008.

[Zok07] ZOK, David: *A Distributed Data Structure for Disseminating Traffic Information via Publish/Subscribe*, Department of Computer Science, Heinrich Heine University Duesseldorf, Diplomarbeit, November 2007

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Neuss, 14. November 2008 Norbert Goebel