Institut für Parallele und Verteilte Systeme

Abteilung Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 3115

Design and Implementation of a Scalable Mobile Target Tracking System
Konzeption und Implementierung eines skalierbaren Systems zur Verfolgung mobiler Objekte

Daniel Alexander Biliniewicz

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer:	Dr. rer. nat. Frank Dürr
begonnen am:	8. Dezember 2010
beendet am:	9. Juni 2011
CR-Klassifikation:	C.2.1, C.2.3, C.2.4

Abstract

Today, many mobile devices like smart phones occupy an increasing number of various sensors. The usage of these, particularly in urban areas highly available, devices as mobile sensor network, is currrently establishing a new research field called public sensing.

This work considers the usage of such a system for mobile target tracking. Therby we concentrate on efficiency concerning the total transmitted data between mobile nodes and the infrastructure, while keeping high effectiveness regarding reported object positions. To make this possible, we develop a grid-based approach, that selectively distributes search tasks to the mobile devices while considering object mobility.

The developed approach was analyzed by using the network simulator ns2. To measure efficiency and effectiveness we compared the grid-approch with different grid sizes to a basic approach, by using realistic mobility data. It shows that the grid-approach has nearly the same effectiveness as the basic approch, but minimizing the amount of total transmitted data.

Zusammenfassung

Heutzutage besitzen viele mobile Geräte wie Smartphones eine immer größere werdende Anzahl verschiedener Sensoren. Die Nutzung dieser, insbesondere in städtischen Gebieten, in hoher Zahl vorhandenen Geräte als mobiles Sensornetzwerk, eröffnet zur Zeit einen neuen Forschungsbereich, der als public sensing bezeichnet wird.

In dieser Arbeit betrachten wir hierbei die Nutzung eines solchen Systems zur Verfolgung mobiler Objekte. Dabei legen wir unseren Fokus auf die Effizienz, hinsichtlich der gesamt übertragenen Daten zwischen mobilen Geräten und der Infrastruktur, bei gleichzeitig hoher Effektivität bezüglich der gemeldeten Objektpositionen. Um dies zu realisieren entwickeln wir einen auf einem Gitter basierenden Ansatz, der unter Einbeziehung der Mobilität der Objekte Suchaufträge gezielt an die mobilen Geräte verteilt.

Der entworfene Ansatz wird in Simulationen mittels des Netzwerksimulators ns2 hinsichtlich seiner Effizient und Effektivität für verschiedene Gittergrößen untersucht. Hierfür vergleichen wir ihn mittels realistischer Mobilitätsdaten mit einem maximal effektiven Basis-Ansatz. Es zeigt sich, dass der Gitter-Ansatz bei nahezu gleichbleibender Effektivität, im Vergleich zum Basis-Ansatz, die Menge der übertragenen Daten minimiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Viele mobile Geräte besitzen heutzutage neben den zahlreichen Kommunikationsmöglichkeiten zusätzliche Sensoren, mit Hilfe derer es möglich, ist die nähere Umgebung zu erfassen. Somit lassen sich Informationen über ein Gebiet durch Sensoren erlangen, die dort ohne vorher geplante Platzierung vorhanden sind und somit Sensornetze großen Ausmaßes realisieren. Eine typische Aufgabenstellung ist das Verfolgen oder auch die Positionsbestimmung mobiler Objekte über eine gewisse Zeitspanne oder zu einem bestimmen Zeitpunkt. Dafür lassen sich Sensoren wie zum Beispiel GPS und RFID Leser verwenden, die zukünftig in vielen Smartphones vorhanden sein werden. Um die Mitarbeit zu solch einem System auch für die Besitzer solcher mobilen Geräte attraktiv zu machen ist es nötig, das Verfahren so resourcenschonend wie möglich zu halten, denn nur durch eine ausreichend hohe Zahl an teilnehmenden mobilen Geräten lässt sich eine gute Abdeckung eines Gebiets gewährleisten. Weiterhin spielt die Skalierbarkeit des Systems eine große Rolle um es großflächig einsetzen zu können.

1.1.1 Public Sensing

Unter Public-Sensing versteht man im Allgemeinen die Nutzung alltäglicher mobiler Geräte als Sensorknoten. Typische mobile Geräte sind hierbei Mobiltelefone oder im allgemeinen Smartphones, welche Fußgängern bei sich tragen. Diese nahezu allgegenwärtigen Geräte werden beauftragt bestimmte Daten über ihre unmittelbare, d.h. durch ihre Sensoren erfassbare, Umgebung aufzuzeichnen. Diese Sensordaten werden dann zusammen mit zusätzlichen Kontextinformationen dem System zur Verfügung gestellt. Auf diese Art und Weise ist es möglich verschiedene Anwendungsszenarien zu realisieren, die mit herkömmlichen Sensornetzen aufgrund der Ausdehnung des Gebiets nicht realisiert werden können.

Ein in der Literatur verbreiteter Begriff ist Urban Sensing[CHK08] [CEL+06] oder auch People-Centric Sensing[CEL+08]. Der Begriff Urban Sensing leitet sich davon ab, dass die Gebiete in denen typische Anwendungsfälle liegen städtisch sind, da dort das Aufkommen von mobilen Geräten, die als Sensorknoten in Frage kommen entsprechend hoch ist. Der zweite Begriff hingegen betont den Sachverhalt, dass durch das Nutzen der mobilen Geräte, die Menschen bei sich tragen, als Konsequenz des Umfeld genau dieser Menschen durch Sensoren erfasst wird und somit im Mittelpunkt steht.

Es gibt zahlreiche Anwendungsfälle für Public-Sensing, wie beispielsweise die großflächige

Erhebung von Umweltdaten. Hier betrachten wir als Anwendung des Verfolgen mobiler Objekte mittels Public Sensing.

Vergleich zu drahtlosen Sensornetzen

Für gewöhnlich werden drahtlose Sensornetze darauf ausgelegt in einem festgelegten Gebiet Sensordaten über einen festgelegten Zeitraum zu erfassen. Dabei werden die Sensorknoten nach einem vorgegebenen Muster verteilt. Die Sensorknoten erfassen dann Sensordaten ihrer Umgebung und die so erhaltenen Ergebnisse werden in Richtung einer Nachrichtensenke von Knoten zu Knoten durch das Netzwerk transportiert. Dies ist nötig, da nicht jeder Knoten in direktem Kontakt mit der Nachrichtensenke steht. Ein Sensorknoten ist dabei ein einfaches Gerät, bestehend aus den benötigten Sensoren und drahtloser Kommunikationsmöglichkeit. Das Hauptaugenmerk bei diesen Systemen liegt auf einem möglichst niedrigen Energieverbrauch der Sensorknoten, da diese im Allgemeinen ihre Energiereserven nicht wieder erneuern können, das System jedoch über eine möglichst lange Zeitspanne laufen soll. Dazu werden beispielsweise Methoden verwendet, die jeden Knoten möglichst gleich belasten was das generieren von Sensordaten angeht und auch bei der Meldung der Ergebnisse werden unter anderem Techniken zur Aggregation der Daten benutzt um das Datenvolumen gering zu halten. Diese Systeme können nur in begrenzter Größe eingesetzt werde, da zum einen eine Flächendeckende Verteilung zu teuer wäre und zum anderen die großflächige Verteilung von Sensorknoten mit begrenzter Funktionsdauer wenig Sinn macht.
Vergleicht man dies nun mit einem Public-Sensing System, so stellt sich dabei nicht die Frage der Ausbringung der Sonsorknoten, denn dies geschieht von selbst. Daher ist die Größe des Gebiets in dem das System eingesetzt werden kann nur durch die Verteilung bzw. Dichte der alltäglichen mobilen Geräte beschränkt. So lassen sich auf einfach Art ganze Städte abdecken. Der Nachteil ist, dass nur dort Sensordaten generiert werden können, wo sich gerade ein mobiles Gerät befindet.
Weiterhin liegt bei Public-Sensing der Schwerpunkt nicht so sehr auf einer lange Funktionsdauer, da die mobilen Geräten von den Benutzern selbst wieder aufgeladen werden. Jedoch sollte eine Public-Sensing Anwendung ein Gerät nicht zu sehr belasten, damit die Teilnahme am System für den Besitzer attraktiv bleibt.
Ein markanter Unterschied ist die Tatsache, dass in gewöhnlichen drahtlosen Sensornetzen die Sensorknoten komplett kontrolliert werden können, insbesondere bei mobilen Sensorknoten wohin sie sich bewegen sollen. Bei Public-Sensing ist dies im Allgemeine nicht der Fall, sprich der Besitzer des mobilen Geräts bewegt sich uneingeschränkt. Daher ist eine komplette Abdeckung eines Gebiets zu jeder Zeit , wie sie von gewöhnlichen Sensornetzen angestrebt wird, nicht in jedem Fall möglich. Die Erhebung von Sensordaten ist daher rein opportunistisch und je nach Dichte der mobilen Geräte in manchen Bereichen unvollständig. Beispielsweise kann sich in einem Public-Sensing System ein Objekt unbemerkt eine Straße entlang bewegen, falls sich dort gerade kein mobiles Gerät befindet. In gewöhnlichen Sensornetzen würde man die Knoten so platzieren und koordinieren, dass in genanntem Fall nach Möglichkeit jedes Objekt registriert wird.

1.1.2 Verfolgung mobiler Objekte

In dieser Arbeit betrachten wir Public-Sensing System, welches dazu benutzt wird um mobile Objekte zu verfolgen. Ein mögliches Szenario wäre die Verfolgung eines Busses der durch die Stadt fährt und die Aktualisierung von Ankunftszeiten auf Basis der aufgezeichneten Positionen. Ein weiterer Anwendungsfall wäre die Suche nach einem entlaufenen Haustier dessen Halsband einen RFID-Tag beinhalted, oder allgemein die kontinuierliche Überwachung seines Aufenthaltsbereichs. Neben diesen Beispielen, gibt es noch zahlreiche Andere Anwedungsfälle bei denen die Position mobiler Objekte kontinuierlich übermittelt werden muss.
Was die Systemkomponenten angeht, so gibt es im Allgemeinen sehr viele Endgeräte, die von einer Serverinfrastruktur beauftragt werden nach bestimmten Objekten zu suchen. Unser primärer Fokus liegt dabei auf der Skalierbarkeit im Sinne der Reduktion der Last auf den Servern bzw. den zwischen mobilen Knoten und Serverinfrastruktur übertragenen Datenmengen.
Würde man einen naiven Ansatz wählen, der einfach jedes gefundene Objekt in Reichweite an die Infrastruktur übermittelt, so müssten nur genug Teilnehmer des Systems beispielsweise durch ein Kaufhaus laufen, wo viele Gegenstände mit RFID-Tags versehen sind. Generell kann man sagen, dass nur ein geringer Teil von Objekten in der Umgebung tatsächlich gesucht wird. Daher macht es zum einen generell Sinn Suchaufträge an die mobilen Knoten zu verteilen, damit diese nicht jedes Objekt melden. Wir versuchen die Effizienz zu erhöhen, indem wir nach einem Objekt nur in bestimmten Gebieten suchen. Dafür ist es nötig den möglichen Aufenthaltsbereich des Objekts abzuschätzen. Wir versuchen so Nachrichten einzusparen und erhöhen somit auch indirekt die Energieeffizienz.

1.2 Ziele

In dieser Arbeit betrachten wir einen speziellen Anwendungsfall für ein Public-Sensing-System, die Verfolgung mobiler Objekte. Die Verfolgung vieler mobiler Objekte durch eine große Anzahl von mobilen Knoten ist bezüglich der Skalierbarkeit keine einfache Aufgabe. Ein naiver Ansatz, bei dem jedes gefundene Objekt gemeldet wird skaliert nicht, da zu viele Objekte gemeldet werden, die gar nicht gesucht werden. Daher Verfolgen wir den Ansatz Suchaufträge an die mobilen Knoten zu versenden. Jedoch ist die Verteilung der Suchaufträge keine einfache Aufgabe, wenn viele mobile Objekte von einer sehr großen Anzahl von mobilen Geräten gesucht werden soll. Beispielsweise skaliert es nicht, wenn die mobilen Geräte ständig die Menge aller gesuchten Objekte abfragen.
Daher ist es das Ziel dieser Arbeit Konzepte für ein skalierbares System zu entwerfen, zu implementiern und zu evaluieren. Der wesentliche Beitrag dieser Arbeit zur Lösung des beschriebenen Problems ist der Entwurf eines Gitter-Ansatzes, welcher die Menge der übertragenen Daten zwischen Infrastruktur und mobilen Knoten minimiert und dabei die Effektivität nahezu maximal hält.

1.3 Aufbau der Arbeit

In Kapitel 2 werden verwandet Arbeiten aus dem Bereich Public-Sensing kurz vorgestellt und unterschiede bzw. Gemeinsamkeiten mit dem hier entwickelten System verdeutlicht. In Kapitel 3 wird das grundlegende Systemmodell vorgestellt auf wir dann unsere entwickelten Ansätze aufbauen. In Kapitel 4 werden zunächst grundsätzliche Methoden aufgeführt, mittels deren Weiterführung und Anpassung an das Systemmodell wir zwei verschiedene Ansätze entwickeln. Kapitel 5 beschreibt dann die Umsetzung der Ansätze in einer Simulation. Anschließend werden in Kapitel 6 die Simulationsergebnisse vorgestellt und analysiert. In Kapitel 7 fassen wir die Arbeit nochmals abschließend zusammen und diskutieren kurz weitere Verbesserungsmöglichkeiten.

2 Verwandte Arbeiten

In diesem Kapitel stellen wir verwandte Arbeiten im Bereich Public-Sensing vor, die relevant erscheinen. Da mit dieser Arbeit ein eher noch unerforschter Bereich betrachtet wurde, gibt es wenige Prinzipien oder gar Algorithmen aus bisherigen Arbeiten, auf denen sich aufbauen lässt.
Wir Stellen zunächst Arbeiten vor, die mehr eine Übersicht über das Forschungsgebiet Public-Sensing und zukünftige Visionen erläutern.

2.1 Urban Sensing: out of the woods

In [CHK08] wird zunächst grob ein statisches Sensornetz beschrieben, welches vorher definierte Messaufgaben durchführt und dargelegt, dass ein so spezialisiertes System nicht in der Größenordnugn einer Stadt genutzt werden kann. Da es zum einen zu teuer wäre und zum anderen die Berechtigung fehlt solche Sensornetze flächendeckend in städtischen Gebieten zu installieren.
Daher liegt der Schluss nahe die Smartphones als mobile Sensorknoten einzusetzen. Also weg statischen geplanten Sensornetzen hin zur Verteilung von Aufträgen zur Erhebung bestimmter Messdaten durch die Öffentlichkeit. Dafür wird von den Autoren auch der Begriff "participatory sensing" gebraucht, der in diesem Fall ausdrückt, dass die am System teilnehmenden Menschen Zusammenarbeiten um eine sonst nicht zu schaffende Messaufgabe zu bewerkstelligen. Dieser Begriff meint hier im Unterschied zu [WWDR11] nicht die Abstimmung der mobilen Geräte untereinander, um die geforderte Aufgabe besonders energiersparend zu gestalten. Viel mehr sehen die Autoren nicht mobile Geräte, die von ihren Besitzern zur Verfügung gestellt werden, um darauf prinizipiell beliebige Anwendungen laufen zu lassen, sondern eine aktive Rolle des Menschen. Sie gehen sogar soweit, dass sie die Vision einer "public sphere" beschreiben, in der Menschen ihre Sensordaten sammeln und sie einander frei zur Verfügung stellen. Dies weicht in seiner Gesamtheit ziemlich stark von dem ab was wir unter Public-Sensing verstehen. Wir sehen wie schon weiter obenbeschrieben keine aktive Teilnahme des Menschen am System, sondern nur die Bereitschaft Resourcen zur Verfügung zu stellen. Daher spielt es mehr eine Rolle die Teilnahme attraktiv zu halten, indem Public-Sensing-Systeme resourcenschonend angelegt werden.

2.2 People-Centric Sensing

In [CEL+08] und [CEL+06] wird der Begriff "people-centric sensing" verwendet, um auszudrücken, dass durch die Verwendung von Mobiltelefonen als Sensorknoten der Mensch und seine unmittelbare Umgebung im Fokus liegen, was die Erhebung von Sensordaten anbelangt. Hierbei sehen sie den Benutzer in zwei verschiedenen Rollen. Zum einen, ähnlich wie im vorherigen Abschnitt, in einer teilnehmenden Rolle, was auch hier als "participatory sensing" bezeichnet wird. Dabei nimmt der Mensch aktiv am System teil und entscheidet darüber welche Daten er zur Verfügung stellen möchte. So sehen sie bei dieser Art von Sensordatenerhebung den Schwerpunkt bei Anwendungen, die den Nutzer unterstützen. Die zweite Rolle des Benutzers bezeichnen sie als rein opportunistischen Ansatz, bei dem der Benutzer sein mobiles Gerät dem System zur Verfügung stellt und das System entscheidet, ob ein Gerät zur Erhebung von Daten benutzt wird. Die Besitzer erlauben dabei, dass Anwendungen auf ihrem mobilen Gerät laufen, haben jedoch keinen Einfluss darauf welche Anwendungen zu einem bestimmten Zeitpunkt aktiv sind.
In [CEL+06] wird zudem eine Architektur skizziert, Gemeinsamkeiten mit dem von uns verwendeten Systemmodell aufweist. So sehen die Autoren einen dreistufigen Aufbau des Systems. Ganz oben befinden sich die Server-Infrastruktur, welche die Aufgabe übernimmer alle Systemkomponenten untereinder zu verbinden. Danach sehen sie sogenannte Gateways, die den mobilen Knoten den Zugang zur Infrastruktur ermöglichen. Auf unterster Stufe befinden sich die mobilen Sensorknoten zusammen mit statischen Sensorknonten. Dabei können Aufträge auch an statische Sensorknoten verteilt werden, indem sie einem mobilen Knoten mitgegeben werden, der am statischen Sensorknoten vorbeikommt. In umgekehrter Weise werden die gemessenen Sensordaten eingesammelt und an die Infrastruktur weitergeleitet.

2.3 MobEyes

In dieser Arbeit wird ein an Fahrzeuge gebundenes Sensorsystem betrachtet. Dabei unterliegen die einzelnen Sensorknoten, die durch Fahrzeuge realisiert werden prinzipiell nicht den Resourcenbeschränkungen wie es bei Mobiltelfonen der Fall ist. Daher können sie eine sehr große Menge an Sensordaten generieren. Die Aufgabe hierbei ist nun aus dieser riesigen Datenmenge einen bestimmten Teil auszuwählen, beispielsweise aufgenommene Bilder von Nummernschildern. Dabei ist es aufgrund der Datenmenge nicht möglich diese an eine zentrale Stelle zu übermitteln und dort auszuwerten. Stattdessen verbleiben die Daten zunächst bei den Sensorknoten, welche aus den aufgenommenen Sensordaten sogenannten Summaries, also Zusammenfassungen erstellen. Dabei beinhaltet so eine Zusammenfassung für die gespeicherten Daten beispielsweise wann und wo diese aufgenommen wurden. Anhand dieser Meta-Daten lassen sich die eigentlichen Daten auswählen. Das Ziel ist es alle generierten Summaries in einem bestimmten Gebiet einzusammeln. Dabei geschieht das Sammeln durch sogennante "police agents", wobei sich der Name aus dem gewählten Szenario der Autoren ergibt.
Interessanter sind die Mechanismen zur Verbreitung und zum Sammeln von von Sum-

maries. Zum einen erzeugt der Knoten, welcher die eigentlich Daten besitzt regelmäßig Summaries und verteilt diese oppertunistisch über Ad-Hoc Kommunikation an andere Knoten. Wobei hier zwischen einer aktiven und einer passiven Verbreitung unterschieden wird. Passiv bedeutet, dass nur der Urheber der Daten die Summaries verbreitet. In der aktiven Variante verteilt jeder Knoten Summaries. Der sammelnde Knoten fragt zudem aktive die Summaries ab. Dies geschieht mittels eines Bloom-Filters in dem er spezifiziert welche Datenpakete er schon besitzt. Die Nachbarn senden daraufhin nur noch fehlende Datenpakete.

2.4 Locating Objects Using Mobile Phones

Diese Arbeit[FBRK07] entspricht am ehesten den von uns untersuchten Fragenstellungen. Dabei ist die allgemeine Grundidee mobile Sensorknoten zu benutzen, um passive Objekte zu finden. Dabei wird hier ähnlich zu unserem Ansatz die Idee verwendet Anfragen in ihrer räumlichen Ausdehnung einzuschränken, um möglichst effizient zu suchen. Jedoch gibt es einige wesentliche Unterschiede. Zum einen wird nicht die Annahme getroffen, dass es sich bei den gesuchten Objekten um mobile Objekte handelt und diesbezüglich werden auch keine Maßnahmen getroffen. Gemeinsam ist eine Art “Query-Scoping”, das heißt es wird anhand von Heuristiken bestimmt in welche Sensoren zum Suchen des Objekts benutzt werden sollen. Hierbei wird jedoch auf Vorabinformation über das zu suchende Objekt aufgebaut, wohingegen wir davon ausgehen, dass wir außer der maximalen Geschwindigkeit keinerlei zusätzlich Information als Ausgangsbasis haben.
Insgesamt wird eher ein intelligenter Dienst zum Suchen von Objekten, die in einem bestimmten Breich bzw. Umfeld verloren wurden. Bei unserem Ansatz hingegen sind die Objekte ständig in Bewegung und so reicht es nicht initial einen Suchbereich zu bestimmen. Auch wird stark auf Suchheuristiken, wie beispielsweise werden Charakteristiken wie “der Ort an dem ich mich zuletzt befunden habe” oder ”der Ort an dem das Objekt zuletz gesehen wurde“ mit einbezogen. Wir berücksichtigen in dieser Arbeit viel stärker die Mobilität der Objekte und versuchen deren Position anhand der durch das System selbst erhobenen Daten den Bereich in dem sich da Objekt befinden kann abzuschätzen.

3 Systemmodell und Problembeschreibung

3.1 Systemmodell

In dieser Arbeit wird ein Public-Sensing-System betrachtet, dessen Aufgabe es ist mobile Objekte zu verfolgen. Um Objekte zu orten, werden mobile Geräte (fortan als mobile Knoten bezeichnet), wie beispielsweise Smartphones, benutzt, welche geeignete Sensoren besitzen. Das Gebiet, in dem Objekte gesucht werden, ist prinzipiell nicht beschränkt und somit kann weltweit gesucht werden. Da bei dieser Art von System typischerweise viele mobile Knoten benötigt werden, bietet sich als Beispielszenario eine Innenstadt an oder generell ein Gebiet in dem die Dichte an geeigneten mobilen Geräten relativ hoch ist, um eine möglichst gute Abdeckung zu gewährleisten. Die Anzahl der mobilen Objekte nach denen gesucht werden kann bewegt sich im Millionenbereich, wobei wir davon ausgehen, dass nur ein kleiner Teil davon tatsächlich zu einem bestimmten Zeitpunkt gesucht wird.

3.1.1 Komponenten des Systems

Das von uns betrachtete System besteht aus den folgenden Komponenten, die jeweils weiter unten genauer beschrieben werden:

- mobile Knoten
- mobile Objekte
- Anfrage-Verwaltungs Knoten
- Wurzel Knoten
- Suchauftrag-Koordinator (kurz: Koordinator)

Die Verbindung zwischen den Komponenten ist in Abbildung 3.1 dargestellt. Den Einstiegspunkt für die Anwendung bildet ein Anfrage-Verwaltungs Knoten. Über diesen kann mittels eines API-Aufrufs eine Suchanfrage an das System gestellt werden. Eine Suchanfrage besteht aus der ID des gesuchten Objekt, der Dauer der Suchanfrage und der Angabe, ob während dieser Zeit permanente Aktualisierungen der Position des Objekts verlangt werden, was wir als permanente Anfrage bezeichnen, oder ob ein einmaliger Fund des Objekts, innerhalb der angegebenen Dauer, die Anfrage erfüllt, was wir als einmalige Anfrage bezeichnen. In letzterem Fall wird die Anfrage entfernt, sobald ein mobiler Knoten das Objekt mit der angegebenen ID gefunden hat, ansonsten werden alle

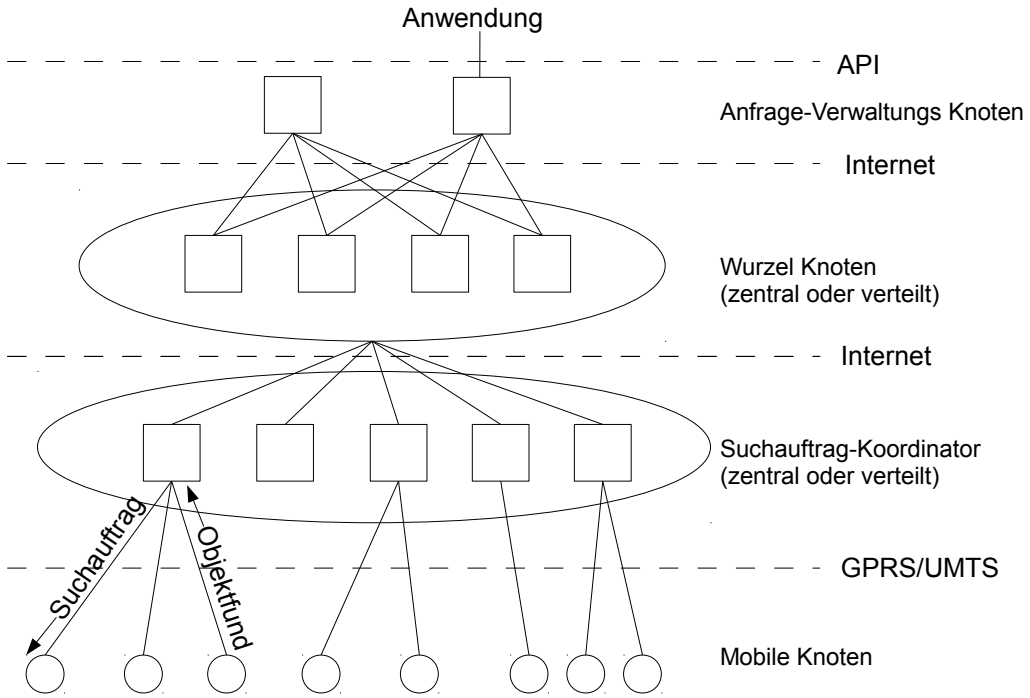

Abbildung 3.1: schematischer Aufbau des Systems

Funde des Objekts an den Anfragenden weitergeleitet bis die Dauer der Anfrage erreicht wurde und sie entfernt wird. Was wir genau unter einer ID eines mobilen Objekts verstehen, wird im Abschnitt über mobile Knoten näher erläutert.
Die Signatur des API-Aufrufs durch die Anwendung ist wie folgt definiert: *Anfrage(id, dauer, permanent)* Die genauen Vorgänge eines solchen Aufrufs werden in späteren Kapiteln beschrieben. Letztlich werden von den Koordinatoren Suchaufträge an die mobilen Knoten verteilt. Ein Suchauftrag besteht hierbei aus einer Nachricht, die eine Menge von Objekts-IDs enthält. Durch so eine Nachricht kann ein mobiler Knoten beauftragt werden den Fund eines bestimmten Objekts zu melden oder auch nicht mehr länger danach zu suchen. Falls ein mobiler Knoten ein passendes Objekt findet, so meldet er dies in einer Objektfund Nachricht an seinen zuständigen Koordinator. Dieser leitet die Nachricht über den Fund weiter an den, im verteilten Fall des Wurzel Knoten, für diese ID zuständigen Wurzel Knoten weiter. Von dort aus gelangt der Fund über einen Anfrage-Verwaltungs Knoten zur Anwendung.
Die Knoten zwischen der Anwendung und den mobilen Knoten bezeichnen wir allgemein als Infrastruktur-Knoten bzw. ihre Gesamtheit als Infrastruktur. Diese Infrastruktur bildet eine Abstraktionsebene zwischen Suchanfragen und mobilen Knoten. Die Anwendung muss nicht wissen, welche mobilen Knoten beauftragt werden ein Objekt zu suchen und genauso muss ein mobiler Knoten nicht wissen, wie er einen Objektfund an die Anwendung übermittelt.

3.1.2 Mobile Knoten

Als mobile Knoten eigenen sich solche Geräte, die über eine drahtlose Kommunikationsschnittstelle verfügen, ihre Position in Form einer geometrischen Koordinate (Breite, Länge), zum Beispiel mittels GPS, ermitteln können und einen Möglichkeit besitzen um mobile Objekte in ihrer unmittelbaren Umgebung zu erfassen. Da wir im weiteren Verlauf von kartesischen Koordinaten ausgehen werden, nehmen wir implizit eine Umrechnung der geometrischen Koordinaten in kartesische Koordinaten an.
Weiterhin unterliegen die Systeme zur Positionsbestimmung einer gewissen Ungenauigkeit. Daher definieren wir die Position P eines mobilen Knotens als Kreis, dessen Mittelpunkt M=(x,y) den durch das System zur Positionsbestimmung ermittelten, kartesischen Koordinaten entspricht und dessen Radius $r_{Abweichung}$ gleich der Ungenauigkeit ist. Somit lässt sich die Position eines mobilen Knotens durch ein Tupel P = (x,y,$r_{Abweichung}$) schreiben.
Wir gehen davon aus, dass jedes mobile Objekt einen global eindeutigen Bezeichner, fortan ID genannt, besitzt, welcher durch mobile Knoten ausgelesen werden kann und der Identifizierung des Objekts innerhalb des Systems dient. Zur Ermittlung dieser ID und damit der Erfassung eines mobilen Objekts, bietet sich ein RFID-Tag-Leser oder eine Bluetooth-Schnittstelle an. Mobile Objekte besitzen dann entweder einen RFID-Tag oder einen Bluetooth-Beacon. Bei einem Bluetooth-Beacon dient die MAC-Adresse als ID. Wir nehmen außerdem an, dass die maximale Entfernung r_{Leser}, bis zu der eine ID ausgelesen werden kann, sehr gering ist, sprich wenige Meter beträgt, da dies Auswirkungen auf die ermittelte Position des Objekts hat. Je kleiner die Reichweite, desto kleiner ist die Ungenauigkeit der Objektposition. Des Weiteren gibt es keine Beeinflussung der Bewegung der mobilen Knoten durch das System, sie bekommen lediglich Suchaufträge, jedoch beispielsweise keine Anweisung eine bestimmte Positiob anzusteuern. Daher ist das Auffinden und Verfolgen mobiler Objekte rein opportunistisch, das heißt es ist prinzipiell nur möglich ein Objekt zu finden, falls es sich in der Sensorreichweite eines sich vorbeibewegenden mobilen Knotens befindet. In unserem Modell gehen wir davon aus, dass als mobile Knoten beispielsweise Handys bzw. Smartphones dienen, welche Fußgänger von sich aus bei sich tragen und die bereit sind einen Teil der Resourcen ihres Mobiltelefons dem System zur Verfügung zu stellen. Neben dem zuvor erwähnten exemplarischen Fall ist im Allgemeinen jedoch jede Form von mobilen Geräten möglich, welche obige Anforderungen erfüllen. Zum Beispiel können Fahrzeuge mit entsprechenden Sensoren genauso am System teilnehmen. Wir gehen davon aus, dass zu jedem Zeitpunkt Kontakt zu einer Basisstation besteht, die für das Gebiet verantwortlich ist, in dem sich der mobile Knoten gerade befindet. Sie ermöglicht die Kommunikation zwischen dem mobilen Knoten und dem Suchauftrag-Koordinator.

3.1.3 Mobile Objekte

Mobile Objekte besitzen, wie im Kapitel über mobile Knoten beschrieben, eine global eindeutige Kennung, die wir als ID bezeichnen. Als Träger dieser ID bieten sich RFID-Tags oder Bluetooth-Beacons an. Bei passiven RFID-Tags muss seitens des Objekts keine

Energieversorgung zur Verfügung gestellt werden. Bei Bluetooth-Beacons oder aktiven RFID-Tags gehen wir von sehr einfachen Geräten aus, die nur sehr wenig Energie benötigen, so dass gewährleistet ist, dass permanent eine ID ausgelesen werden kann. Außerdem ist es so möglich eine sehrgroße Anzahl von Objekten kostengünstig mit einer solchen ID zu versehen. Die Position des Objekts kann im Allgemeinen beim Auslesen der ID nicht genau bestimmt werden. Die möglichen Positionen eines Objekts liegen in einem Kreis um die Position des auslesenden mobilen Knotens. Der Radius des Kreises entspricht der maximalen Entfernung r_{Leser} bis zu der eine ID ausgelesen werden kann. Hinzu kommt, die Ungenauigkeit der Positionsbestimmung des mobilen Knotens, die wir im Abschnitt über mobile Knoten beschrieben haben. Daher definieren wir die Position P_{Objekt} eines von einem mobilen Knoten MK_i, mit zugehöriger Position $P_{MK_i}=(x_{MK_i},y_{MK_i}, r_{Abweichung})$, ausgelesenen Objekts als Kreis mit Mittelpunkt $M=(x_{MK_i},y_{MK_i})$ und Radius $r=r_{Abweichung}+r_{Leser}$. Die Position eines mobilen Objekts ist dann wiederum als Tupel $P_{Objekt}=(x,y,r)$ darstellbar. Außerdem ist die maximale Geschwindigkeit über alle Objekte, die wir mit v_{max} bezeichnen, bekannt.

3.1.4 Anfrage-Verwaltungs Knoten

Diese Art von Knoten stellt der Anwendung die im Abschnitt 3.1.1 beschriebene Schnittstelle für Suchanfragen zur Verfügung und repräsentiert somit den Einstiegspunkt in das System aus Sicht der Anwendung. Beim Aufruf der API durch die Anwendung wird die dadurch entstehende Suchanfrage von diesem Knoten gespeichert. Wenn es bereits eine gespeicherte Suchanfrage für die im API-Aufruf spezifizierte ID gibt, so wird nichts unternommen, da schon vom System nach dem Objekt gesucht wird. Sonst wird der für die ID zuständige Wurzel Knoten benachrichtigt. Dadurch ist eine etwaige Verteilung des Wurzel Knotens aus Anwendungssicht transparent. Neben der Speicherung der Anfrage und Benachrichtigung eines Wurzelknotens, leitet dieser Knoten auch die Objektfunde weiter an die Anwendung, welche die Suchanfrage gestellt hat.

3.1.5 Wurzel Knoten

Ein Wurzel Knoten bildet den internen Einstiegspunkt des System. Er regelt die Weiterleitung der Suhaufträge an den Suchauftrag-Koordinator bzw. im verteilten Fall an die Koordinatoren. Dabei besteht ein Suchauftrag, im Gegensatz zu einer Suchanfrage, nur aus einer Menge von Objekt-IDs und wird systemintern verwendet. Für den Fall, dass der Suchauftrag-Koordinator verteilt ist, bestimmt der Wurzel Knoten an welche Koordinatoren ein Suchauftrag initial verteilt wird. Außerdem regelt er die Weiterleitung der Objektfunde an die Anfrage-Verwaltungs Knoten, welche eine Suchanfrage für das jeweils gefundene Objekt gespeichert haben.
Der Wurzel Knoten kann entweder zentral oder verteilt sein. Im verteilten Fall ist ein Wurzel Knoten für jeweils einen bestimmten Bereich von Objekt-IDs zuständig.

3.1.6 Suchauftrag-Koordinator

Dieser Teil des Systems koordiniert die Verteilung der Suchaufträge an die mobilenKnoten. Die Kommunikation mit den mobilen Knoten erfolgt über das Mobilfunnetz. Dazu benutz ein Suchauftrag-Koordinator Knoten eine oder auch mehrerer Basisstationen. Konzeptionell kann man sich Basisstation und Koordinator als am gleichen Ort befindlich vorstellen und die Suchaufträge werden entsprechend in dem Gebiet verteilt, welches die Basisstation abdeckt.
Neben einem einzelnen, zentralen Koordinator ist es ebenso möglich diesen auf mehrere Knoten zu verteilen. Beim verteilten Fall kommunizieren die Knoten untereinander um die Verteilung der Suchaufträge zu koordinieren. Die von den mobilen Knoten empfangenen Nachrichten über gefundene Objekte werden zum für die jeweilige Objekt-ID zuständigen Wurzel Knoten weitergeleitet.

3.2 Problembeschreibung

In dieser Arbeit sollen Strukturen und Algorithmen für die im Systemmodell beschriebenen Infrastruktur-Knoten entwickelt und analysiert werden, so dass diese die geforderte Abstraktionsebene bereitstellen. Dabei liegen die Schwerpunkte auf der Skalierbarkeit des gesamten Systems, womit wir meinen, dass wir die über die Funkschnittstelle übertragenene Daten so weit wie möglich minimieren wollen (Effizienz),jedoch ohne zu Lasten der Effektivität bezüglich der gefundenen Objekten durch die mobilen Knoten

Von außen betrachtet stellt sich das Problem so dar, dass mittels mobiler Knoten die Positionen mobiler Objekte einmalig oder kontinuierlich ermittelt und an den Anfragenden gemeldet werden sollen. Die Schwierigkeit besteht nun darin, dass nur ein kleiner Teil aller mobilen Objekte zu einem bestimmten Zeitpunkt gesucht wird und es daher nahe liegt Suchaufträge für bestimmte Objekte an die mobilen Knoten zu verteilen. Die Hauptaufgabe des Systems ist es die Verteilung dieser Aufträge unter den genannten Schwerpunkten zu realisieren und zu definieren wie die einzelnen Komponenten interagieren müssen, im die angestrebte Minimierung der Nachrichtenlast zwischen Infrastruktur und mobilen Knoten zu erreichen.
Unter Skalierbarkeit verstehen wir, dass tausende mobiler Knoten unterstütz werden können und dass das Gebiet in dem Objekte gesucht werden können prinzipiell beliebig groß sein kann. Als Grundlage für die Effizienz des Systems betrachten wir die über die Funkschnittstelle, d.h. zwischen Koordinator und mobilen Knoten und umgekehrt, übertragenen Daten. Dabei versuchen wir insbesondere die Menge der Daten von Koordinatoren zu mobilen Knoten zu minimieren und dies gleichzeitig ohne mögliche Objektfunde zu verlieren. Des Weiteren fordern wir, dass wenn die Position eines Objekts von einem mobilen Knoten gemeldet wird, dieses auch tatsächlich vom betreffenden Knoten gefunden wurde, d. h. wir lasse diesbezüglich keine Falschmeldungen oder unpräzise Meldungen zu. Es muss eindeutig sein welches Objekt wo gefunden wurde.
Plakativ ausgedrück wollen wir maximale Effektivität mit minimalem Aufwand bezüglich der Kommunikation erreichen.

4 Systementwurf

In diesem Kapitel werden zunächst grundlegenden Überlegungen zum Entwurf des Systems auf Basis des in Kapitel 3.1 beschriebenen Systemmodells und den in Kapitel 3.2 genannten Anforderungen dargelegt. Aufbauend darauf werden mögliche Strategien und Strukturen für das System entworfen und detailliert beschrieben.

4.1 Grundlegendes

Aufgrund der Annahme, dass nur ein kleiner Teil aller Objekte gesucht wird, liegt es nahe Suchaufträge für Objekte an die mobilen Knoten zu verteilen, woraufhin diese nur die Objekte melden für die ein Suchauftrag vorliegt. Würde jeder mobile Knoten alle Objekte, die er findet, melden, so würde er zu viele Objekte finden, die nicht gesucht werden. Diese würde er alle melden, was zu einer zu großen Menge an unngewollten Daten führen würde. Insgesamt wäre dies zwar maximal effektiv, was die Objektpositionen der gesuchten Objekte angeht, jedoch gleichzeitig im Bereich von maximal ineffizient. Unser Ziel ist daher die Verteilung der Suchaufträge so effizient wie möglich zu gestalten, das heißt sie nur an die mobilen Knoten zu verteilen, die das Objekt auch finden können und dabei keine Nachrichten über Objektfunde zu verlieren. Eine weitere Möglichkeit die Effizienz zu erhöhen ohne die Effektivität zu ernidrigen bietet die Kompression von Nachrichten.

4.1.1 Selektive Verteilung von Suchanfragen

Eine wichtige Überlegung ist die Tatsache, dass nicht alle mobilen Knoten alle IDs der gesuchten Objekte kennen müssen, sondern im Grunde nur die IDs von Objekten welche sie, bezogen auf die Mobilität, überhaupt finden können. Ein Suchauftrag für ein Objekt welchem ein mobiler Knoten nicht begegnet ist letztlich eine Menge von unnötig gesendeten Daten.
Es ist klar, dass sich ein Objekt generell und insbesondere nach einem Fund nur in einem bestimmten Bereich befinden kann, unter der Annahme, dass es eine maximale Geschwindigkeit v_{max} für mobile Objekte gibt. Um diesen Sachverhalt auszunutzen ist der naheliegendste Schritt das Gebiet in dem der Dienst zur Verfügung gestellt wird in kleinere Bereiche, von nun an Zelle genannt, aufzuteilen. Jeder Zelle ist ein Suchauftrags-Koordinator zugeordnet, welcher die Aufträge verteilt und die Funde weiterleitet. Als geometrische Struktur bietet sich eine Aufteilung in Quadrate, oder allgemein Rechtecke, gleicher Größe an.

Um die Suchaufträge für Objekte selektiv in den einzelnen Zellen verteilen zu können, ist es nötig die den Bereich in dem sich ein mobiles Objekt aufhalten abzuschätzen und anschließend wird dieser Bereich auf die Zellen abgebildet. Man kann nicht davon ausgehen, dass ein momentan gesuchtes Objekt zu jedem Zeitpunkt von einem mobilen Knoten gefunden wird und somit ist seine Position unter Umständen für längere Zeit nicht bekannt. Für die Abschätzung verwenden wir daher eine vereinfachte Variante der Koppelnavigation (engl. dead reckoning). Diese Variante bezieht nicht die Bewegungsrichtung des Objekts mit ein.
Voraussetzung ist mindestens eine bekannte Position des Objekts zu einem Zeitpunkt t und die Kenntnis seiner momentanen bzw. in unserem Fall seiner maximalen Geschwindigkeit. Die Menge möglicher Positionen eines mobilen Objekts ist nun eine über die Zeit stetig mit der Geschwindigkeit des Objekts wachsende Kreisfläche, deren Mittelpunkt die letzte bekannte Position des Objekts zu einem Zeitpunkt t ist. In Abbildung 4.1 ist dies dargestellt. Zum Zeitpunkt t_1 wird das Objekt von einem mobilen Knoten gefunden,

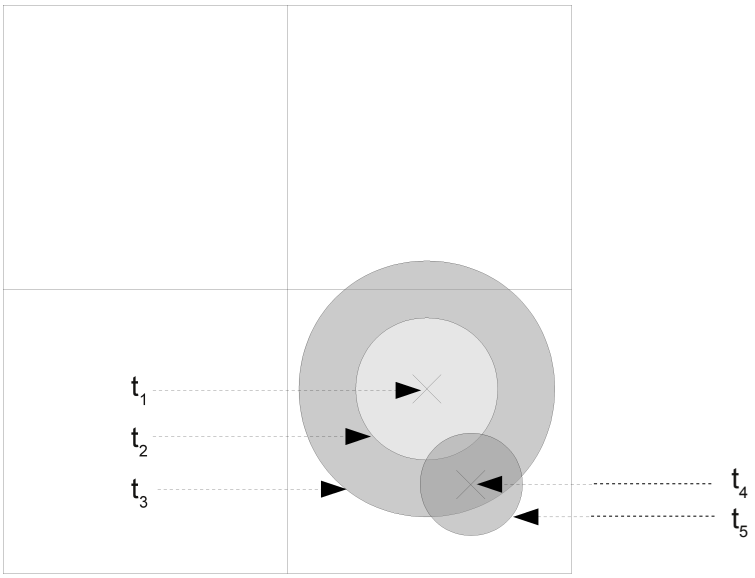

Abbildung 4.1: Abschätzung der Objektposition, dead reckoning

der Kreis hat den Radius 0. Zwischen t_1 und t_2 wird das Objekt nicht gefunden und so wächst der Radius um $(t_2 - t_1) \cdot v_{Objekt}$. Analog gilt dies für die Zeit zwischen t_2 und t_3. Zuätzlich wird beim Zeitpunkt t_3 eine der Grenzen der Zelle überschritten und je nach Infrastruktur hat dies entsprechende Auswirkungen. Falls an die überschrittene Grenze wie in diesem Fall eine weitere Zelle angrenzt, so wäre eine offensichtliche Maßnahme auch in dieser Nachbarzelle nach dem Objekt zu suchen. Zum Zeitpunkt t_4 wird das Objekt erneut gefunden. Daraus ergibt sich ein neuer Mittelpunkt für den Kreis. Falls um Zeitpunkt t_3 die Nachbarzelle informiert wurde, so wäre eine mögliche Konsequenz nach dem erneuten Fund nun nicht mehr dort nach dem Objekt zu suchen. Zum Zeitpnukt t_5 ist das mögliche Aufenthaltsgebiet des Objekts auf einen Kreis mit Radius $(t_5 - t_4) \cdot v_{Objekt}$

gewachsen.

4.1.2 Komprimierung von Daten

Neben der Entscheidung welche Objekte in welcher Zelle gesucht werden, gibt es noch einen weiteren Faktor um die übertragenen Datenmengen zu reduzieren, nämlich die Datenpakete selbst. Hierbei lassen sich zunächst zwei Richtungen unterscheiden für die verschiedene Bedingungen gelten. Zum einen von den Koordinatoren zu den mobilen Knoten und zum anderen die Rückrichtung.
Wenn wir die Richtung Koordinator zu mobilen Knoten betrachten, so ist es prinzipiell legitim, wenn der Suchauftrag, den ein mobiler Knoten enthält, eine größere Menge an IDs von Objekten enthält als in Wirklichkeit gerade in dieser Zelle gesucht werden. Dies hat zur Folge, dass unnötige Objektfunde gemeldet werden, jedoch lassen sich diese, wie wir sehen werden, begrenzen. Dies lohnt sich nur, wenn die Summe der Daten der unnötig gesendeten Objektfunde die eingesparte Datenmenge bei den gesendeten Suchaufträgen nicht übersteigt. Eine Datenstruktur welche die geforderten Eigenschaften besitzt ist ein sogennanter Bloom-Filter([Blo70],[BM02]). Es handelt sich hierbei um eine probabilistische Datenstruktur, welche im allgemeinen eine Obermenge der tatsächlichen Menge repräsentiert.
In Rückrichtung, also von den mobilen Knoten zum Koordinator haben wir die Einschränkung, dass der Bericht eines Funds kein Objekt liefern darf, welches nicht auch tatsächlich gefunden wurde. Daher ist es nicht einfach hierfür eine probabilistische Datenstruktur wie Bloom-Filter zu verwenden.

4.2 Entwurf von Ansätzen

Die zuvor vorgestellten Überlegungen werden wir in den folgenden Kapitel in funktionsfähige Algorithmen zur Verteilung von Suchaufträgen umsetzen und so zu einem vollständigen System gelangen. Dabei beginnen wir mit einem maximal effektiven, jedoch sehr ineffizienten Ansatz, der bei der Evaluierung als Vergleich dient. Danach analysieren wir das Prinzip der selektiven Suchauftragsverteilung und Enden bei einer dynamischen Suchbereichsanpassung. Die Kompression von Daten hat sich, bis auf die verlustlose Kompression, die man immer anwenden kann nicht als vielversprechend gezeigt. Wir diskutieren jedoch kurz am Ende wie man Bloom-Filter nutzen könnte.

4.3 Basis-Ansatz

Bei diesem Ansatz wird das Gebiet in dem nach mobilen Objekten gesucht wird nicht unterteilt. Es gibt genau einen Knoten K_0 welcher für das komplette Dienstgebiet verantwortlich ist. K_0 dient bei diesem Ansatz als Anfrage-Verwaltungs Knoten, Wurzel Knoten und Suchauftrag-Koordinator. Alle Suchanfragen werden an K_0 gestellt, welcher

diese dann an alle mobilen Knoten weiterleitet (siehe Abbildung 4.2). Da K_0 auch als Wurzel Knoten fungiert und er der zentrale Knoten zur Verteiung der Suchaufträge an die mobilen Knoten ist, bezeichnen wir ihn im Weiteren bei diesem Ansatz auch einfach nur als Wurzel. Wie in Kapitel 3.1.1 beschrieben setzt sich eine Suchanfrage zusammen aus

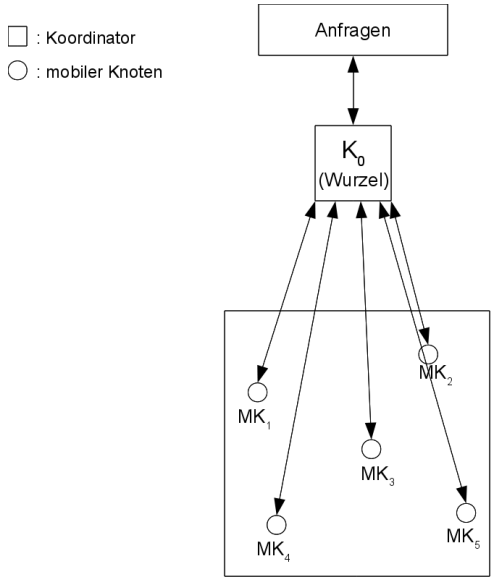

Abbildung 4.2: Architektur Basis-Ansatz

der ID des gesuchten Objekts, der Dauer der Suchanfrage und der Angabe ob dauerhaft während des angegebenen Zeitraums nach dem Objekt gesucht werden soll. Falls nicht dauerhaft gesucht werden soll, wird nur die nächste gefundene Position des Objekts gemeldet und danach die Anfrage gelöscht. Um jederzeit zu wissen welche Objekte gesucht werden speichert K_0 zum einen eine Liste $L_{Anfragen_{K_0}}$ mit allen aktuellen Anfragen und eine Liste $L_{ObjektID_{K_0}}$ mit den IDs der aktuell gesuchten Objekte. Analog speichert sich jeder mobile Knoten MK_i eine Liste $L_{ObjektID_{MK_i}}$. Diese Liste vergleicht MK_i periodisch mit der Liste $L_{ObjektIDgefunden_{MK_i}}$ der IDs von Objekten, die er in seiner Umgebung finden kann und schickt dann $L_{Fund_{MK_i}}$=$L_{\ ObjektID_{MK_i}} \cap L_{ObjektIDgefunden_{MK_i}}$ als Nachricht über gefundene Objekte an K_0. Diese Nachrichten werden dann an die Anwendung weitergeleitet. Eine Nachricht über den Fund eines Objektes enthält eine Auflistung mehrer Einträge der Form <Position X, Position Y, Zeitstempel, Liste der Objekt-IDs>, wobei die Positionen den GPS-Koordinaten des mobilen Knoten entsprechen, welcher das Objekte gefunden hat und das Feld Zeitstempel entspricht dem Zeitpunkt zu dem die Objekt gefunden wurden. Ob eine Nachricht über einen Fund eine oder mehrere Einträge der oben beschriebenen Form enthält, hängt davon ab, ob das Suchen nach Objekten und die Meldung dieser gekoppelt ist. Ist es nicht gekoppelt, so wird nach jedem Suchvorgang die Position des mobilen Knotens, der Zeitpunkt der Suche und die Liste der gefundenen Objekt-IDs, also ein Eintrag obiger Form, zwischengespeichert. Bei der nächsten Benachrichtigung über eienn Fund werden dann

die bis dahin gespeicherten Einträge an K_0 übermittelt. Bei diesem Ansatz gehen wir, zugunsten einer möglichst niedrigen Antwortzeit auf eine Anfrage, von einer Kopplung aus, sprich ein mobiler Knoten meldet direkt nach der Suche nach Objekten wie oben beschrieben die passenden IDs an K_0. Daher besteht die Meldung über eine Fund aus der Position des mobilen Knoten, dem Zeitstempel und der Liste der gefundenen IDs.

Um das Verhalten der Komponenten näher zu beschreiben, unterscheiden wir zunächst die Interaktionen zwischen ihnen. Die Kommunikation zwischen der Anwendung und der Wurzel (K_0) ist als einfacher API-Aufruf modelliert bei dem die vorher genannenten Parameter übergeben werden. Die Weiterleitung der Funde gestaltet sich als einfacher Event bzw. Nachricht welche an die Anwendung gesendet wird. Zwischen der Wurzel und den mobilen Knoten wird mittels Nachrichten verschiedenen Typs kommuniziert. Bei diesem Ansatz verwenden wir vier unterschiedlichen Typen von Nachrichten. Diese sind in Abbildung 4.3 dargestellt.

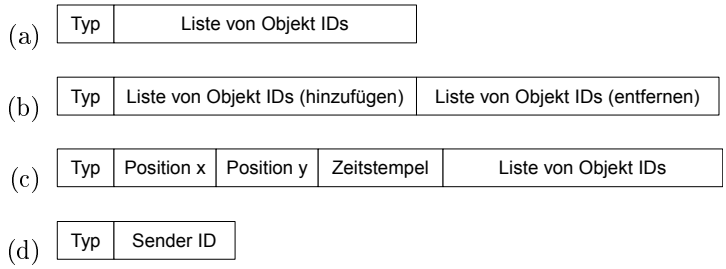

Abbildung 4.3: Unterschiedliche Typen von Nachrichten für den Basis-Ansatz
(a) Nachricht vom Typ AUFTRAGS_LISTE
(b) Nachricht vom Typ AUFTRAGS_LISTE_DELTA
(c) Nachricht vom Typ OBJEKT_FUND
(d) Nachricht vom Typ ANFRAGE_AUFTRAGS_LISTE

Das Typ-Feld wird dazu benutzt um die verschiedenen Nutzdaten zu unterscheiden. Beim Typ AUFTRAGS_LISTE handelt es sich bei den Nutzdaten um eine bloße Aufzählung der IDs der gesuchten Objekte. Diese Art von Nachricht wird von der Wurzel an die mobilen Knoten geschickt um die komplette Liste der gesuchten Objekt-IDs zu verteilen. Ähnlich dazu ist die Nachricht vom Typ AUFTRAGS_LISTE_DELTA. Diese reflektiert die Änderung der Liste der gesuchten IDs und enthält demnach zwei Listen mit Objekt-IDs. In der einen sind die IDs die neu hinzugefügt wurden und die andere enthält analog die entfernten IDs. Die Nachricht vom Typ OBJEKT_FUND wird dazu benutzt um die Position einer oder mehrerer gefundener Objekte an K_0 zu übermitteln. Des Weiteren benutzt ein mobiler Knoten beim Eintritt in das System eine Nachricht vom Typ ANFRAGE_AUFTRAGS_LISTE um die komplette Liste der momentan gesuchten IDs zu erhalten.

Im Folgenden werden nun die Reaktionen auf die verschiedenen Nachrichten und

Ereignisse genauer beschrieben.

Suchanfrage

Erhält die Wurzel eine neue Suchanfrage für ein Objekt o_i und wird o_i in diesem Moment nicht schon gesucht, dann wird eine Nachricht vom Typ AUFTRAGS_LISTE_DELTA an alle mobilen Knoten geschickt, deren Inhalt die ID von o_i ist. Die mobilen Knoten wiederum fügen beim Erhalt dieser Nachricht, welche einem Suchauftrag entspricht, die darin enthaltenen IDs zu ihrer lokalen Liste $L_{ObjektID}$ hinzu und melden als unmittelbare Folge von nun an jeden Fund des Objekts o_i mittels einer Nachricht vom Typ OBJEKT_FUND an die Wurzel (siehe Algorithmus 1 und 2).

Algorithmus 1 Basis Ansatz: K_0 bekommt neue Suchanfrage

procedure $Anfrage(id, dauer, permanent)$

```
1: if ∀e ∈ L_Anfragen : e.id ≠ id then
2:    L_Anfragen := L_Anfragen ∪ {(generiereAnfrageID(), id, dauer, permanent)};
3:    L_Objekt_ID := L_Objekt_ID ∪ {id};
4:    nachricht.typ := AUFTRAGS_LISTE_DELTA;
5:    nachricht.fügeHinzu(id);
      //sende neuen Suchauftrag an alle mobilen Knoten
6:    BroadcastAnMobileKnoten(nachricht);
7: end if
```

end procedure

Algorithmus 2 Basis Ansatz: Mobiler Knoten bekommt eine Nachricht

procedure $Empfange(nachricht)$

```
1: if nachricht.typ == AUFTRAGS_LISTE_DELTA then // falls bisher keine kom-
   plette Liste von IDs empfangen wurde -> verwerfen
2:    if L_ObjektID aktuell then
3:       L_ObjektID := L_ObjektID ∪ nachricht.IDsHinzufügen(); // hinzuzufügende IDs
4:       L_ObjektID := L_ObjektID \ nachricht.IDsEntfernen(); // zu entfernende IDs
5:    end if
6: else if nachricht.typ == AUFTRAGS_LISTE then
7:    L_ObjektID := nachricht.IDListe;
8: end if
```

end procedure

Eintritt eines mobilen Knotens in das System

Falls ein mobiler Knoten neu oder wieder in das System kommt, dann fragt er zunächst die Liste der IDs aller zu diesem Zeitpunkt gesuchten Objekte bei K_0 mittels einer Nachricht vom Typ ANFRAGE_AUFTRAGS_LISTE ab. Als Antwort schickt K_0 die komplette

Algorithmus 3 Basis Ansatz: K_0 bekommt eine Nachricht

procedure *Empfange*(*nachricht*)

```
1: if nachricht.typ == ANFRAGE_AUFTRAGS_LISTE then
2:    antwort.typ := AUFTRAGS_LISTE;
3:    antwort.IDListe := L_ObjektID;
4:    UnicastAnMobilenKnoten(nachricht.SenderID, antwort);
5: else if nachricht.typ == OBJEKT_FUND then
6:    for all e ∈ nachricht.IDListe do
7:       leite nachricht.PositionX, nachricht.PositionY an Anfragenden weiter;
8:    end for
9: end if
```

end procedure

Liste an den mobilen Knoten in einer Nachricht vom Typ AUFTRAGS_LISTE (siehe Algorithmus 3 und 2). Nach seinem Eintritt oder Wiedereintritt in das System verwirft ein Knoten alle Nachrichten vom Typ AUFTRAGS_LISTE_DELTA, solange er keine vollständige Auftragsliste erhalten hat (siehe Algorithmus 2).

Objektfund

Erhält K_0 eine Nachricht über einen Objektfund von einem mobilen Knoten, dann werden die in der Nachricht enthaltenen Koordinaten an die jeweiligen Anfragenden weitergeleitet (siehe Algorithmus 3). Eine weitergehende Verarbeitung bzw. Interpretation der Daten eines Objektsfunds durch das System ist bei diesem Ansatz, im Gegensatz zu den anderen Ansätzen, nicht nötig.

Periodisch ausgeführte Aktionen

Um den Ansatz zu vervollständigen, wird von der Wurzel K_0 und von den mobilen Knoten zusätzlich periodisch folgendes durchgeführt:
Die Wurzel überprüft bei jeder momentan gestellten Suchanfrage, ob der darin spezifizierter Zeitraum vorüber ist und falls ja, wird die Anfrage aus der Liste entfernt. Zusätzlich wird überprüft, ob nach dem Entfernen der Anfrage e noch eine weitere Anfrage e' existiert, so dass e.id = e'.id, sprich es wird getestet ob es immer noch eine Suchanfrage für das Objekt gibt. Wenn dem so ist, muss nichts weiter unternommen werden, da die mobilen Knoten immer noch nach dem Objekt suchen. Andernfalls werden die mobilen Knoten mittels einer Nachricht vom Typ AUFTRAGS_LISTE_DELTA darüber informiert, die Suche nach dem Objekt einzustellen, um so unnötige Nachrichten zu vermeiden.
Die mobilen Knoten suchen, wie schon weiter oben beschrieben, periodisch nach Objekten in ihrer Umgebung und vergleichen deren IDs mit ihrer lokalen Liste $L_{ObjektID}$ der gesuchten IDs. Falls eine oder mehrere gesuchte Objekte dabei sind, sendet der mobile Knoten seine ihm aktuell bekannte Position zusammen mit der Liste der IDs in einer Nachricht vom Typ OBJEKT_FUND an die Wurzel. Weiterhin führt der Knoten in

einem bestimmten Intervall eine Positionsbestimmung durch, was beispielsweise durch GPS geschehen kann. Dies ist jedoch entkoppelt von der Suche nach mobilen Objekten und daher ist nicht gewährleistet, dass die Koordinaten, welche der mobile Knoten an die Wurzel übermittelt, aktuell sind. Die mögliche Position eines so gemeldeten mobilen Objekts ergibt sich daher wie in Kapitel 3.1.3 beschrieben zu einem Kreis, welcher die Ungenauigkeit der Positionsbestimmung des mobilen Knotens und die Reichweite des Lesers zum Auslesen der ID des mobilen Objekts berücksichtigt.

4.3.1 Zusammenfassung

Dieser Ansatz stellt die einfachste Variante dar um mobile Objekte mittels der Verteilung von Suchaufträgen an die mobilen Knoten zu suchen. Da jeder beauftragt wird nach dem jeweiligen Objekt zu suchen ist gewährleistet, dass es, wenn überhaupt möglich, gefunden wird und dass die Anzahl der gemeldeten Funde maximal ist. Jedoch wird außer Acht gelassen, dass nicht jeder mobile Knoten nach einem bestimmten Objekt suchen muss, da es sich nur in einem bestimmten Bereich bewegt und von einer gewissen Teilmenge aller mobilen Knoten nie gefunden werden kann. Dies werden wir nun im weiteren näher verfolgen um den Ansatz zu verbessern und unser Ziel die zwischen Suchanfrage-Koordinator bzw. Wurzel und mobilen Knoten übertragenen Daten bei möglichst hoher Rate an gefundenen Objekten zu minimieren, zu erreichen.

4.4 Erweiterung Basis-Ansatz

Aufbauend auf dem Basis-Ansatz werden wir nun die Tatsache ausnutzen, dass nicht jeder mobile Knoten zwingend nach allen zu einem Zeitpunkt gesuchten mobilen Objekten suchen muss, was bisher beim Basis-Ansatz der Fall war. Dabei ist die Grundidee die Suchaufträge an die mobilen Knoten räumlich eingeschränkt zu verteilen (selektive Verteilung von Suchaufträgen) und den Bereich in dem nach einem Objekt gesucht wird kontinuierlich anzupassen (dynamische Suchbereichsanpassung). Dadurch bekommen nun nur die mobilen Knoten einen Suchauftrag, in deren Nähe sich das gesuchte Objekt aufhalten kann und die, aus Sicht des Systems, die Möglichkeit haben das Objekt zu finden. Dabei versuchen wir jedoch die Anzahl der Objektfunde maximal zu halten und gleichzeitig die gesendeten Nachrichten an die mobilen Knoten zu reduzieren. Um dies zu erreichen, unterteilen wir das Gebiet in dem nach Objekten gesucht werden kann in mehrere Zellen und schätzen zudem die möglichen Positionen eines Objekts ab. Den durch die Positionsabschätzung erhaltenen Bereich bilden wir auf die Zellen ab und erhalten so unseren effektiven Suchbereich. Die Suchauträge werden dann nur in den dem Suchbereich entsprechenden Zellen verteilt. Je nach Granularität der Zellen und Abbildung des möglichen Aufenthaltsbereichs auf die Zellen, ist das resultierende Suchgebiet gleich bzw. in den meisten Fällen größer als die ursprüngliche Positionsabschätzung für das Objekt.

4.4.1 Hierarchischer Ansatz

Bei diesem Ansatz verbinden wir die Suchauftrag-Koordinatoren so untereinander, dass eine Baumstruktur entsteht. Dazu unterteilen wir das Gebiet rekursiv in jeweils 4 identische Bereiche und jeden so entstandenen Bereich unterteilen wir wiederum. Dabei ist ein Bereich B als ein Tupel (x_{min}, x_{max}, y_{min}, y_{max}) definiert, welches die Grenzen des Bereichs festlegt. Die rekursive Unterteilung geschieht so lange bis eine vorgegebene Anzahl von Rekursionsstufen erreicht wurde. (siehe Abbildung 4.4)
Jedem auf diese Art entstehenden Bereiche wird ein Suchauftrag-Koordinator zugewiesen. Die Koordinatoren sind entsprechende der rekursiven Unterteilung untereinander verbunden und bilden insgesamt einen Baum dessen Wurzel K_0 ist. Daraus ergibt sich, dass das Gebiet B_K des Vaterknotens K' das Gebiet B_K des Kindknotens vollständig enthält. Da jeder Knoten 0 oder 4 Kinder hat, bezeichnet man diese Struktur auch als Quadtree (siehe Abbildung 4.5).
Der Knoten K_0, der die Wurzel des Quadtrees darstellt, ist für das komplette Gebiet verantwortlich. Er bildet Stufe 0 der Hierarchie und ist mit $K_{1,1}$ bis $K_{1,4}$ verbunden, welche den Bereich von K_0 in 4 gleich große Bereiche aufteilen. Da K_0 bei diesem Ansatz Ansatz analog zum Basis-Ansatz als Anfrage-Verwaltungs Knoten, Wurzel Knoten und Suchauftrag-Koordinator fungiert, bezeichen wir ihn auch nur als Wurzel.
Die restlichen Knoten im Baum sind reine Suchanfrage-Koordinatoren und sind wie beschrieben untereinander verbunden. Die Knoten auf den Hierarchiestufen zwischen Wurzel und unterster Hierarchiestufe kommunizieren ausschließlich mit der Wurzel oder anderen Suchanfrage-Koordinatoren.

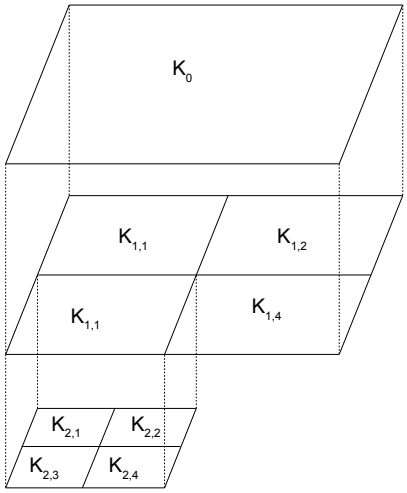

Abbildung 4.4: Unterteilung des Gebiets mit Rekursionstiefe 2

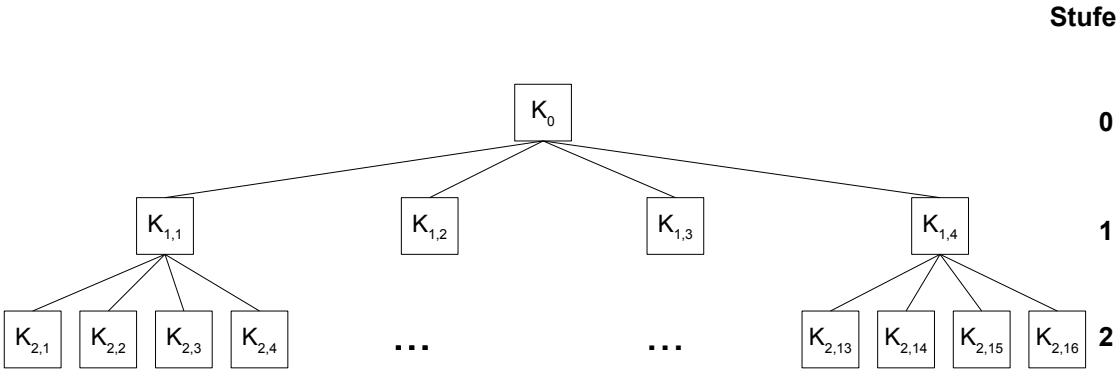

Abbildung 4.5: Verbindung der Koordinator Knoten innerhalb der Hierarchie

Allgemein befinden sich auf Tiefe i 4^i Knoten. Daraus ergibt sich, dass sich auf Blattebene $4^{\text{Baumtiefe}}$ Knoten befinden. Wir nehmen an, dass die Blattknoten mittels einer oder auch mehreren Basisstationen mit allen mobilen Knoten in ihrem Bereich, den wir bei diesen Knoten auch als Zelle bezeichnen, kommunizieren können. Die zugehörigen Suchanfrage-Koordinatoren der untersten Hierarchiestufe nennen wir daher auch Zell-Knoten.
Wie beim Basis-Ansatz besitzt auch hier jeder Zell-Knoten jeweils eine lokale Liste L_{ObjektID} mit den IDs der Objekte, die in seiner Zelle gesucht und an die mobilen Knoten in Form von Suchaufträgen verteilt werden. Diese Listen werden durch die Interaktion der Koordinatoren so angepasst, dass ein Objekt nur in den Zellen gesucht wird, in denen es sich aus Sicht des Systems befinden kann. Dabei wird die letzte bekannte Position des Objekts benutzt, um dies abzuschätzen.

Grundidee dynamische Suchbereichsanpassung

Wir werde nur den Algorithmus für die dynamische Suchbereichsanpassung bezüglich des hierarchischen Ansatz grobgranular und schrittweise für verschiedene Ereignisse beschreiben. Dabei liegt der Schwerpunkt darauf wie sich die Abschätzung der Objektposition auf den Suchbereich auswirkt und wie eine Veränderung des Suchbereichs zwischen den Knoten kommuniziert wird.
Als Ausgangspunkt nehmen wir den in Abbildung 4.6 gezeigten Baum.

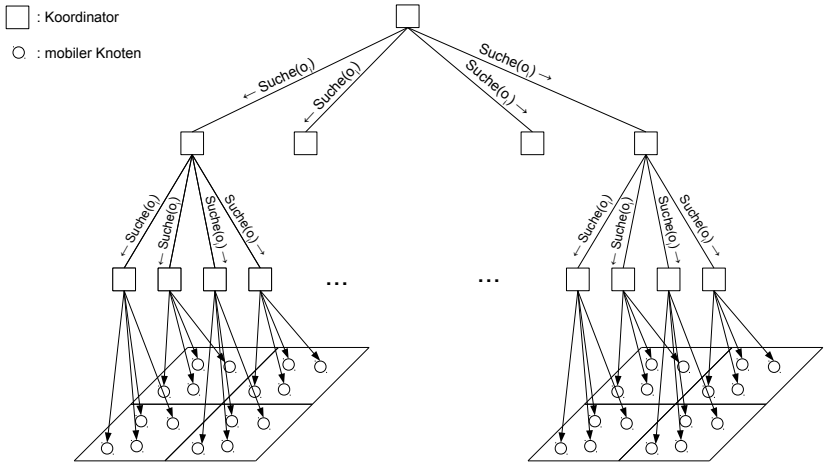

Abbildung 4.6: Verteilung Suchanfrage für Objekt o_i, welches zuvor noch nicht gefunden wurde

Auf Blattebene ergibt sich eine Aufteilung des Dienstgebiets in allgemein n Zellen. Eine Zelle entspricht dem kleinstmöglichen Suchbereich eines Objekts, da das Suchen eines Objekts in einer bestimmten Zelle gleichbedeutend damit ist jeden mobilen Knoten, der sich innerhalb der Zelle befindet, zu beauftragen das Objekt zu suchen.
Wir unterscheiden nun verschiedene Ereignisse und beschreiben die Reaktion des Systems darauf. Eine Suchanfrage für ein Objekt, das bisher nicht gefunden wurde oder dessen geschätzte Position größer oder gleich dem Dienstgebiet ist, bewirkt, dass an alle Suchauftrag-Koordinatoren ein Suchauftrag für dieses Objekt geschickt wird. (siehe Abbildung 4.6). Genauer wird dies vom Wurzel Knoten initiiert und jeder Knoten leitet die Nachricht an alle Kindknoten weiter. Empfängt ein Blattknoten diese Nachricht, so speichert er sich die Objekt-Id und weiterhin wird der Suchauftrag an alle mobilen Knonten in seiner Zelle verteilt.

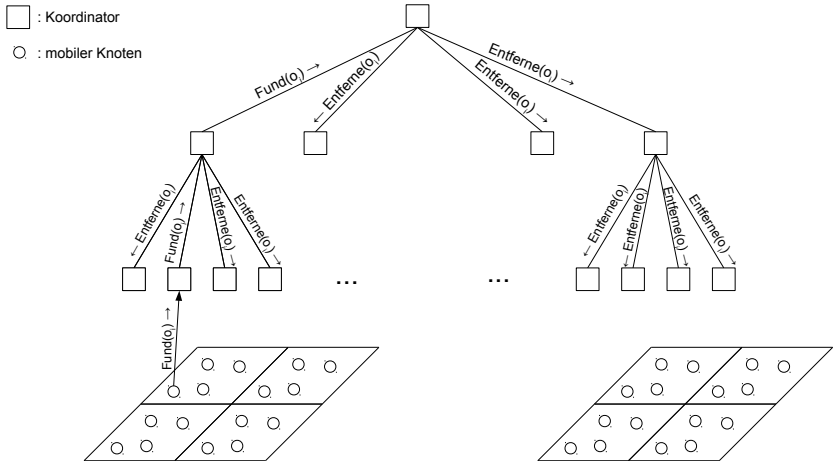

Abbildung 4.7: Fund von Objekt o_i und Anpassung des Suchbereichs

Bekommt ein Blattknoten eine Nachricht über einen Objektfund (siehe Abbildung 4.7 von einem mobilen Knoten, dann berechnet er ausgehend von der gemeldeten Position den Zeitpunkt zu dem das Objekt auch von mobilen Knoten in angrenzenden Zellen gefunden werden kann. Falls keine Objektfund Nachricht mehr eintrifft, so wird ein interner Timeout ausgelöst, welchen wir weiter unten beschreiben. Die Objektfund Nachricht wird nun an den Vaterknoten weitergeleitet, welcher an alle Kinknoten, außer von dem er die Nachricht bekommen hat, eine Nachricht zum Entfernen dieser Objekt-ID schickt. Außerdem merkt sich der Vaterknoten bei welchem Kind das Objekt zuletzt gefunden wurde und löscht eventuell vorhandene Timeout Einträge für das Objekt. Danach reicht der Vaterknoten die Objektfund Nachricht wiederum an seinen Vaterknoten weiter, welcher darauf analog reagiert. Jede Nachricht zum Entfernen einer Objekt-ID wird bis zu einem Blattknoten weitergeleitet. Dort angekommen löscht der Knoten die Objekt-ID aus seiner lokalen Liste, was zur Folge hat, dass neu hinzukommende mobile Knoten nicht beauftragt werden nach dem Objekt zu suchen. Mobile Knoten,die bereits nach dem Objekt suchen müssen nicht benachrichtig werden damit aufzuhören, dies wäre im Gegenteil sogar kontraproduktiv, da ja nach wie vor eine Suchanfrage für das Objekt existiert. Aus Sicht des Systems ist der Suchbereich für das Objekt nun nur noch die Zelle in der, so nehmen wir an, die Position des Objekts bzw. deren Abschätzung liegt.
Bekommt ein Knoten eine interne Benachrichtigung über einen Timeout (siehe Abbildung 4.8, so reicht er diese Nachricht an seinen Vaterknoten weiter. Dieser schickt an alle anderen Kindknoten eine Benachrichtigung nun auch nach dem Objekt zu suchen. Außerdem prüft der Vaterknoten, ob die abgeschätzte Position des Objekts eine der Grenzen des ihm zugehörigen Bereichs überschreitet. Ist dies der Fall, dann reicht er die Timeout Nachricht wiederum an seinen Vaterknoten weiter, der darauf analog wie gerade beschrieben reagiert. Im schlechtesten Fall wird die Wurzel erreicht und in der Folge im gesamten Dienstgebiet nach dem Objekt gesucht. Falls das Objekts noch innerhalb des Bereichs ist, berechnet der Vaterknoten den Zeitpunkt zu dem eine seiner Grenzen

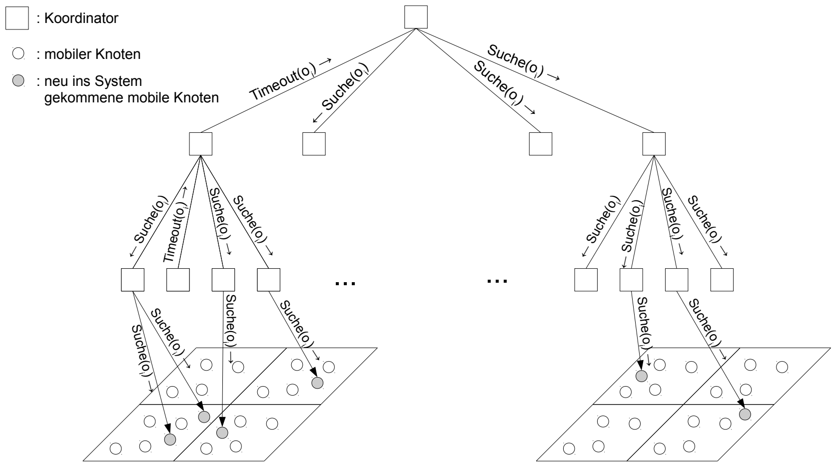

Abbildung 4.8: Timeout Benachrichtigung für Objekt o_i und Anpassung des Suchbereichs

überschritten wird und verwaltet dies wieder als Timeout für dieses Objekt. Durch diesen Mechanismus wird beim Ausbleiben von Objektfunden der Suchbereich für ein Objekt schrittweise erhöht bzw. bei einem Fund wieder verringert. Daher auch die Bezeichnung dynamische Suchbereichsanpassung.
Gibt es zu einem zuvor gesuchten Objekt keine Suchanfrage mehr, so wird eine Nachricht an alle mobilen Knoten geschickt den Fund des Objekts nicht mehr zu melden. Die Anpassung des Suchbereichs wird jedoch fortgeführt. Dieser vergrößert sich über Timeout Nachrichten so lange, bis die Wurzel erreicht wurde. Eine Änderung des Suchbereichs eines Objekts zu dem momentan keine Suchanfrage existiert bewirkt keine Benachrichtigung der mobilen Knoten. Der Hintergrund für das Verwalten des Suchbereichs, auch ohne Suchanfrage, besteht darin, bei einer erneuten Anfrage für dieses Objekt initial einen möglichst kleinen Suchbereich zu haben.
Im Weiteren werden nun die dargestellten Vorgänge detailliert beschrieben.

Objekt-Position-Cache und Positionsabschätzung

Um die Liste der Objekts-IDs entsprechend anpassen zu könne, besitzt jeder Suchanfrage-Koordinator, außer der Wurzel, einen sogenannten Objek-Position-Cache. Dies ist eine Datenstruktur mit Einträgen der Form <id, Position, Zeitstempel, Timeout, Anfrage> . Dabei entspricht das Feld id der Objekt-ID, Position und Zeitstempel repräsentieren die letzte bekannte Position des Objekts und wann es dort gefunden wurde. Dabei ist die Position P eine Objekts, die uns ein mobiler Knoten übermittelt, generell wie schon zuvor beschrieben ein Kreis, bzw. lässt sich durch ein Tupel P=(x,y,r) repräsentieren, wobei wir annehmen, dass $r=r_{Abweichung}+r_{Leser}$ global bekannt ist und daher in einer Objektfund Nachricht nur die Koordinaten des mobilen knoten übermitelt werden. Um die Position eines Objekts geometrisch einfacher abschätzen zu können, approximieren wir diesen Kreis durch ein achsenparalleles Quadrat mit Seitenlänge 2r und Mittelpunkt (x,y). Da unsere abgeschätzte Position aus dem Schnitt mehrerer Quadrate bzw. Rechtecke bestehen wird, speichern wir als Position im Objekt-Position-Cache allgemein ein Rechteck. Dabei definieren wir ein Rechteck, analog zu einem Bereich, durch seine 4 Grenzen mittels eines Tupels $(x_{min},x_{max},y_{min}, y_{max})$. Der Eintrag für die Position im Objekt-Position-Cache ist entsprechend ein Rechteck bzw. das entsprechende Tupel. Bevor wir uns diese Positionsabschätzung genauer ansehen, seien noch die Felder Timeout und Anfrage beschrieben. Das Feld Timeout gibt an, wann ein Objekt die Grenzen des Bereichs, für die der Suchauftrag-Koordinator zuständig ist welcher zu diesem Objekt-Position-Cache gehört, überschritten haben kann und daher der Vaterknoten benachrichtigt werden muss. Das Feld Anfrage signalisiert, ob momentan für diese Objekt-ID eine Suchanfrage vorliegt und je nachdem hat das Hinzufügen oder Löschen eines Objekt-Position-Cache Eintrags Auswirkungen auf die lokale List der gesuchten IDs.
Als Beispiel für die Abschätzung der Position eines Objekts o_i nehmen wir an, dass wir zwei Objektfund Nachrichten jeweils zum den Zeitpunkten t_1' und t_2' empfangenen haben und deren Inhalt jeweils ein Eintrag der Form (x_j,y_j,t_j) , $j\in\{1,2\}$, ist (vergleiche Abbildung 4.9). Weiterhin nehmen wir an, dass wir das Objekt o_i zuvor noch nicht gefunden haben. Daher wird zum Zeitpunkt t_1' die Position durch ein Quadrat mit Mittelpunkt $M_1=(x_1,y_1)$ und Seitenlänge $s_1(t_1')=2\cdot(r_{Abweichung}+r_{Leser}+(t_1'-t_1)\cdot v_{max})$ repräsentiert. Zum Zeitpunkt t_2' erhalten wir die zweite Objektfund Nachricht. Diese beschreibt analog ein Quadrat mit Mittelpunkt $M_2=(x_2,y_2)$ und Seitenlänge $s_2(t_2')=2\cdot(r_{Abweichung}+r_{Leser}+(t_2'-t_2)\cdot v_{max})$. Da sich das Objekt seit dem Zeitpunkt t_1 mit Geschwindigkeit v_{max} weiterbewegen kann, muss die Seitenlänge des Quadrats der ersten gemeldeten Position auf $s_1(t_2')=2\cdot(r_{Abweichung}+r_{Leser}+(t_2'-t_1)\cdot v_{max})$ angepasst werden. Um nun die Position auf Grundlage dieser beiden Quadrate abzuschätzen, schneiden wir diese miteinander und berechnen des aus dem Schnitt resultierenden Rechtecks. Der Schnitt zweier achsenparalleler Rechtecke lässt sich wie in Algorithmus 4 beschrieben berechnen. Dabei ist die Idee, dass bei Überlappung der Rechtecke eine oder beide Grenzen in x-Richtung eines der Rechtecke zwischen den Grenzen in x-Richtung des anderen Rechtecks liegen. Ist dies nicht der Fall, so schneiden sich die Rechtecke nicht.

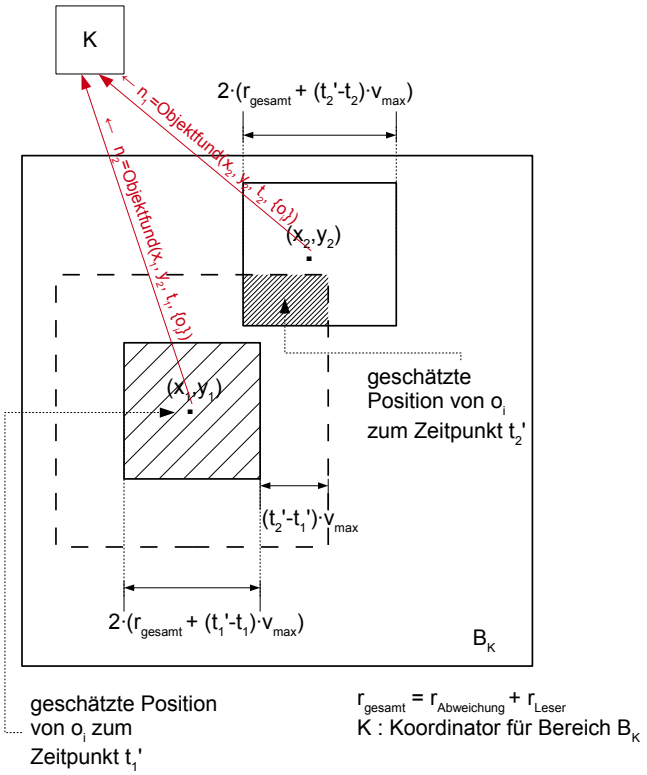

Abbildung 4.9: Quadtree: Abschätzung der Objektposition

In unserem Beispiel mit den 2 Objektfund Nachrichten wäre der Aufruf zum Zeitpunkt t_2' entsprechend für die 2 Rechtecke mit den Grenzen $(x_j - \frac{s_j(t'_2)}{2}, x_j + \frac{s_j(t'_2)}{2}, y_j - \frac{s_j(t'_2)}{2}, y_j + \frac{s_j(t'_2)}{2}), j \in \{1, 2\}$). Nachdem wir mittels der Funktion das Rechteck für den Schnitt $R_{Schnitt}$ berechnet haben, wird im nächsten Schritt der Zeitpunkt berechnet zu dem das über die Zeit wachsende Rechteck, das am Anfang $R_{Schnitt}$ entspricht, eine der Bereichsgrenzen des Koordinator Knotens , welcher, so nehmen wir an, die beiden Objektfund Nachrichten erhalten hat, überschritten wurde. Dabei bedeutet eine Überschreitung einer der Bereichsgrenzen, dass das Objekt auch in anderen Zellen gesucht werden muss. Bei gegebenem Bereich B_K eines Koordinator Knotens K und einem Rechteck R welches zum Zeitpunkt t die Position des Objekts abschätz, berechnet sich der Zeitpunkt des Überschreitens einer der Grenzen wie in Algorithmus 5 beschrieben. Hierbei werden zunächst die Abstände der sich entsprechenden

Algorithmus 4 Funktion zum Schnitt zweier achsenparalleler Rechtecke

Rechteck function *Schnitt*(R_1, R_2)

```
1: R_Schnitt := (-1,-1,-1,-1);
2: for all e ∈ {x_min, x_max} do
3:    for i = 1 to 2 do
4:       if (R_i.e ≤ R_((i mod 2)+1).x_max) ∧ (R_i.e ≥ R_((i mod 2)+1).x_min) then
5:          R_Schnitt.e := R_i.e;
6:       end if
7:    end for
8: end for
9: for all e ∈ {y_min, y_max} do
10:   for i = 1 to 2 do
11:      if (R_i.e ≤ R_((i mod 2)+1).y_max) ∧ (R_i.e ≥ R_((i mod 2)+1).y_min) then
12:         R_Schnitt.e := R_i.e;
13:      end if
14:   end for
15: end for
16: return R_Schnitt;
```

end function

Grenzen berechnet und daraus das wird das Minimum ermittelt. Ist es negativ, so wurde die Grenze bereits überschritten. Ansonsten ist dies der minimale Abstand des Objekts zur Grenze der Zelle. Zusammen mit der maximalen Objektgeschwindigkeit lässt sich daraus die Dauer bis zum erreichen der Grenze berechnen.
Betrachtet man nun zusammengefasst nacheinander die lokalen Abläufe eines wie in Abbildung 4.9 dargestellten Koordinators K nach dem Erhalt zweier Objektfund Nachrichten für Objekt o_i, so sind diese wie folgt. Beim Erhalt der ersten nachricht zum Zeitpunkt t', gehen wir davon aus, dass kein Eintrag für o_i im Objekt-Position-Cache von K existiert und dass k bekannt ist, dass eine Suichanfrage für o_i existiert, welche während des betrachteten Zeitraums bestehen bleibt. Dann hat der Erhalt der Nachricht n_1=(x_1, y_1, t_1, {o_i}) zur Folge, dass K einen neuen Eintrag e=(o_i, KoordinatenZuPosition(x_1, y_1,t_1,t_1'), t_1', BerechneTimeout(B_K, KoordinatenZuPosition(x_1, y_1, t_1, t_1')), true) in den Objekt-Position-Cache einfügt. Da zwischen dem Fund und dem Empfang der Nachricht beim Koordinator eine gewisse Zeit vergehen kann, muss das Quadrat, welches den Bereich in dem sich das Objekt befinden kann eingrenzt, entsprechend vegrößert werden. Würde K nun keine weitere Objektfund Nachrichten für o_i erhalten, so würde der Objekt-Position-Cache nach Ablauf der Zeitspanne e.Timeout dem Koordinator K eine interne Benachrichtigung schicken, dass der Eintrag abgelaufen ist. Eine Reaktion von K auf so eine Benachrichtigung ist den Vaterknoten zu informieren, welcher dann veranlasst, dass das Objekt in einem entsprechend größeren Bereich gesucht wird. Die genauen Schritte werden im Abschnitt über die Interaktion zwischen den Komponenten erläutert.

Algorithmus 5 Funktion zur Timeout Berechnung

Timeout function $BerechneTimeout(B, R)$

1: Abstand := ∞;
2: **if** ($(B.x_{max} - R.x_{max}) < Abstand$) **then**
3: Abstand := B.x_{max}-R.x_{max};
4: **end if**
5: **if** ($(R.x_{min} - B.x_{min}) < Abstand$) **then**
6: Abstand := R.x_{min}-B.x_{min};
7: **end if**
8: **if** ($(B.y_{max} - R.y_{max}) < Abstand$) **then**
9: Abstand := B.y_{max}-R.y_{max};
10: **end if**
11: **if** ($(R.y_{min} - B.y_{min}) < Abstand$) **then**
12: Abstand := R.y_{min}-B.y_{min};
13: **end if**
14: **if** Abstand < 0 **then**
15: return 0;
16: **end if**
17: return $\frac{\text{Abstand}}{v_{max}}$;

end function

Algorithmus 6 Funktion um Koordinate in abgeschätzte Position umzurechnen

Rechteck function $KoordinateZuPosition(x, y, t_{gesendet}, t_{empfangen})$

1: s_{halbe}:=$r_{Abweichung}$ +r_{Leser} +($t_{empfangen}$-$t_{gesendet}$)·v max;
2: return (x-s_{halbe}, x+s_{halbe}, y-s_{halbe},y+s_{halbe});

end function

Beim Erhalt der zweiten Objektfund Nachricht n_2=(x_2, y_2, t_2, {o_i}) zum Zeitpunkt t_2'>t_1' findet K den zuvor eingefügten Eintrag e im Objekt-Position-Cache. Das in e gespeicherte Quadrat bzw. allgemein Rechteck e.Position wirdproportional zur vergangenen Zeitspanne t_2'-e.Zeitstempel=t_2'-t_1' vergrößert. Das resultierende Rechteck R_1=passeRechteckAn(e.Position, e.Zeitstempel, t_2') wird wie in Algorithmus 7 angegeben berechnet. Die mögliche Position von o_i zum Empfangszeitpunkt t_2' berechnet sich aus Nachricht n_2 zu R_2=KoordinateZuPosition(n_2.x, n_2.y, n_2.Zeitstempel, t_2'). Es wird nun der Schnitt beider Rechtecke $R_{Schnitt}$=Schnitt(R_1, R_2) berechnet. Falls v_{max} der maximalen Geschwindigkeit des Objekts o_i entspricht oder größer ist, dann ist sichergestellt, dass dieser Schnitt nicht leer ist. Ansonsten wäre der resultierende Schnitt R_2, was dem ungünstigsten Fall entspricht, da wir generell durch den Schnitt der beiden Rechtecke versuchen R_2 zu verkleinern. Dieser Fall tritt auch auf, wenn R_2 komplett in R_1 liegt. Den Eintrag e im Objekt-Position-Cache für o_i aktualisieren wir entsprechend zu

Algorithmus 7 Funktion um Rechteck proportional zur maximalen Objektgeschwindigkeit zu vergößeren

Rechteck function *passeRechteckAn*(R, t, t')

1: delta := (t-t')·v_{max};
2: return ($R.x_{min}$-delta, $R.x_{max}$+delta, $R.y_{min}$-delta, $R.y_{max}$+delta);

end function

e.Zeitstempel=t_2' und e.Position=$R_{Schnitt}$. Weiterhin aktualisieren wir den Zeitpunkt für das Überschreiten einer der Grenzen zu e.Timeout=berechneTimeout(B_K, $R_{Schnitt}$, t_2').

Interaktion zwischen den Komponenten

Nachdem wir nun das Abschätzen der Position eines Objekts und die Verwendung dieser im Zusammenhang mit dem Objekt-Position-Cache des jeweiligen Koordinators beschrieben haben, erläutern wir nun die Interaktion zwischen den Knoten. Aus zeitlichen Gründen konnte dieser Teil des hierarchischen Ansatzes nicht mehr vollständig konsistent beschrieben werden. Der vielversprechendere Gitter-Ansatz ist jedoch vollständig.

Nachrichtentypen

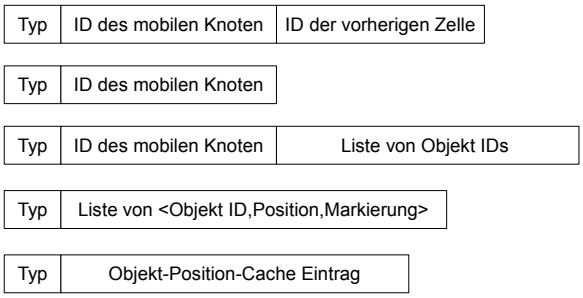

Abbildung 4.10: Nachrichtentypen für hierarchischen Ansatz,
ZELL_WECHSEL
INFRASTRUKTUR_AUFTRAGS_LISTE_ANFRAGE
INFRASTRUKTUR_AUFTRAGS_LISTE_ANTWORT
INFRASTRUKTUR_OBJEKT_FUND
INFRASTRUKTUR_TIMEOUT

Die Kommunikation zwischen den Suchanfrage-Koordinatoren untereinander und zwischen Suchanfrage-Koordinatoren und mobilen Knoten geschieht wie beim Basis Ansatz durch Nachrichten verschiedenen Typs. Dabei verwenden wir zusätzlich zu den beim Basis-Ansatz benutzten Nachrichtentypen die in Abbildung 4.10 dargestellten Typen. Da wir das Gebiet in mehrere Zellen aufgeteilt haben und die Menge der

gesuchten Objekte in jeder dieser Zellen unterschiedlich sein kann, fragt ein mobiler Knoten beim Wechsel in eine andere Zelle den Unterschied mittels einer Nachricht vom Typ ZELL_WECHSEL beim Koordinator der neuen Zelle ab. Der Koordinator antwortet darauf mit einer Nachricht vom Typ TASK_LIST_DELTA, welche die Objekt-IDs enthält, die nun zusätzlich vom neu in die Zelle gewechselten Knoten gesucht werden müssen. Alle Nachrichten mit der Vorsilbe "INFRASTRUKTUR_" werden nur zwischen Suchanfrage-Koordinatoren verwendet. Um herauszufinden welche Objekt-IDs ein mobiler Knoten bereits sucht und um ihm dann die passende Menge an zusätzlich zu suchenden Objekt-IDs zu senden, werden die Nachrichten vom Typ INFRASTRUKTUR_AUFTRAGS_LISTE_ANFRAGE und INFRASTRUKTUR_AUFTRAGS_LISTE_ANTWORT verwendet, mittels derer die Koordinatoren die Liste Objekt-IDs eines mobilen Knotens austauschen. Eine Nachricht vom Typ INFRASTRUKTUR_TIMEOUT wird dazu verwendet dem Vaterknoten mitzuteilen, dass ein bestimmtes Objekt nun in einem größeren Bereich gesucht werden muss.

Suchanfrage

Algorithmus 8 Wurzel bekommt neue Suchanfrage

procdure $Anfrage(id, dauer, permanent)$

```
1: if ∀e ∈ L_Anfragen : e.id ≠ id then
2:    nachricht.Typ := INFRASTRUKTUR_AUFTRAGS_LISTE_DELTA;
3:    nachricht.fuegeHinzu(id);
4:    if id ∈ T_ObjCache then
5:       for all k ∈ T_ObjCache[id] do
6:          sende(nachricht, k);
7:       end for
8:    else
9:       for all k ∈ L_Kindknoten do
10:         sende(nachricht, k);
11:      end for
12:   end if
13: end if
14: L_Anfragen := L_Anfragen ∪ {(generiereAnfragenID(), id, dauer, permanent)};
```

end procedure

Erhält die Wurzel eine neue Suchanfrage für Objekt o_i, so speichert sie diese und prüft zunächst, wie beim Basis-Ansatz, ob eine andere Suchanfrage für o_i existiert. Falls dem so ist, muss nichts getan werden. Ansonsten prüft die Wurzel, ob zu o_i ein Eintrag in einer von ihr lokal verwalteten Lookup-Tabelle existiert. Dabei bildet die Lookup-Tabelle eine Objekt-ID auf die Menge der Kindknoten ab, welche selbst einen Objekt-Position-Cache Eintrag für o_i besitzen oder dies für einen Knoten in ihrem Unterbaum zutrifft. Wie genau diese Lookup-Tabelle aufgebaut wird, werden wir im

Algorithmus 9 Koordinator ≠ Wurzel oder Blattknoten erhält Nachricht

procdure *Empfange(nachricht)*

1: **if** $nachricht.Typ = INFRASTURKTUR\_WURZEL\_AUFTRAGS\_LISTE\_DELTA$ **then**
2: **for all** $e \in nachricht.IDsHinzufuegen()$ **do**
3: **if** $e \in T_{ObjCache}$ **then**
4: nachricht'.Typ := INFRASTURKTUR_WURZEL_AUFTRAGS_LISTE_DELTA;
5: nachricht'.fügeHinzu(e);
6: **for all** $k \in T_{ObjCache}$ **do**
7: sende(nachricht',k);
8: **end for**
9: **else**
10: **if** $e \in C_{ObjPosition}$ **then**
11: $C_{ObjPosition}$[e].AnfrageExistiert := true;
12: **end if**
13: nachricht'.Typ := INFRASTURKTUR_AUFTRAGS_LISTE_DELTA;
14: nachricht'.fügeHinzu(e);
15: **for all** $k \in L_{Kindknoten}$ **do**
16: sende(nachricht',k);
17: **end for**
18: **end if**
19: **end for**
20: **else if** $nachricht.Typ = INFRASTURKTUR\_AUFTRAGS\_LISTE\_DELTA$ **then**
21: **for all** $k \in L_{Kindknoten}$ **do**
22: sende(nachricht,k);
23: **end for**
24: **end if**

end procedure

Verlauf beschreiben. Existiert ein Eintrag in der Lookup-Tabelle der Wurzel, so wurd eine Nachricht vom Typ INFRASTRUKTUR_WURZEL_AUFTRAGS_LISTE_DELTA deren Liste der hinzuzufügenden Objekt-IDs die ID von o_i enthält an der dem Eintrag entsprechenden Menge von Kinknoten geschickt. Ansonsten bekommen alle Kindknoten die Nachricht. Das Format der Nachricht vom Typ INFRASTRUKTUR_WURZEL_AUFTRAGS_LISTE_DELTA ist gleich mit dem der aus dem Basis-Ansatz bekannten Nachricht vom Typ AUFTRAGS_LISTE_DELTA.
Die mögliche Existenz eines Objekt-Position-Cache Eintrags für o_i rührt daher, dass trotz fehlender Suchanfrage für ein Objekt, sein vorhandener Objekt-Position-Cache Eintrag weiter verwaltet und weitergegeben wird, bis er bei der Wurzel ankommt. Dieser Mechanismus wird bei der Behandlung einer Nachricht vom Typ INFRASTRUKTUR_TIMEOUT näher erläutert. Wir fahren nun mit dem nächsten Schritt fort, bei

Algorithmus 10 Blattknoten erhält Nachricht

procedure *Empfange*(*nachricht*)

1: **if** *nachricht.Typ* = *INFRASTURKTUR_WURZEL_AUFTRAGS_LISTE_DELTA* **then**
2: **for all** $e \in$ *nachricht.IDsHinzufuegen*() **do**
3: $C_{ObjPosition}$[e].AnfrageExistiert:=true;
4: $L_{ObjektID} \cup \{e\}$;
5: **end for**
6: **else if** *nachricht.Typ* = *INFRASTRUKTUR_AUFTRAGS_LISTE_DELTA* **then**
7: **for all** $e \in$ *nachricht.IDsHinzufuegen*() **do**
8: $L_{ObjektID} \cup \{e\}$;
9: **end for**
10: **for all** $e \in$ *nachricht.IDsEntfernen*() **do**
11: $L_{ObjektID} \setminus \{e\}$; $C_{ObjPosition}$[e].AnfrageExistiert:=true;
12: **end for**
13: **else if** *nachricht.Typ* = *OBJEKT_FUND* **then**
14: $t_{Empfang}$:=jetzt();
15: **for all** $e \in$ *nachricht.IDListe* **do**
16: **if** $e \in C_{ObjPosition}$ **then**
17: position:=Schnitt(KoordinateZuPosition(nachricht.x, nachricht.y, nachricht.Zeitstempel, $t_{Empfang}$), passeAn($C_{ObjPosition}$[e].Position, $C_{ObjPosition}$[e].Zeitstempel, $t_{Empfang}$));
18: $C_{ObjPosition}$[e].Timeout := BerechneTimeout(B_K, Position, $t_{Empfang}$);
19: $C_{ObjPosition}$[e].Zeitstempel := $t_{Empfang}$;
20: **else**
21: position := KoordinateZuPosition(nachricht.x, nachricht.y, nachricht.Zeitstempel, $t_{Empfang}$);
22: timeout := BerechneTimeout(B_K, position, $t_{Empfang}$);
23: $C_{ObjPosition}$.fügeHinzu(e, position, $t_{Empfang}$);
24: **end if**
25: **end for**
26: **else if** *nachricht.Typ* = *ZELL_WECHSEL* **then**
27: nachricht'.Typ := ZELL_WECHSEL_ANFRAGE;
28: nachricht'.mobilerKnoten := nachricht.mobilerKnoten;
29: nachricht'.Zelle := this.ID; sende(nachricht', nachricht.ZellID);
30: **end if**

end procedure

dem ein Suchanfrage-Koordinator von der Wurzel bzw. ganz allgemein eine Nachricht vom Typ INFRASTRUKTUR_WURZEL_AUFTRAGS_LISTE_DELTA bekommt. Dieser prüft für alle Objekt-IDs, die laut Nachricht hinzugefügt werden sollen, ob ein Eintrag für die jeweilige Objekt-ID in seiner Lookup-Tabelle existiert. Falls ja,

so schickt er analog zur Wurzel den betreffenden Kindknoten eine Nachricht vom Typ INFRASTRUKTUR_WURZEL_AUFTRAGS_LISTE_DELTA, welche nur die gerade betrachtete ID enthält. Gibt es für eine der hinzuzufügenden Objekt-IDs keinen Eintrag in der Lookup-Tabelle, so wird eine Nachricht welche dies Objekt-ID enthält an alle Kindknoten geschickt. Wenn bei diesem Schritt mehrere Nachrichten an einen Kindknoten geschickt werde sollen, so können diese vom Koordinator zuvor zu einer Nachricht zusammengefasst werden.
Dieses Weiterreichen einer Nachricht vom Typ INFRASTRUKTUR_WURZEL_AUFTRAGS_LISTE_DELTA geschieht so lange rekursiv bis entweder ein Blattknoten erreicht wurde oder ein Objekt-Position-Cache Eintrag beim empfangenden Koordinator existiert. Falls es bei einem Blattknoten endet, der keinen Eintrag für die Objekt-ID im Cache hat, fügt dieser die ID in seine lokalte Liste $L_{ObjektID}$ ein und schickt an alle mobilen Knoten seiner Zelle eine AUFTRAGS_LISTE_DELTA Nachricht. Weiterhin besitzt jeder Blattknoten zu jedem mobilen Knoten seiner Zelle eine Liste mit Objekt-IDs, welche der entsprechende mobile Knoten gerade sucht. Zur Speicherung verwenden wir auch hier eine Lookup-Tabelle. Beim Erhalt der Auftragsliste bestätigt der mobile Knoten jede Objekt-ID, die er nun zusätzlich sucht. Beim Empfang der Betätigung aktualisiert der Koordinator den Eintrag des mobilen Knoten in der Lookup-Tabelle.
Zusammengefasst bewirkt dies, dass ein Suchauftrag für ein Objekt o_i initial nur dort verteilt wird, wo sich das Objekt aus Sicht des Systems befinden kann. Da unsere Abschätzung pessimistisch ist, verlieren wir dadurch keine Objektfunde.

Objektfund

Findet ein mobiler Knoten ein oder mehrere momentan gesuchte Objekte, dann schickt er wie beim Basis-Ansatz eine Nachricht vom Typ OBJEKT_FUND. Besitzt der empfangende Koordintor einen Objekt-Cache Eintrag für die jeweilige Objekt-ID, so wird der Eintrag, wie im Abschnitt über die Abschätzung der Objektposition erläutert, aktualisiert. Ansonsten erstellt er eine neuen Eintrag. Beide male berechnet er außerdem den Wert für den Timeout. Ist dieser kleinergleich Null, so wird der Eintrag nicht in den lokalen Objekt-Position-Cache eingefügt. Stattdessen wird in der Nachricht über den Objektfund an dern Vaterknoten eine Markierung *TIMEOUT* für den zur Objekt-ID gehörenden Eintrag gesetzt. Wurde der Objekt-Position-Cache Eintrag für ein Objekt aktualisiert und bisher seit Reaktivierung oder Erstellung des Eintrags kein Fund des Objekts an den Vaterknoten weitergeleitet, so wird eine weiter Markierung *SCHICKE_ENTFERNEN* gesetzt. Die an den Vaterknoten geschickte Nachricht vom Typ INFRASTRUKTUR_OBJEKT_FUND enthält jeweils die abgeschätzte Position eines Objekt und je eine Markierung für die beschriebenen Fälle.
Beim Empfang einer Nachricht vom Typ INFRASTRUKTUR_OBJEKT_FUND wird für jeden Eintrag bei dem eine Markierung SCHICKE_ENTFERNEN existiert an alle Kindknoten, außer vom Absender der Nachricht, eine Benachrichtigung geschickt die entsprechenden Objekt-IDs aus der Liste der gesuchten IDs zu entfernen. Als Resultat wird nur noch in einem bestimmten Bereich nach dem Objekt gesucht.

4.4.2 Gitter-Ansatz

Wir verwenden nun eine Gitterstruktur um die Infrastruktur-Knoten untereinander zu verbinden. Dabei unterteilen wir das Gebiet in n x m verschiedene Bereiche. Der Begriff Zelle wird hier gleichbedeutend mit dem Begriff Bereich verwendet. Eine Zelle ist als ein Tupel $Z=(x_{min}, x_{max}, y_{min}, y_{max})$, welches die Zellgrenzen festlegt. Jeder Zelle Z_i ist analog zum hierarchischen Ansatz ein Zell-Knoten ZK_i zugeordnet. Dieser ist für die Verteilung von Suchaufträgen an alle mobilen Knoten, die sich innerhalb seiner Zelle befinden, zuständig. Findet ein mobiler Knoten ein oder mehrere Objekte, so schickt er eine Objektfund-Nachricht an den Zell-Knoten, in dessen Bereich er sich gerade befindet. Somit erhält ein Zell-Knoten alle Nachrichten über gefundene Objekte von den mobilen Knoten innerhalb seiner Zelle.

Wir gehen davon aus, dass das gesamte Gebiet in dem der Dienst zur Verfügungn gestellt wird ein Rechteck mit Seitenlängen a und b darstellt. Wir wählen die Unterteilung so, dass alle Zellen exakt gleich groß sind und im Allgemeinen achsenparallele Rechtecke bilden. Demnach sind die Abmessungen einer Zelle $\frac{a}{n}$ in der einen und $\frac{b}{m}$ in der anderen Dimension. Insgesamt ergibt sich eine regelmäßige Gitterstruktur (vergleiche Abbildung 4.11).

Neben der geometrischen Aufteilung sind die Zell-Knoten wie folgt entweder direkt bzw. indirekt miteinander verbunden. Jeder Zell-Knoten kennt den horizontal, vertikal und diagonal von ihm gelegenen Nachbar-Zell-Knoten und kann mittels Nachrichten mit ihnen kommunizieren. Dabei spielen die diagonalen Nachbarn, wie wir sehen werden, nur beim Zellwechsel eines mobilen Knoten eine Rolle. Der Algorithmus zur dynamischen Suchbereichsanpassung verwendet nur die horizontalen und vertikalen Nachbarn. Jeder Zell-Knoten besitzt außerdem wie schon zuvor eine lokale Liste $L_{ObjektID}$ mit den IDs der in seiner Zelle momentan gesuchten Objekte und in Form von Suchaufträgen an die mobilen Knoten verteilt werden. Außerdem speichert sich der Zell-Knoten eine Lookup-Tabelle T_{MK}, die zu jedem mobilen Knoten in seiner Zelle die Menge der von diesem Knoten bereits gesuchten Objekt-IDs zurückgibt. Genauso besitzt ein Zell-Knoten analog zum hierarchischen Ansatz eine Objekt-Positions-Cache C_{ObjPos}, dessen Aufbau wir weiter unten beschreiben. Wie beim hierarchischen Ansatz ist das Ziel diese Listen so anzupassen, dass ein Objekt nur in Zellen gesucht wird, in denen sich es aus Sicht des Systems befinden kann. Dies geschieht hier im Gegensatz zum hierarchischen Ansatz jedoch über direkte Kommunikation der Zell-Knoten untereinander.
Neben den Zell-Knoten gibt es außerdem noch einen puren Infrastruktur-Knoten K_0, der wie zuvor in Kapitel 3.1 definiert, ausschließlich mit anderen Infrastruktur-Knoten Nachrichten austauscht. In diesem Fall kommuniziert er mit den einzelnen Zell-Knoten, vermittelt jedoch keine Nachrichten zwischen ihnen. Er dient allein als Einstiegspunkt für Suchanfragen und der initialen Verteilung der Suchaufträge an die entsprechenden Koordinatoren bzw. Zell-Knoten. Die Zell-Knoten leiten die Suchaufträge wiederum an die mobilen Knoten weiter. Des Weiteren leiten die Zell-Knoten die Nachrichten über

Objektfunde an K_0 weiter. Der Knoten K_0 leitet die Objektfunde dann weiter an die Anwendung, welche die Suchanfrage gestellt hat. Wie an den gerade beschriebenen Funktionen zu erkennen, dient der Knoten K_0 bei diesem Ansatz als Suchanfrage-Verwaltungs Knoten, Suchauftrag-Koordinator und Wurzel Knoten. Daher bezeichnen wir K_0 auch nur als die Wurzel. Analog zu den vorherigen Ansätzen verwaltet K_0 eine Liste der aktuell aktiven Suchanfragen. Außerdem besitzt er eine Lookup-Tabelle T_{ObjC}, welche zu einer Objekt-ID die Menge der Kindknoten angibt, die einen Objekt-Cache Eintrag für diese Objekt-ID verwalten.

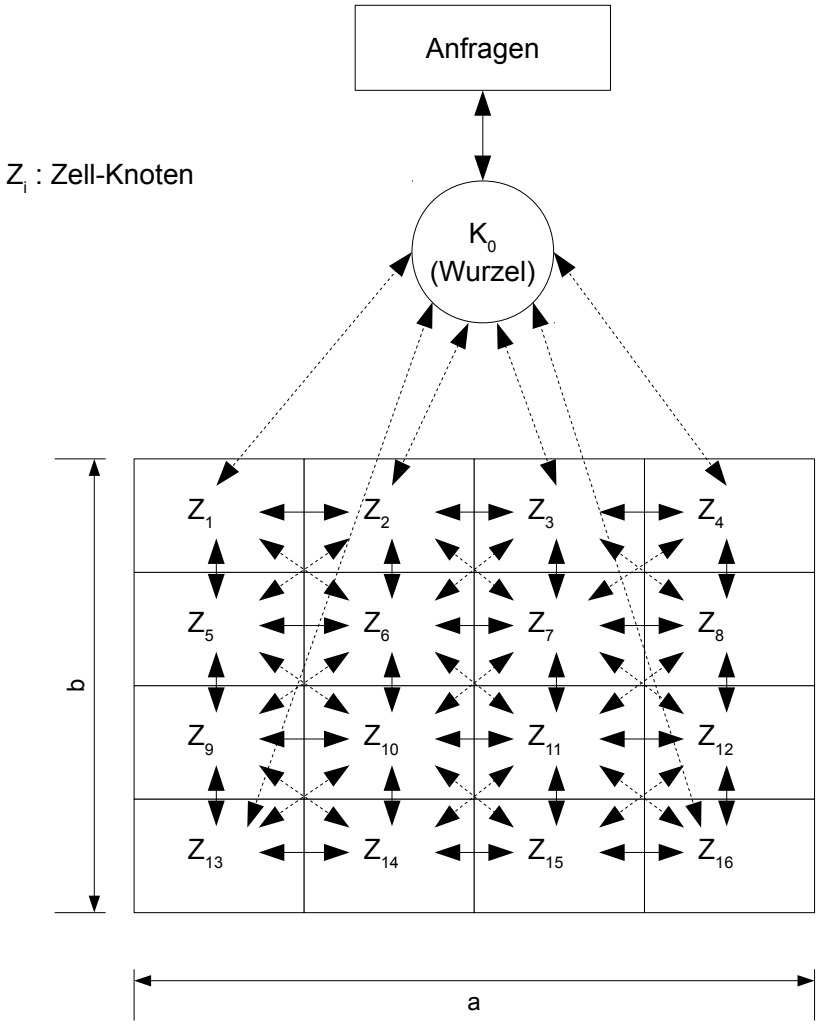

Abbildung 4.11: Architektur Gitter-Ansatz

dynamische Suchbereichsanpassung

Wie schon zuvor beim hierarchischen Ansatz, beschreiben wir nun, wie der Algorithmus zur dynamischen Suchbereichsanpassung bei der Gitter-Struktur prinzipiell funktioniert. Auch hier liegt der Fokus wieder darauf wie sich die Abschätzung der Objektposition auf den effektiven Suchbereich auswirkt, d.h. welche Zellen nach dem Objekt suchen, und wie reagiert wird, wenn sich die Abschätzung der Positon durch das System mit der Zeit ändert.

Wir beginnen mit unserer Betrachtung bei Abbildung 4.12. Wir nehmen an, dass

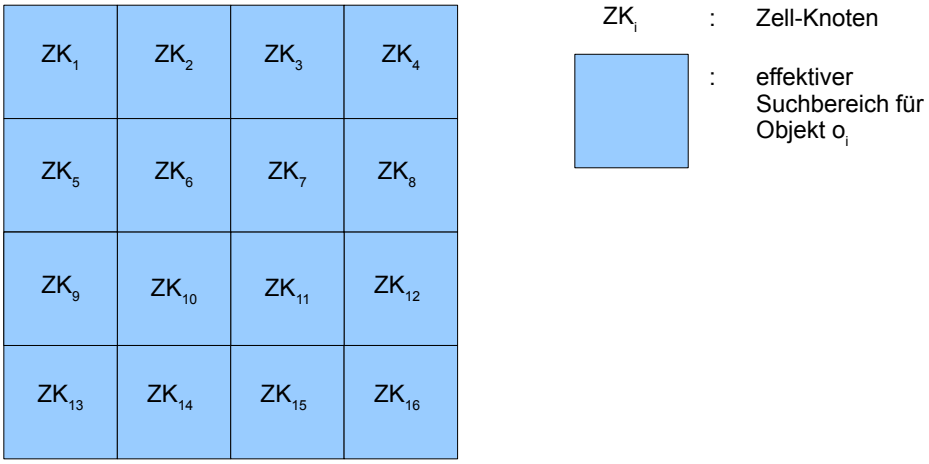

Abbildung 4.12: Suchbereich für Objekt o_i, welches zuvor noch nicht gefunden wurde

eine Suchanfrage für ein Objekt o_i bei der Wurzel eingetroffen ist und o_i bisher noch nicht gefunden wurde. Außerdem existiert kein andere Suchanfrage für o_i. Die vom System abgeschätzte Position für o_i entspricht dem kompletten Dienstgebiet. Daher beauftrag die Wurzel jeden Zell-Knoten ZK_i nach o_i zu suchen. Dies hat wiederum zur Folge, dass jeder Zell-Knoten alle mobilen Knoten in seiner Zelle beauftrag nach o_i zu suchen. Der effektive Suchbereich entspricht somit initial dem ganzen Dienstgebiet.

Wird nun das Objekt o_i beispielsweise in Bereich von Zell-Knoten ZK_6 zum Zeitpunkt t_1 gefunden (siehe Abbildung 4.13), so prüft ZK_6 zunächst, ob er einen Eintrag für o_i in seinem Objekt-Positions-Cache, den wir noch näher beschreiben werden, besitzt. Dies ist nicht der Fall, da o_i bisher noch nicht gefunden wurde. Daher schickt ZK_6 eine Broadcast-Nachricht zum Entfernen von o_i an seine horizontalen und vertikalen Nachbarn. Diese Broadcast-Nachricht wird gemäß dem in [RFH+01] vorgestellten Mechanismus an alle Zell-Knoten verteilt, die nun o_i aus ihrer lokalen Liste mit den IDs der gesuchten Objekte entfernen. Danach berechnet ZK_6, analog zum hierarchischen Ansatz, für jede seiner Zellgrenzen den Zeitpunkt bei dem das über die Zeit wachsende Quadrat, welches die Objektposition darstellt, die jeweilige Grenze überschreitet. Daraus ergeben sich 4 Timout-Werte. Im vorliegenden Fall ist der Wert für die linke Grenze

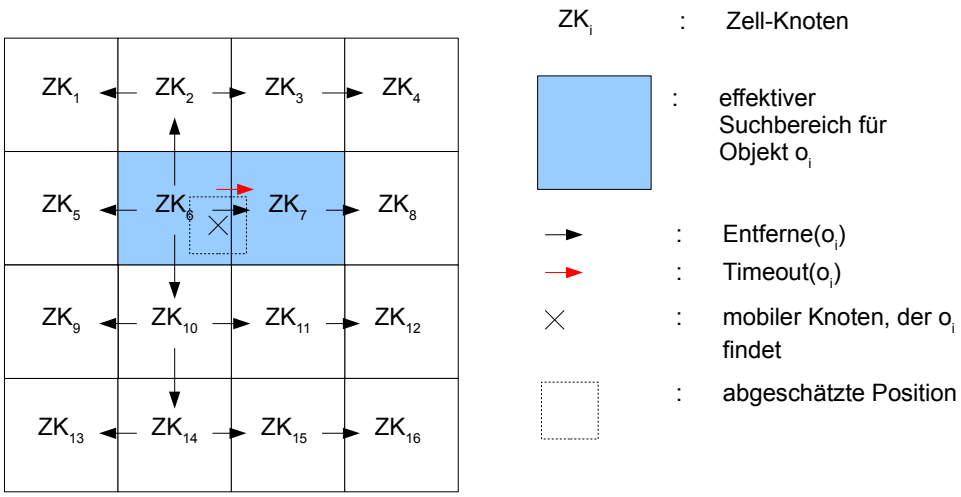

Abbildung 4.13: Suchbereich für Objekt o_i, nach Fund durch mobilen Knoten

von ZK_6 negativ, was bedeutet, dass ZK_6 eine Timout-Nachricht an ZK_7 schickt. Diese Nachricht beinhaltet die Position von o_i, sprich ein Rechteck, und den zugehörigen Zeitpunkt t_1. Danach speichert ZK_6 folgende Daten in seinem Objekt-Positions-Cache für o_i: die Position, den Zeitpunkt t_1, die 3 übrigen Timeout-Wert und den Status seiner 4 Grenzen. Dies werden wir noch genauer erklären. Beim Erhalt der Timeout-Nachricht berechnet ZK_7 analog zu ZK_6 die Timeout-Werte für die restlichen 3 seiner Grenzen und legt analog einen Objekt-Positions-Cache Eintrag an. In diesem Eintrag merkt er sich insbesondere, dass er die Nachricht von seinem rechten Nachbarn erhalten hat und wie beim zuvor vorgestellten Broadcast, weiß ZK_7 so, dass er den Timeout nur nach rechts weitergeben muss.
Außerdem fügt er o_i wieder in seine lokale Objekt-ID Liste ein und verteilt gegebenen Falls Suchaufträge für o_i an alle mobilen Knoten in seiner Zelle, die nicht bereits nach o_i suchen.

Wird nun o_i eine Zeit lang nicht gefunden und es kommt zu dem Fall, dass der Zeitpunkt des Timeouts für die untere Grenze von ZK_6 erreicht wird, so sendet ZK_6 eine Timeout-Nachricht für o_i an ZK_{10}, welche wieder die letze bekannte Position und den Zeitpunkt t_1 enthält (vergleiche Abbildung 4.14). Analog wie zuvor ZK_7 berechnet nun ZK_{10} die Timout-Werte seiner Grenzen und bemerkt, dass seine rechte Grenze schon von der abgeschätzten Position überschritten wurde. Da er die Timeout-Nachricht von seinem oberen Nachbarn bekommen hat, weiß ZK_{10}, dass er den Timeout nach links, rechts und unten weitergeben muss. Daher schickt er an ZK_{11} ebenfalls eine Timeout-Nachricht. Dieser sucht in der Folge nun auch nach o_i.

In Abbildung 4.15 wird nun o_i in Zelle ZK_{11} gefunden. Da ZK_{11} einen Objekt-

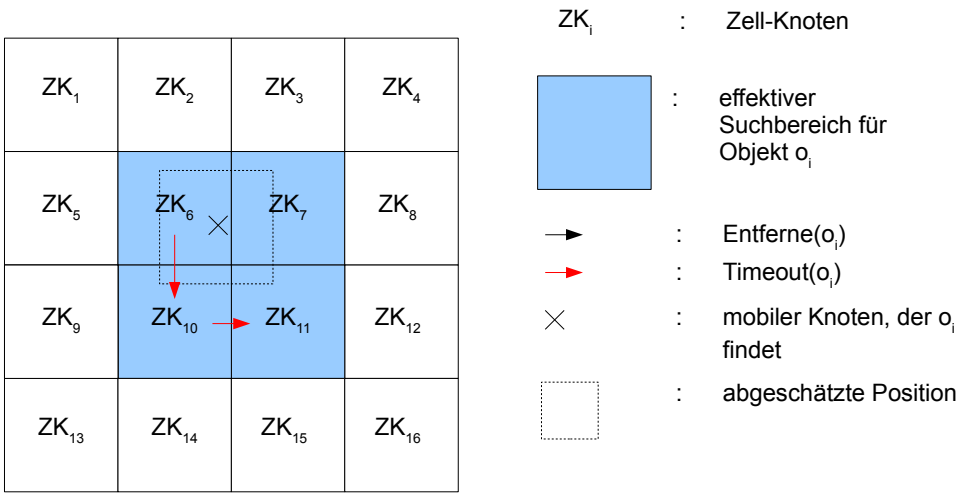

Abbildung 4.14: Suchbereichsanpassung bei Ausbleiben eines Objektfunds

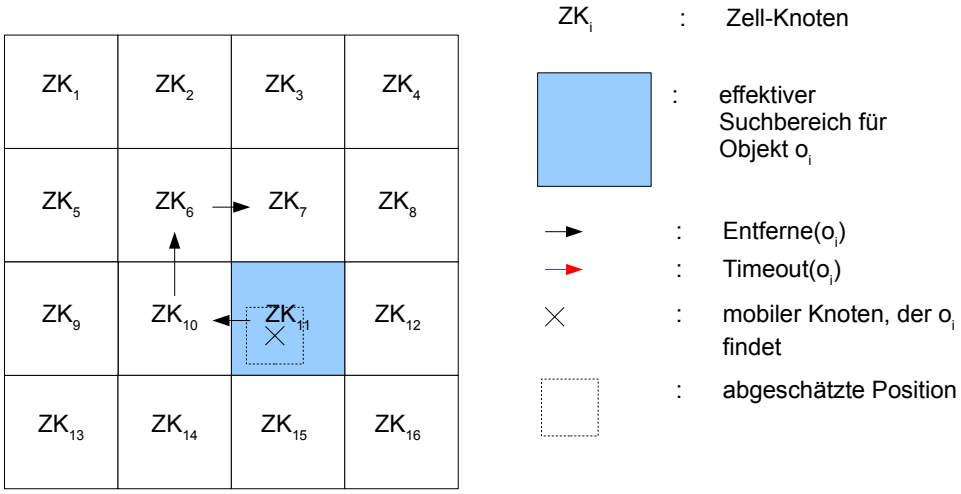

Abbildung 4.15: Suchbereichsanpassung bei erneutem Objektfund

Positions-Cache Eintrag für o_i besitzt, prüft er welche Grenzen den Status "VERTEILT" besitzen. Dies bedeutet, dass diese Nachbarn ebenfalls nach o_i suchen. An diese Nachbarn schickt er nun eine Nachricht zum entfernen von o_i. Hier bekommt nur ZK_{10} so eine Nachricht. ZK_{10} wiederum entfernt o_i aus seinem Cache und seiner Objekt-ID List und schickt die Nachricht an alle Nachbarn weiter deren Status im Objekt-Positions-Cache Eintrag auf "VERTEILT" stand. Die Zell-Knoten ZK_6 und ZK_7 verhalten sich analog. Am Ende entspricht der effektive Suchbereich der Zelle von Knoten ZK_{11}.

Objekt-Positions-Cache

Analog zum hierarchischen Ansatz besitzt auch hier jeder Zell-Knoten einen Objekt-Positions-Cache. Wie dieser verwendet wird, wurde im vorherigen Abschnitt schon deutlich gemacht. Er besteht aus mehreren Einträgen der Form <id, Position, Zeitstempel, Timeout, Anfrage, Grenz-Zustand> und unterscheidet sich vom Objekt-Positions-Cache des hierarchischen Ansatzes im Wesentlichen nur durch das zusätzliche Speichern des Zustands der Grenzen. Dabei gibt es 4 verschiedene Werte, die den Zustand der Grenze beschreiben:
Der Wert RELEVANT bedeutet, dass diese Grenze zur Timeout-Berechnung mit herangezogen werden muss.
Der Wert AKTIV bedeutet, dass der Timout für diese Grenzen aktuell minimal ist und falls der Timeout erreicht wird, muss eine Timeout-Nachricht an diesen Nachbarn geschickt werden. Es ist möglich, dass mehrere Grenzen gleichzeitig diesen Status haben. Dies bedeutet, dass ihr Timeout-Wert identisch ist.
Der Wert VERTEILT bedeutet, dass an diesen Nachbar-Zell-Knoten schon eine Timeout-Nachricht verschickt wurde. Weiterhin wird beim Empfang einer Nachricht zum entfernen einer Objekt-ID diese an alle Nachbarn mit dem Status VERTEILT geschickt. Außer natürlich an den Nachbarn von dem die Nachricht empfangen wurde.
Der Wert NICHT_RELEVANT sagt aus, dass an diesen Nachbarknoten keine Timeout-Nachricht verteilt werden muss und daher wird diese Grenze auch nicht zur Timeout-Berechnung herangezogen. Dieser Wert wird für eine Grenze nur gesetzt, wenn der Zell-Knoten eine Timeout-Nachricht von einem Nachbarknoten erhalten hat. Welche Grenzen diesen Wert bekommen leitet sich davon ob, von welchem Nachbarn die Nachricht erhalten wurde. Das genaue setzen der Werte der Grenze ist in Algorithmus 11 angegeben. Die Position entspricht wieder einem Rechteck, welches die reale Position des Objekts abschätzt. Das Feld *Zeitstempel* gibt an, wann die abgeschätzte Position für das Objekt gültig war. Der Eintrag *Anfrage* gibt an, ob momentan eine Suchanfrage für diese Objekt vorliegt. Falls dem nicht so ist, hat der Empfang einer Timeout-Nachricht keine Auswirkung auf die lokale Liste der gesuchten Objekt-IDs.

Interaktion zwischen den Komponenten

Wir beschreiben nun wieder genauer die Interaktion zwischen den einzelnen Komponenten für verschiedene Ereignisse, analog zu den Ansätzen zuvor.

Suchanfrage

Bekommt die Wurzel eine neue Suchanfrage für ein Objekt o_i (siehe Algorithmus 12), für das bisher keine Suchanfrage vorliegt, so wird zunächst in der Lookup-Tabelle für die Objekt-Positions-Caches der Zell-Knoten nachgesehen, ob ein Eintrag für o_i vorliegt. Ist dies der Fall, so wird an die dort verzeichneten Zell-Knoten eine Nachricht vom Typ WURZEL_AUFTRAGS_LISTE_DELTA geschickt, welche die ID von o_i enthält. Falls kein Eintrag vorliegt wird die Nachricht an alle Zell-Knoten geschickt. Falls es Zell-Knoten gibt, die einen Objekt-Positions-Cache Eintrag für o_i haben, so spiegelt die

Algorithmus 11 Gitter: Initiales Setzen des Grenz-Status beim Empfang einer Timeout-Nachricht

CacheEintrag function *TimeoutNachrichtSetzeGrenzen*(*sender*, *eintrag*)

```
1: if sender = Nachbar[oben] then
2:    eintrag.GrenzZustand(Nachbar[links]):= RELEVANT;
3:    eintrag.GrenzZustand(Nachbar[rechts]):=RELEVANT;
4:    eintrag.GrenzZustand(Nachbar[oben]):= VERTEILT;
5:    eintrag.GrenzZustand(Nachbar[unten]):= RELEVANT;
6: else if sender = Nachbar[unten] then
7:    eintrag.GrenzZustand(Nachbar[links]):= RELEVANT;
8:    eintrag.GrenzZustand(Nachbar[rechts]):=RELEVANT;
9:    eintrag.GrenzZustand(Nachbar[oben]):= RELEVANT;
10:   eintrag.GrenzZustand(Nachbar[unten]):= VERTEILT;
11: else if sender = Nachbar[links] then
12:   eintrag.GrenzZustand(Nachbar[links]):= VERTEILT;
13:   eintrag.GrenzZustand(Nachbar[rechts]):=RELEVANT;
14:   eintrag.GrenzZustand(Nachbar[oben]):= NICHT_RELEVANT;
15:   eintrag.GrenzZustand(Nachbar[unten]):= NICHT_RELEVANT;
16: else if sender = Nachbar[rechts] then
17:   eintrag.GrenzZustand(Nachbar[links]):= RELEVANT;
18:   eintrag.GrenzZustand(Nachbar[rechts]):= VERTEILT;
19:   eintrag.GrenzZustand(Nachbar[oben]):= NICHT_RELEVANT;
20:   eintrag.GrenzZustand(Nachbar[unten]):= NICHT_RELEVANT;
21: end if
```

function

Menge all dieser Zell-Knoten den effektiven Suchbereich für o_i wieder und in der Folge werden die Suchaufträge für o_i initial nur in den Zellen verteilt in denen sich das Objekt aus Sicht des System befinden kann. Für den Fall, dass bereits eine Suchanfrage vorliegt muss, wie bei den bisherigen Ansätzen auch, nichts weiter getan werden. Der Aufbau der Nachricht vom WURZEL_AUFTRAGS_LISTE_DELTA ist analog zur Nachricht vom Typ AUFTRAGS_LISTE_DELTA und enthält zwei Listen von Objekt IDs. Eine steht für die zu hinzuzufügenden Objekt-IDs, die andere für die zu entfernenden Objekt-IDs.

Erhält ein Zell-Knoten eine Nachricht vom Typ WURZEL_AUFTRAGS_LISTE_DELTA, so prüft er, ob ein Objekt-Positions-Cache Eintrag für o_i vorliegt und falls ja, setzt er das Feld *Anfrage* auf true. Weiterhin fügt er die id von o_i in seine Liste $L_{ObjektID}$ ein. Anschließend schickt er an alle mobilen Knoten in seiner Zelle eine Nachricht vom Typ AUFTRAGS_LISTE_DELTA. Dies bewirkt wieder, dass die mobilen Knoten ab sofort den Fund von Objekt o_i melden. Außerdem fügt der Zell-Knoten die ID von o_i allen Einträge seiner Lookup-Tabelle T_{MN} hinzu (vergleiche Algorithmus 13). Es sei angemerkt, dass die Wurzel sofort nach der

Algorithmus 12 Gitter-Ansatz: Wurzel bekommt neue Suchanfrage

procedure $Anfrage(id, dauer, permanent)$

```
1: if ∀e ∈ L_Anfragen : e.id ≠ id then
2:   L_Anfragen := L_Anfragen ∪ {(generiereAnfrageID(), id, dauer, permanent)};
3:   L_Objekt_ID := L_Objekt_ID ∪ {id};
4:   nachricht.typ := WURZEL_AUFTRAGS_LISTE_DELTA;
5:   nachricht.fügeHinzu(id);
6:   if id ∈ T_ObjC then
7:     sende(nachricht,T_ObjC[id]);
8:   else
9:     sendeAnAlleZellKnoten(nachricht);
10:  end if
11: end if
```

end procedure

Suchanfrage eine Nachricht vom Typ WURZEL_AUFTRAGS_LISTE_DELTA schickt und so die Liste der hinzuzufügenden Objekt-IDs nur ein Element enthält.
Für die zu entfernenden Objekt-IDs wird das Feld *Anfrage* eines Objekt-Positions-Cache Eintrags für die entsprechende Objekt-ID auf false gesetzt. Das bedeutet, dass bei der Verbreitung einer Timout-Nachricht für dieses Objekt sich der Suchbereich nur bezüglich der Objekt-Positons-Cache Einträge erweitert, jedoch keine Nachrichten an die mobilen Knoten geschickt wird. Das System schätzt also weiterhin intern die Objektposition ab, aber verteilt keine Suchaufträge, da keine Suchanfrage vorliegt. Weiterhin werden die mobilen Knoten benachrichtigt, nicht mehr nach den Objekten zu suchen, deren IDs in der Liste der zu entfernenden Objekt-IDs enthalten ist. Dies begründet sich darin, dass die Wurzel in die Liste der zu entfernenden Objekt-IDs nur solche IDs einfügt, für die momentan keine Anfrage mehr existiert. Weiterhin passt der Zell-Knoten seine Lookup-Tabelle für die mobilen Knoten an (siehe Algorithmus 13).

Objektfund

Erhält ein Zell-Knoten eine Objektfund-Nachricht von einem mobilen Knoten (siehe Algorithmus 14) seiner Zelle, so prüft er zunächst für jede Objekt-ID, ob ein Objekt-Positions-Cache Eintrag für vorliegt.
Falls ja, sendet der Zell-Knoten an alle Nachbarn deren Grenz-Zustand laut dem Objekt-Positions-Cache Eintrag auf VERTEILT steht eine Nachricht vom Typ GRID_REMOVE (siehe Algorithmus 13). Diese enthält die ID des Zell-Knoten als Sender der Nachricht und die Objekt-ID. Der empfangende Nachbar-Zell-Knoten entfernt die Objekt-ID aus seiner lokalen Liste $L_{ObjektID}$, informiert jedoch nicht die mobilen Knoten. Diese können das Objekt aus Sicht des System in dieser Zelle nicht finden, aber es macht nichts, wenn sie trotzdem nach dem Objekt suchen. Erst wenn keine Suchanfrage mehr existiert müssen die mobilen Knoten informiert werden. Im nächsten Schritt holt sich der Nachbar-Zell-Knoten den Objekt-Positions-Cache Eintrag

Algorithmus 13 Gitter-Ansatz: Zell-Knoten bekommt Nachricht vom Typ WURZEL_AUFTRAGS_LISTE_DELTA

procedure *Empfange(nachricht)*

```
 1: for all e ∈ nachricht.IDsHinzufuegen do
 2:   if e.id ∈ C_ObjPos then
 3:     C_ObjPos[e.id].Anfrage := true;
 4:   end if
 5:   L_ObjektID := L_ObjektID ∪ {e.id};
 6:   nachrichtAnMK.typ = AUFTRAGS_LISTE_DELTA;
 7:   nachrichtAnMK.fuegeHinzu(e.id);
 8:   sendeAnMobileKnoten(nachrichtAnMK);
 9:   for all g ∈ T_MK do
10:     g.fuegeHinzu(e.id);
11:   end for
12: end for
13: for all e ∈ nachricht.IDsEntfernen do
14:   if e.id ∈ C_ObjPos then
15:     C_ObjPos[e.id].Anfrage := false;
16:   end if
17:   L_ObjektID := L_ObjektID ∖ {e.id};
18:   nachrichtAnMK.typ = AUFTRAGS_LISTE_DELTA;
19:   nachrichtAnMK.fuegeZuEntferneHinzu(e.id);
20:   for all g ∈ T_MK do
21:     if e.id ∈ g.ObjektIDs then
22:       sende(nachrichtAnMK, g.IDMobilerKnoten);
23:     end if
24:     g.entferne(e.id);
25:   end for
26: end for
```

end procedure

für die Objekt-ID, welchen er garantiert besitzt. Danach schickt er an alle Nachbar-Zell-Knoten, außer von dem er die Nachricht bekommen hat, deren Grenz-Zustand auf VERTEILT steht ebenfalls eine Nachricht vom Typ GRID_REMOVE. Diese enthält nun die ID des Nachbar-Zell-Knoten als Sender und wiederum die Objekt-ID. Nach dem Senden entfernt der Nachbar-Zell-Knoten den Objekt-Positions-Cache Eintrag.

Zurück zum Zell-Knoten der die Objektfund-Nachricht erhalten hat und für die gerade betrachtete Objekt-ID einen Objekt-Positions-Cache Eintrag besitzt. Die abgeschätzte Position des Objekts ergibt aus dem Schnitt der im Cache gespeicherten Position und der in der Objektfund Nachricht enthaltenen Position, welche zuvor in ein Rechteck umgerechnet wird (siehe Algorithmus 6). Vor dem Schnitt der beiden Rechtecke werden sie jeweils zusätzlich noch bezüglich der vergangenen Zeit angepasst,

Algorithmus 14 Gitter-Ansatz: Zell-Knoten bekommt Nachricht vom Typ OBJEKT_FUND

procedure *Empfange*(*nachricht*)

```
 1: for all e ∈ nachricht.ObjektIDs do
 2:   if e.id ∈ C_ObjPos then
 3:     g := C_ObjPos[e.id];
 4:     nachrichtR.Typ:=GRID_REMOVE;
 5:     nachrichtR.Sender := this.id;
 6:     nachrichtR.ObjektID := e.id;
 7:     for all a ∈ Nachbarknoten do
 8:       if g.GrenzZustand[a] = VERTEILT then
 9:         sende(nachricht, a);
10:       end if
11:     end for
12:     temp1 := KoordinateZuPosition(nachricht,jetzt());
13:     temp2 := PasseAn(g.Position,g.Zeitstempel,jetzt());
14:     g.Position := Schnitt(temp1,temp2);
15:   else
16:     nachrichtB.Typ := GRID_BROADCAST_REMOVE;
17:     nachrichtB.ObjektID := e.id;
18:     gridBroadcast(nachrichtB);
19:     g.id := e.id;
20:     g.Position := KoordinateZuPosition(nachricht,jetzt());
21:     g.setzeGrenzZustände(RELEVANT);
22:   end if
23:   g.Zeitstempel := jetzt;
24:   g.Timeout := -1;
25:   while g.Timeout ≤ 0&&g.GrenzZustandEx(RELEVANT) do
26:     g := berechneTimeout(g);
27:     if g.Timeout <= 0 then
28:       for all a ∈ Nachbarn do
29:         if g.GrenzZustand[a] = AKTIV then
30:           g.GrenzZustand[a] := VERTEILT;
31:           sendeTimeout(g,a);
32:         end if
33:       end for
34:     end if
35:   end while
36: end for
```

end procedure

Algorithmus 15 Gitter-Ansatz: Timeout-Berechnung

ObjektCacheEintrag **function** *berechneTimeout*(*eintrag*)

```
1:  minTimeout := ∞;
2:  for all a ∈ Nachbarknoten do
3:    if eintrag.GrenzZustand[a] = RELEVANT then
4:      if a = Nachbar[oben] then
5:        timeout[oben]:=((Z.ymax-eintrag.ymax)/v_max)-(jetzt()-eintrag.Zeitstempel);
6:      else if a = Nachbar[unten] then
7:        timeout[unten]:=((eintrag.ymin-Z.ymin)/v_max)-(jetzt()-eintrag.Zeitstempel);
8:      else if a = Nachbar[links] then
9:        timeout[links]:=((eintrag.xmin-Z.xmin)/v_max)-(jetzt()-eintrag.Zeitstempel);
10:     else
11:       timeout[rechts]:=((Z.xmax-eintrag.xmax)/v_max)-(jetzt()-
          eintrag.Zeitstempel);
12:     end if
13:   end if
14: end for
15: for all a ∈ oben, unten, links, rechts do
16:   if timeout[a] < minTimeout then
17:     minTimeout := timeout[a];
18:   end if
19: end for
20: eintrag.Timeout := minTimeout;
21: for all a ∈ oben, unten, links, rechts do
22:   if timeout[a] = minTimeout then
23:     eintrag.GrenzZustand[Nachbar[a]] := AKTIV;
24:   end if
25: end forreturn eintrag;
end procedure
```

d.h. gegebenenfalls vergrößert. Für die im Objekt-Positions-Cache gespeicherte Position ist der Zeitpunkt zu dem die Position vom System als aktuell angesehen wurde im Feld *Zeitstempel* gespeichert. Genauso enthält die Objektfundnachricht einen Zeitstempel. Die Funktionen für die Anpassung und den Schnitt achsenparalleler Rechtecke, wurden bereits im Kapitel für den hierarchischen Ansatz angegeben (siehe Algorithmen 7 und 4).

Das resultierende Rechteck $R_{Schnitt}$ ist nun aus Sicht des Systems die neue Position des Objekts. Der Zell-Knoten legt einen neuen, temporären Objekt-Positions-Cache Eintrag e_{neu} an. Dieser neue Eintrag wird nun schrittweise vervollständigt. Zunächst wird e_{neu}.Position gleich $R_{Schnitt}$ gesetzt. Das Feld *id* von e_{neu} wird auf die momentan betrachtete Objekt-ID gesetzt. Die Grenz-Zustände von e_{neu} haben alle den Wert RELEVANT. Das Feld e_{neu}.Zeitstempel bekommt den jetzigen Zeitwert und das Feld e_{neu}.Anfrage erhält den Wert wahr. Der Eintrag e_{neu} wird nun der Funktion zur Berech-

Algorithmus 16 Gitter-Ansatz: Zell-Knoten bekommt Nachricht vom Typ GRID_REMOVE

procedure *Empfange(nachricht)*

```
1: L_ObjektIDs := L_ObjektIDs \ {nachricht.ObjektID};
2: e := C_ObjPos[nachricht.ObjektID];
3: nachrichtR.Typ := GRID_REMOVE;
4: nachrichtR.Sender := this.id;
5: nachrichtR.ObjektId := nachricht.ObjektID;
6: for all a ∈ Nachbarknoten do
7:    if e.GrenzZustand[a] = VERTEILT then
8:       sende(nachrichtR, a);
9:    end if
10: end for
11: C_ObjPos.entferne(nachricht.ObjektID);
```

end procedure

nung der Timeout-Wert für die Zellgrenzen übergeben. Diese beachtet nur Grenzen deren Zustand den Wert RELEVANT besitzen. Ist der von der Funktion gesetzte Timout-Wert negativ, so wurde bereits eine der Zellgrenzen überschritten. Die betreffende Zellgrenze wird dadurch identifiziert, dass ihr Zustand im Eintrag e_{neu} auf AKTIV gesetzt wurde. An den zugehörigen Nachbar-Zell-Knoten wird nun eine Timout-Nachricht geschickt, welche die Werte e_{neu}.id, e_{neu}.Position, e_{neu}.Zeitstempel und e_{neu}.Anfrage enthält. Der Zell-Knoten wiederholt diese Schritt so lange bis entweder der Timout-Wert nicht negativ ist oder kein Grenze mehr existiert deren Zustand den Wert RELEVANT besitzt. Abschließend wird der Eintrag e_{neu} zum Objekt-Positions-Cache hinzugefügt und ersetzt den alten Wert.

Existiert beim Erhalt einer Objektfund-Nachricht kein Cache-Eintrag für die betrachtete Objekt-ID, so wird eine Nachricht vom Typ GRID_BROADCAST_REMOVE an alle horizontalen und vertikalen Nachbarn geschickt, welche die Objekt-ID enthält. Diese wird, wie bereits beschrieben, so verteilt, dass jeder Zell-Knoten die Nachricht genau ein einziges mal bekommt. Die abgeschätzte Position $R_{Schnitt}$ entspricht hier nur dem aus der Position der Objektfund-Nachricht berechneten Rechteck. Die restlichen Schritte entsprechen dem Fall bei dem ein Objekt-Positions-Cache Eintrag vorliegt.

Jedes Hinzufügen oder Entfernen eines Eintrags des Objekt-Positions-Cache eines Zell-Knotes wird zusätzlich der Wurzel mittels einer Nachricht vom Typ OBJEKT_CACHE_DELTA gemeldet. Dies wurde in den Algorithmen nicht vermerkt, wir nehmen es aber implizit an. Die Nachricht enthält die ID des Zell-Knoten und wieder 2 Listen mit Objekt-IDs. Die eine enthält die Objekt-IDs der neu hinzugefügten Objekt-Positions-Cache Einträge, die andere enthält die Objekt-IDs der entfernten Einträge. Beim Erhalt der Nachricht aktualisiert die Wurzel ihre Lookup-Tabelle wie in Algorithmus 17 angegeben.

Algorithmus 17 Gitter-Ansatz: Wurzel bekommt Nachricht vom Typ OBJEKT_CACHE_DELTA

procedure *Empfange*(*nachricht*)

1: **for all** $e \in nachricht.fuegeHinzu$ **do**
2: T_{ObjC}[nachricht.Sender].fuegeHinzu(e);
3: **end for**
4: **for all** $e \in nachricht.entferne$ **do**
5: T_{ObjC}[nachricht.Sender].entferne(e);
6: **end for**

end procedure

Timeout

Wir betrachten in diesem Abschnitt sowohl die Reaktion auf eine Timeout-Nachricht, als auch auf einen internen Timeout eines Objekt-Positions-Cache Eintrags.
Erhält ein Zell-Knoten eine Timeout-Nachricht (vergleiche Algorithmus 18), so enthält diese die Objekt-ID, die letzte bekannte Position des Objekts,den Zeitpunkt zu dem diese Position gültig war und ob momentan eine Suchanfrage für das Objekt existiert. Gibt es eine Suchanfrage, so verteilt der Zell-Knoten an alle mobilen Knoten in seiner Zelle, die noch nicht nach dem Objekt suchen einen Suchauftrag und aktualisiert die Lookup-Tabelle der mobilen Knoten (siehe Algorithmus 18). Außerdem fügt er die ID in seine Liste der in der Zelle gesuchten Objekt-IDs ein, damit neu ins System kommende mobile Knoten oder zur Zelle wechselnde Knoten auch nach diesem Objekt suchen.

Ist keine aktive Suchanfrage vorhanden, so bewirkt dieser Mechanismus eine Anpassung des Suchbereichs ohne negativen Einfluss auf die mobilen Knoten. Danach setzt er mittels der in Algorithmus 11 angegebenen Funktion die Werte für die Zustände der Grenzen und berechnet dann mittels Algorithmus 15 den Timeout-Wert. Ergibt sich dabei ein negativer Wert für den Timeout, so wird die Timeout-Nachricht an den Nachbarknoten weitergeleitet, der zur Grenze gehört, die überschritten wurde. Der Zustand der Grenze wird auf den Wert VERTEILT gesetzt und der Timeout neu berechnet. Dies geschieht wieder so lange bis entweder kein negativer Timeout mehr auftritt oder es keine Grenzen mehr gibt deren Zustand den Wert RELEVANT besitzt. Im Fall eines Timeouts der durch einen Objekt-Positions-Cache Eintrag ausgelöst wird, wird allen Nachbarknoten dessen Grenz-Zustand den Wert AKTIV hat eine Timout-Nachricht geschickt und der jeweilige Grenz-Zustand wird auf VERTEILT aktualisiert. Existiert danach noch ein Grenz-Zustand dessen Wert auf RELEVANT steht, so wird der Timeout-Wert neu berechnet (siehe Algorithmus 19).

Zellwechsel

Wechselt ein mobiler Knoten MK_l von Zelle Z_i zu Zelle Z_j, so sendet er eine Nachricht vom Typ ZELL_WECHSEL an den Zell-Knoten ZK_j. Diese enthält nur die ID des mo-

Algorithmus 18 Gitter-Ansatz: Zell-Knoten bekommt Timeout-Nachricht

procedure *Empfange*(*nachricht*)

```
1: if nachricht.Anfrage = true then
2:    L{ObjektIDs} := L{ObjektIDs} ∪ {nachricht.ObjektID};
3:    for all e ∈ T_MK do
4:       if nachricht.ObjektID ∉ e.ObjektIDs then
5:          sende(nachricht.ObjektID,e.IDMobilerKnoten);
6:          e.ObjektIDs := e.ObjektIDs ∪ {nachricht.ObjektID};
7:       end if
8:    end for
9: end if
10: cacheEintrag := TimeoutNachrichtSetzeGrenzen(nachricht.Sender,cacheEintrag);
11: cacheEintrag.id := nachricht.ObjektID;
12: cacheEintrag.Position := nachricht.Position;
13: cacheEintrag.Zeitstempel := nachricht.Zeitstempel;
14: cacheEintrag.Timeout := -1;
15: while cacheEintrag.Timeout ≤ 0&&cacheEintrag.GrenzZustandEx(RELEVANT)
    do
16:    cacheEintrag := berechneTimeout(cacheEintrag);
17:    if cacheEintrag.Timeout <= 0 then
18:       for all a ∈ Nachbarn do
19:          if cacheEintrag.GrenzZustand[a] = AKTIV then
20:             cacheEintrag.GrenzZustand[a] := VERTEILT;
21:             sendeTimeout(cacheEintrag,a);
22:          end if
23:       end for
24:    end if
25: end while
```

end procedure

Algorithmus 19 Gitter-Ansatz: Zell-Knoten bekommt internen Timeout

procedure *Timeout*(*cacheEintrag*)

```
1: for all a ∈ Nachbarknoten do
2:    if cacheEintrag.GrenzZustand[a] = AKTIV then
3:       sendeTimeout(cacheEintrag,a);
4:       cacheEintrag.GrenzZustand[a] := VERTEILT;
5:    end if
6: end for
7: if cacheEintrag.GrenzZustandEx(RELEVANT) then
8:    cacheEintrag := berechneTimeout(cacheEintrag);
9: end if
```

end procedure

bilen Knoten MK_l. ZK_j fragt daraufhin die Liste der an MK_l gesendeten Objekt-IDs bei ZK_i ab. Die Antwort von ZK_i enthält die Liste L_{MK_l}, welche den momentan von MK_l gesuchten Objekt-IDs entspricht, zusammen mit der ID des mobilen Knoten MK_l, damit die Antwort zugeordnet werden kann. Der Zell-Knoten ZK_j vergleicht darauf hin die bisher von MK_l gesuchten Objekt-IDs mit der in seiner Zelle gesuchten Liste $L_{ObjektIDs}$ und schickt $L_{ObjektIDs} \setminus L_{MK_l}$ an den mobilen Knoten, falls die Differenz nicht leer ist.

Periodisch ausgeführte Aktionen

Die periodisch ausgeführten Aktionen entsprechen denen bereits beim Basis-Ansatz eingeführten Aktionen. Für den Fall, dass eine Suchanfrage ausläuft, schickt die Wurzel eine Nachricht vom Typ WURZEL_AUFTRAGS_LISTE_DELTA, bei der die Liste der zu entfernenden IDs die ID des den nicht mehr gesuchten Objekts enthält an alle Zell-Knoten. Darauf hin werden die mobilen Knoten informiert nicht mehr nach dem Objekt zu suchen. Außerdem passt jeder Zell-Knoten seine Lookup-Tabelle für die mobilen Knoten an und setzt bei einem gegebenen Objekt-Positions-Cache Eintrag das Feld *Anfrage* auf false (siehe Algorithmus 13).

Zusammenfassung

Der in diesem Abschnitt vorgestellte Ansatz ist im Vergleich zum Basis-Ansatz recht komplex. Er versucht jedoch Suchaufträge nur dort zu verteilen wo es aus Sicht des Systems Sinn macht. Dabei lässt sich der Suchbereich aufgrund der Gitterstruktur recht gut an die abgeschätzte Position anpassen und es wird erwartet, dass dies um so besser ist, je mehr Zellen vorhanden sind, sprich je feiner das Dienstgebiet unterteilt wird. Durch die pessimistische Abschätzung mit der maximalen Objektgeschwindigkeit sollten zudem keine Objektfund-Nachrichten verloren gehen. Insgesamt entspricht dieser Ansatz unserer Anforderung die übertragenen Daten zwischen mobilen Knoten und der Infrastruktur bei gleichbleibender Effizienz zu minimieren.

4.4.3 Kompression und Bloom-Filter

Ein bisher nicht betrachteter Aspekt ist die Kompression von Daten. Eine probabilistische Datenstruktur die sich hierfür zunächst scheinbar anbietet sind Bloom-Filter. Dabei handelt es sich um eine Datenstruktur zur kompakten Darstellung von Mengen, insbesondere lässt sich in der Grundvariante nur testen, ob ein Element in der Menge enthalten ist oder nicht. Mit einer sehr geringen Wahrscheinlichkeit werden auch Elemente als enthalten gemeldet, für die das nicht zutrifft. Eingefügte Elemente werden immer als enthalten gemeldet, das heißt die repräsentierte Menge ist im schlimmsten Fall größer, aber niemals zu klein. Da Bloom-Filter für Analysen zwar implementiert wurden, sie jedoch schlussendlich nicht eingesetzt wurden, beschreiben wir keine nähern mathematischen Details. Betrachtet man die zuvor vorgestellten Ansätze, so werden von den Koordinatoren bzw. Zell-Knoten jeweils nur "Deltas" an die mobilen Knoten geschickt, die meist relativ klein sind. Hierfür eignen sich Bloom-Filter kaum, da diese wie empirisch mittels einer Implementierung herausgefunden wurde sich erst ab ungefähr 10 Objekt-IDs ammortisieren.

Außer beim Zellwechsel besteht daher was die Richtung von Koordinatoren zu mobilen Knoten angeht wenig Spielraum für Optimierungen. Durch frühe Simulationen der beschriebenen Ansätze wurde herausgefunden, dass dabei die Abweichung der gesuchten IDs zwischen zwei Zellen meist verhältnismäßig gering sind, zumindest bei einer hohen bis mittleren Rate von Suchanfragen. Daher wurde dies nicht weiter verfolgt.
Eine Möglichkeit Bloom-Filter von mobilen Knoten zu Zell-Knoten zu benutzten wurde auch gefunden. Dabei ist das Grundproblem, dass allein mit der Kenntnis eines Bloom-Filters nicht klar ist welche Elemente er enthält. Insbesondere ist es nicht möglich alle Objekt-IDs darauf zu testen ob sie enthalten sind. Jedoch stellt sich heraus, dass die erwartet Menge an Objekt-IDs durch den Zell-Knoten genau der Menge der IDs der in der Zelle gesuchten Objekte entspricht. Diese ist bekannt und kann getestet werden. Das Problem dabei ist nur, dass sich für eine einzelne Objektfund-Nachricht, die meist maximal um die drei bis vier Objekt-IDs enthält sich ein Bloom-Filter nicht lohnt. Daher muss der mobile Knoten mehrer Objektfunde sammeln und diese zusammen in einem Bloom-Filter übertragen.
Der werden für jeden Objektfund die Koordinaten im Klartext hinzugefügt. Doch nun durch die Verwendung eines einzigen Bloom-Filters ergibt sich das Problem der Zuordnung der Objekt-IDs zu der Menge der gemeldeten Koordinaten. Um dies zu lösen nummeriert man die Einträge für die Koordinaten durch und fügt in den Bloom-Filter anstatt der Objekt-ID, die ID konkateniert mit der Position der zugehörigen Koordinaten. Sprich wenn o_i zum dritten Eintrag gehört, so würde ID(o_i).3 eingefügt werden. Der Koordinator muss dann für alle IDs in seiner lokalen Liste und für die Anzahl aller Einträge testen ob ObjektID.PositionEintrag im Bloom-Filter enthalten ist. Um zu vermeiden, dass durch die probabilistische Natur eines Bloom-Filters etwa einem Objekt eine falsche Position zugeordnet wird, testet der mobile Knoten zuvor den Bloom-Filter mit allen Kombinationen von Objekt-Id und Positionen. So lässt sich, bis auf eine Änderung der Menge der gesuchten Objekt-IDs während die so konstruierte Objekfund-Nachricht unterwegs ist, gerantieren, dass keine Objekte gemeldet werden, die so nicht von einem mobilen Knoten gefunden wurde. Da wir jedoch zugunsten unserer Suchbereichsanpassung keine großen Verzögerungen beim Senden von Objektfundnachricht möchten, haben wir die hier beschriebene Methode nicht weiter verfolgt.

5 Implementierung

Um die in Kapitel 4 vorgestellten Ansätze zu evaluieren, wurden diese für einen Netzwerksimulator implementiert. Für die Simulation wurde der Netzwerksimulator ns2 in der Version 2.34 benutzt. Dieser verwendet sowohl C++ als auch die Skriptsprache Tcl.
Die eigentlichen Systemkomponenten sind in C++ implementiert. Tcl-Skripte werden dazu verwendet, um die Komponenten untereinander zu verbinden und die benötigten Strukturen, beispielsweise die Gebietsunterteilung, aufzubauen. Weiterhin werden im Tcl-Skript die Parameter der Simulation konfiguriert.
Im Folgenden werden wir zunächt allgemein den Aufbau des Systems und dessen Komponenten beschreiben. Danach die Realisierung der Komponenten im Speziellen und deren Verknüpfung für jeden der in Kapitel 4 beschriebenen Ansätze.

5.1 Architektur

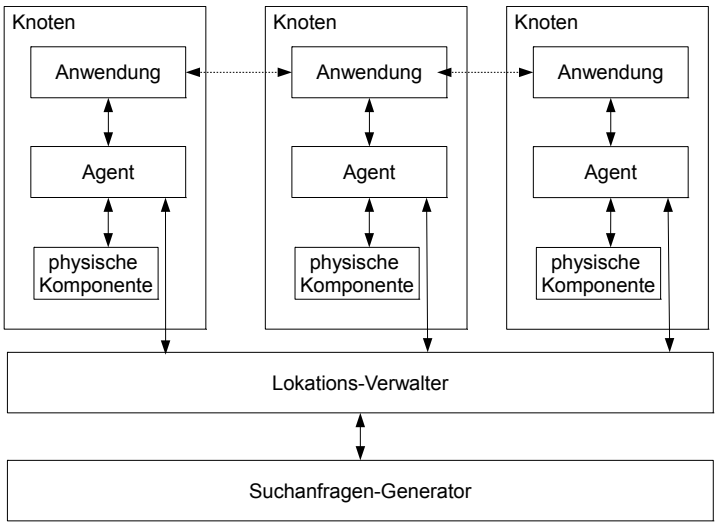

Abbildung 5.1: Architektur für die Kommunikation zwischen den Knoten in der Simulation

Für die Simulation verwenden wir die in Abbildung 5.1 dargestellte Architektur. Die einzelnen Knoten kommunizieren untereinander allgemein mittels des Lokations-Verwalters, bei dem sich jeder Knoten zu Anfang der Simulation registriert und der so

die Position aller Knoten in der Simulation kennt. So ist es beispielsweise möglich zu einem mobilen Knoten den zugehörigen Koordinator zu finden und ihm eine Nachricht zu schicken. Etwas genauer läuft die Interaktion so ab, dass die Anwendungs-Komponente die zu sendende Nachricht an den mit ihr verbundenen Agent übergibt, welcher die Nachricht um Positionsinformationen erweitert und dann diese beiden Informationen an den Lokations-Verwalter weiterreicht. Für das Beispiel eines mobilen Knotens würde der Lokationsverwalter dann anhand der Positionsinformation den zugehörigen Koordinator suchen in dessen Zelle sich der mobile Knoten befindet und danach dessen Agent die Nachricht übergeben. Dieser reicht die Nachricht dann an die Anwendungs-Komponente des Koordinators weiter, welche diese dann verarbeitet.

Der Anfragen-Generator dient zur Generierung von Suchanfragen, welche mittels des Lokations-Verwalters an den für Anfragen zuständigen Knoten übergeben werden. Um dies zu ermöglichen, wird dieser Knoten dem Lokations-Verwalter zu Beginn der Simulation explizit bekannt gemacht.

Wir beschreiben nun die einzelnen in Abbildung 5.1 dargestellten Systemkomponenten genauer.

5.2 Knoten

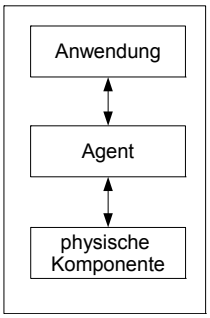

Abbildung 5.2: allgemeiner Aufbau eines Knoten in der Simulation

Ein Knoten wird in der Simulation allgemein wie in Abbildung 5.2 modelliert. Er besteht demnach aus drei Komponenten: Anwendung, Agent und physische Komponente. Außerdem besitzt jeder Knoten eine global eindeutige ID. Mobile Objekte, mobile Knoten und alle Knoten der Infrastruktur sind nach diesem Muster aufgebaut. In den folgenden Abschnitten beschreiben wir zunächst die Grundfunktionalität jeder dieser drei Komponenten. Von diesem Grundgerüst eines Knotens leiten wir dann spezifische Ausprägungen verschiedener Knoten ab, die mobile Objekte, mobile Knoten und Infrastruktur-Knoten realisieren.

Anwendung

```
class BasicApp: public Application {

        virtual void process_data(int size , AppData* data)=0;
        ...

        Agent* agent_;

};
```

Listing 5.1: Aufbau der Klasse BasicApp

Die Anwendungs-Komponente interagiert mit dem Agent um Nachrichten zu versenden und zu empfangen. Ihre Grundfunktionalität wird in der Klasse *BasicApp* festgelegt. Sie besitzt im Allgemeinen nur eine Schnittstelle, welche dazu dient Anwendungsdaten zu empfangen. Die zugehörige Methode lautet *BasicApp::process_ data(int size, AppData* data)* und wird vom jeweils darunterliegenden Agent aufgerufen. Innerhalb dieser Methode wird die Nachricht zunächst anhand ihres Typ-Felds identifiziert und dann anschließend typspezifisch innerhalb der Anwendungs-Komponente weiterverarbeitet (siehe Auflistung 5.2). Um Nachrichten zu

```
virtual void process_data(int size , AppData* data){
        if(data->type_ == APPDATA_TYPE1){
                processAppDataType1(size,(AppDataType1*) AppData);
        } else if(data->type_ == APPDATA_TYPE2){
                processAppDataType2(size,(AppDataType2*) AppData);
        } else if
                ...
        }
};
```

Listing 5.2: beispielhafte Verarbeitung einer Nachricht

versenden wird die Methode *BasicAgent::send(int size, AppData* data)* verwendet.

Agent

Der Agent bietet generell zwei Schnittstellen. Diese werden in der Klasse *BasicAgent* definiert.

Die Anwendung kann mittels der Methode *BasicAgent::send(int size, AppData* data)* Anwendungsdaten verschicken. Dabei erweitert der Agent die Anwendungsdaten um Positionsinformation in Form eines Verweises auf die dem Knoten zugrundeliegenden Instanz der Klasse MobileNode, welche die physische Komponente realisiert. Diese

```
class BasicAgent : public Agent {

        virtual void recv(int size, AppData* data);
        virtual void sendmsg(int nbytes, AppData* data,
                             const char *flags = 0);
        ...
        Application* app_;
        MobileNode* linkToOwnNode;

};
```

Listing 5.3: Aufbau der Klasse BasicAgent

beinhaltet die exakte Position des Knoten. Mit dieser Information ist es möglich den Empfänger der Nachricht zu bestimmen, was die Aufgabe des Lokations-Verwalters ist.

Die Methode *BasicAgent::recv(packet p*, Handlder h*)* ist die ursprünglich von ns2 vorgesehene Methode um Pakete zu empfangen. Da wir jedoch kein echtes Protokoll zwischen zwei Agent-Instanzen benutzen und somit nur ein Typ von Paketen existiert, können die Anwendungsdaten direkt übergeben werden. Somit lautet die Signatur der Methode *BasicAgent::recv(int size, AppData* data)*. Innerhalb dieser Methode wird im Endeffekt *BasicApp::recv(int size, AppData* data)* aufgerufen um die Daten an die Anwendung weiterzuleiten. Mit welcher Instanz von BasicApp ein Agent verknüpft ist wird über das Tcl-Skript geregelt und genauso wird der Anwendung ihr zugehöriger Agent bekannt gemacht.

Physische Komponente

```
class MobileNode : public Node
{
        inline int nodeid() { return nodeid_;}

        inline double X() { return X_; }
        inline double Y() { return Y_; }

        void update_position();
        ...

};
```

Listing 5.4: Auszug aus der ns2-Klasse MobileNode

Diese Komponente eines Knotens zeichnet sich dadurch aus, dass sie die exakte Position des Knotens in Form einer im Allgemeinen dreidimensionalen kartesischen Koordinate

enthält und sich über dies hinaus die Position über die Zeit verändern kann. Somit ist die Bewegung eines Knotens möglich. Wir beschränken uns in der Simulation, wie in unseren Ansätzen, auf den zweidimensionalen Fall und verwenden nur die x- und y-Koordinate, führen also implizit eine senkrechte Projektion auf die x-y-Ebene durch.
In ns2 gibt es zur Realisierung dieser Komponente bereits eine Klasse MobileNode, welche wir verwenden (siehe Auflistung 12. Somit wird die physische Komponente durch eine Instanz der Klasse MobileNode repräsentiert. Im Tcl-Skript kann dabei die Position direkt gesetzt werden (*$phyischeKomponente set X_* bzw. *$phyischeKomponente set Y_*) oder ein Ziel und die Geschwindigkeit mit der sich der Knoten dorthin bewegen soll angegeben werden (*$phyischeKomponente setdest ZielX ZielY Geschwindigkeit*).

5.3 Lokations-Verwalter

```
class LocationManager : public NsObject {

        void getObjectsInRange(MobileNode* mobileNode,
                        double radius,
                        vector<MobileObject*>* objectList);

        void sendToBS(MobileNode* mobileNode, int nbytes,
                        AppData* data);
        void sendToMobileNodes(MobileNode* BS, int nbytes,
                        AppData* data);
        void unicastToMobileNodeId(MobileNode* BS, int nbytes,
                        AppData* data, int NodeId);
        int getBSId(MobileNode* mobileNode);

        ...

};
```

Listing 5.5: Aufbau der Klasse LocationManager

Der Lokations-Verwalter hat im Wesentlichen die folgenden Funktionen:

- Lokationsverwaltung von mobilen Knoten, mobiler Objekte und Infrastruktur-Knoten
- Auffinden mobiler Objekte in Reichweite eines mobilen Knoten
- Kommunikation zwischen Infrastruktur-Knoten und mobilen Knoten

Die Funktionalitäten des Lokations-Verwalters werden in der Klasse *LocationManager* definiert. Von der Klasse LocationManager gibt es in der Simulation genau eine Instanz, welche global zugänglich ist. Sie entspricht dem in Abbildung 5.1 gezeigten Lokations-Verwalter.

Lokationsverwaltung

Der Lokations-Verwalter enthält eine Art Datenbank, bei der sich jeder Knoten registriert. Das bedeutet, dass allgemein die Komponenten eines Knotens, also Anwendung, Agent und physische Komponente, dort bekannt gemacht werden. Die Registrierung erfolgt über das TCl-Skript, wobei dabei unterschieden wird ob es sich um ein mobiles Objekt, einen mobilen Knoten, einen Zell-Knoten oder einen Infrastruktur-Knoten handelt, welcher nicht mit mobilen Knoten kommuniziert. Beispielsweise registriert sich ein mobiler Knoten über *$locdb register-mn $node $MN_ agent*.
Durch die physische Komponente eines Knoten, welche letztlich ein Verweise auf eine Instanz der Klasse *MobileNode* ist, lässt sich zu jedem Zeitpunkt feststellen, an welcher Position sich ein Knoten befindet. Dazu wird die von der Klasse *MobileNode* bereitgestellte Methode zur Aktualisierung der Position eines Knotens benutzt.

Auffinden mobiler Objekte

Um einem mobilen Knoten das Suchen mobiler Objekte zu ermöglichen, stellt der Lokations-Verwalter die Methode *LocationManager::getObjectsInRange(MobileNode* mobileNode, double radius, vector<MobileObject*>* objectList)* bereit. Diese Methode wird vom Agent eines mobilen Knoten aufgerufen, um die mobilen Objekte in Sensorreichweite zu erhalten. Der Agent ermöglicht dann wiederum der Anwendung diese gefundenen Objekte abzufragen.
Zuer Bestimmung der Objekte in Reichweite werden zunächst die Positionen der mobilen Objekte und des mobilen Knotens aktualisiert (vergleiche Algorithmus 20). Anschließend wird für alle Objekte überprüft, ob der euklidsche Abstand zwischen Objektposition und der Position des mobilen Knotens kleiner gleich dem angegebenen Radius ist, welcher der Sensorreichweite entspricht. Ist dies der Fall, so wird das Objekt in die Liste der Objekte in Reichweite übernommen.

Algorithmus 20 Lokations-Verwalter: Auffinden mobiler Objekte in Reichweite eines mobilen Knoten

procedure *findeObjekteInReichweite(mobilerKnoten,radius,objektListe)*

```
1: aktualisiereObjektPositionen();
2: mobilerKnoten.aktualisierePosition();
3: for all o ∈ MobileObjekte do
4:    if euklidscherAbstand(o, mobilerKnoten) ≤ radius then
5:       objektListe := objektListe ∪ {o};
6:    end if
7: end for
end procedure
```

Kommunikation

Da wir davon ausgehen, dass die Kommunkation zwischen mobilen Knoten und Knoten der Infrastruktur über das Mobilfunknetz läuft, beschränken wir uns darauf den passenden Empfänger einer Nachricht zu ermitteln und dann die Nachricht direkt an diesen zu übergeben. Dabei gibt es zwei Kommunikationsrichtungen: Von einem mobilen Knoten zum Koordinator und die entgegengesetzte Richtung. Daher besteht die Aufgabe des Lokations-Verwalters zum einen darin zu einem mobilen Knoten den zugehörigen Koordinator, genauer Zell-Knoten, zu finden, welcher den Bereich verwaltet in dem sich der mobile Knoten gerade befindet und an diesen dann die Nachricht zu übergeben. Zum anderen ermöglicht er es einem Zell-Knoten entweder an alle mobilen Knoten innerhalb seiner Zelle eine Nachricht zu schicken oder per Unicast mit einem bestimmten mobilen Knoten zu kommunizieren. Letzteres geschieht über die eindeutige ID eines Knoten und die Nachricht kommt nur an, falls sich der Knoten innerhalb der Zelle befindet. Daher gibt es drei verschiedene Methoden zum Senden von Nachrichten, die der Lokations-Verwalter bereitstellt.

LoationManager::sendToBS(MobileNode mobileNode, int nbytes, AppData* data)*
wird vom Agent eines mobilen Knoten aufgerufen, um Anwendungsdaten an den aktuelle zu seiner Zelle gehörenden Koordinator (Zell-Knoten) zu schicken. Die Abmessungen einer Zelle sind ein global bekannter Simulationsparameter und für jede Zelle gleich. Zusammen mit der Kenntnis der Positionen der Zell-Knoten, welche dem Mittelpunkt der jeweiligen Zelle entsprechen, lässt sich der zuständige Knoten, wie in Algorithmus 21 angegeben, bestimmen.

Algorithmus 21 Lokations-Verwalter: Senden einer Nachricht an den zuständigen Zell-Knoten

procedure *SendeAnZellKnoten*(*mobilerKnoten*, *grösse*, *daten*)

```
1: for all z ∈ ZellKnoten do
2:   if inRange(z, mobilerKnoten) then
3:     z.Agent.recv(grösse, daten)
4:     return;
5:   end if
6: end for
end procedure
```

Analog wird *LoationManager::sendToMobileNodes(MobileNode* mobileNode, int nbytes, AppData* data)* von einem Zell-Knoten benutzt, um eine Nachricht an die Knoten in seiner Zelle zu schicken (siehe Algorithmus 22).

Die Methode *LoationManager::unicastToMobileNodeId(MobileNode* BS, int nbytes, AppData* data, int NodeId)* wird verwendet um Anwendungsdaten an einen bestimmten mobilen Knoten zu schicken. Dabei lassen sich über die ID des mobilen Knoten direkt

Algorithmus 22 Lokations-Verwalter: Nachricht an alle mobilen Knoten einer Zelle schicken

procedure *SendeAnMobileKnoten*(*zellKnoten*, *grösse*, *daten*)

```
1: for all m ∈ MobileKnoten do
2:   if inRange(m, zellKnoten) then
3:     m.Agent.recv(grösse, daten)
4:   end if
5: end for
```

end procedure

die zugehörigen Komponenten des mobilen Knoten finden. Dadurch ist die Instanz des zum mobilen Knoten zugehörigen Agent bekannt und es folgt ein Aufruf der Methode *BasicApp::recv(int size, AppData* data)* dieser Instanz. Im Endeffekt bewirkt dies, dass die Nachricht bei der zugehörigen Anwendung des mobilen Knoten ankommt.
Die Funktion *LocationManager::getBSId(MobileNode* mobileNode)* wird vom Agent eines mobilen Knoten benutzt um die ID des Zell-Knotens zu erhalten, in dessen Bereich sich der Knoten gerade befindet. Hierdurch ist es möglich festzustellen, ob ein Zellwechsel vorliegt, oder ob sich der mobile Knoten außerhalb des Dienstgebiets befindet. In Letzterem Fall gibt die Funktion den Wert -1 zurück, als Zeichen dafür, dass es keinen zugeordneten Zell-Knoten gibt.

5.4 Mobiler Knoten

Ein mobiler Knoten besitzt alle drei der in Kapitel 5.2 vorgestellten Komponenten. Dabei erweitert er jedoch die Komponenten Agent und Anwendung durch Ableiten von den Klassen *BasicAgent* und *BasicApp*. Die entsprechenden Unterklassen heißen *MobileAgent* und *MobileApp*. Die physische Komponente wird durch die Klasse *MobileNode* realisiert.

MobileAgent

Wie in Auflistung 5.6 dargestellt, erweitert die Klasse MobileAgent die Klasse BasicAgent um knotenspezifische Methoden. Die Methode *MobileAgent::senseObjects()* und die Funktion *MobileAgent::getObjects()* dienen dazu es der Anwendung zu ermöglich die Suche nach mobilen Objekten zu veranlassen und die Liste der gefundenen Objekte abzufragen. Zur Realisierung wird auf die Methode *LocationManager::getObjectsInRange(MobileNode* mobileNode, double radius,vector<MobileObject*>* objectList)* des Lokations-Vewalter zurückgegriffen, welche es ermöglicht zu einer gegebenen Position, in Form eines Verweises auf eine Instanz von MobileNode, die Menge der Objekte, die sich innerhalb eines gegebenen Radius um die aktuelle Position des Knoten befinden, zu erhalten. Wann zuletzt nach mobilen Objekt gesucht wurde wird in *lastObjectSensing* vermerkt und kann von der Anwendung abgerufen werden.

```
class MobileAgent: public BasicAgent {
        void sendmsg(int nbytes, AppData* data, const char *flags);

        void senseObjects();
        vector<MobileObject*> getObjects();

        int getCurrentBSId();

        double getX();
        double getY();

        ...
        double lastObjectSensing;
        double cPosX;
        double cPosY;
        double lastGPSSensing;
        GPS_Timer* gpsTimer;

        LocationManager* locM;
};
```

Listing 5.6: Aufbau der Klasse MobileAgent

Außerdem wird die Positionsbestimmung dahingehende modelliert, dass der Agent in definierten Zeitabständen von der physischen Komponente die tatsächliche Positions abfragt und diese samt der Aktualisierungszeit vom Agent zwischenspeichert. Die Anwendung kann diese Koordinaten dann über die Methoden *MobileAgent::getX()* bzw. *MobileAgent::getY()* abfragen und analog den zugehörigen Zeitpunkt.

Mittels der Funktion *MobileAgent::getCurrentBSId()* kann die Anwendung die Id des momentan zuständigen Koordinators abfragen. Durch Vergleich mit der zuvor gemeldeten Id lässt sich damit ein Zellwechsel feststellen. Die Anwendung reagiert darauf dann mit einer Benachrichtigung über den Zellwechsel an den nun zuständigen Zell-Knoten. Realisiert wird die Funktion über den Lokations-Verwalter, welcher die zuvor erwähnte Funktion *LocationManager::getBSId(MobileNode* mobileNode)* dafür bereitstellt. Der Agent hat hierfür einen Verweis auf die Instanz des Lokations-Verwalters, welcher über das Tcl-Skript initialisiert wird.

Zudem implementiert der Agent die Methode *MobileAgent::sendmsg(int nbytes, AppData* data, const char *flags)* derart, dass er die Methode *LoationManager::sendToBS(MobileNode* mobileNode, int nbytes, AppData* data)* aufruft. Der Anwendung bleibt der Lokations-Verwalter somit verborgen.

MobileApp

```
class MobileApp : public BasicApp {

        void process_data(int size , AppData* data){
                if(data->type()==TRACKING_TASK_LIST){
                        processTaskList(data->size(),
                                        (TrackingDataTaskList*) data);
                } else if (data->type()==TRACKING_TASK_LIST_DELTA){
                        ...
        };

        void reportObjectsToBS();

        reportObjectsTimer* reportTimer;
};
```

Listing 5.7: Aufbau der Klasse MobileApp

Die Klasse *MobileApp* erweitert die Funktionalitäten der Klasse *BasicApp* insoweit, dass sie die Methode *BasicApp::process_data* spezifisch implementiert und zwar derart, dass je nach Typ einer Nachricht, wie im Kapitel zum Systementwurf beschrieben, reagiert wird. Falls der mobile Knoten beispielsweise eine Nachricht vom Typ TRACKING_TASK_LIST bekommt, so weiß er, dass diese die Ids der Objekte enthält die in der Zelle gesucht werden, in der er sich gerade befindet und speichert sie sich daher.
Außerdem wird ein Timer benutzt, welcher den Knoten periodisch dazu veranlasst nach

Algorithmus 23 MobileApp: mobile Objekte suchen und Fund melden

procedure *meldeObjektfund*()

1: **if** *Zellwechsel*() **then**
2: frageObjektIdListeAb();
3: **end if**
4: objekteInReichweite := MobilerAgent->holeListeDerObjekteInReichweite();
5: **for all** $o \in objekteInReichweite$ **do**
6: **if** $o.Id \in gesuchteObjektIds$ **then**
7: gefundeneObjekte := gefundenObjekte $\cup$ {o.Id};
8: **end if**
9: **end for**
10: **if** $gefundeneObjekte \neq \emptyset$ **then**
11: MobilerAgent->sende(gefundeneObjekte);
12: **end if**

end procedure

mobilen Objekten in seiner Umgebung zu suchen, was wie beschrieben über den Agent passiert, und dann gegebenenfalls den Fund an den zugehörigen Koordinator weiterzuleiten. Bevor die Ids der gefundenen Objekte mit den Ids der momentan gesuchten Objekte verglichen werden, prüft die Anwendung außerdem, ob ein Zellwechsel vorliegt und fragt gegebenenfalls zuvor die nun möglicherweise zusätzlich zu suchenden Objekte ab (vergleiche Algorithmus 23).

5.5 Mobile Objekte

```
class MobileObject : public MobileNode {

        bool isActive();
        ...
};
```

Listing 5.8: Aufbau der Klasse MobileApp

Mobile Objekte besitzen nur eine physische Komponente und weder Anwendungs-Komponente noch einen Agent. Die Klasse *MobileNode* wird nur geringfügig erweitert, um festzustellen, ob sich ein mobiles Objekt im Dienstgebiet befindet. Die entsprechende Klasse heißt *MobileObject*. Für die Id eines mobilen Objekts wird die in der Klasse *MobileNode* schon vorhandene Variable *nodeid_* verwendet.

Da der Bereich des Dienstgebiets als Simulationsparameter global bekannt ist, wird einfach überprüft ob die aktuelle Position des Objekts innerhalb diese Bereichs liegt. Die zugehörige Funktion ist *MobileObject::isActive()*. Verwendung findet sie bei der Generierung von zufälligen Suchanfragen, damit diese nur nach Objekten suchen, die sich schon im Dienstgebiet befinden. Der Hintergrund dazu ist, dass je nach Generator für die Mobilität, viele Objekte erst nach einiger Zeit in das Dienstgebiet kommen bzw. dieses wieder verlassen. So gibt es Objekte die zu einem bestimmten Zeitabschnitt garantiert nicht gefunden werden können, da sie sich außerhalb des Dienstgebiets befinden und somit keinen Nutzen für die Evaluierung besitzen.

5.6 Infrastruktur-Knoten

Wie schon beim Systementwurf unterscheiden wir bei der Implementierung zwischen Knoten, die nur mit anderen Infrastruktur-Knoten kommunizieren, diese werden im Abschnitt Koordinator behandelt, und Knoten, die zusätzlich mit mobilen Knoten kommunizieren, diese behandeln wir im Abschnitt Zell-Knoten als spezialisierte Koordinatoren.

5.6.1 Koordinator

Diese Knoten besitzen nur eine Anwendungs-Komponente und weder einen Agent noch eine physische Komponente. Zur Realisierung dieser wird die Klasse *InfrastructureApp* verwendet (siehe Auflistung 5.9).

```
class InfrastructureApp : public BasicApp {

        void addQuery(int queryId,int targetId,double duration,bool permanent);
        ...

        double borderLeft;
        double borderRight;
        double borderTop;
        double borderBottom;

        BasicApp* parent;
        vector<BasicApp*> childNodes;

        ObjectCache* objectCache;
        ...
};
```

Listing 5.9: Aufbau der Klasse InfrastructureApp

In der Anwendungs-Komponente wird eine Menge von Kindknoten gespeichert, denen der Knoten übergeordnet ist und mit denen er kommuniziert. Im Fall des hierarchischen Ansatzes wird außerdem der Vaterknoten gespeichert, außer natürlich es handelt sich um die Wurzel. Die Kommunikation mit den Kind- bzw. mit dem Vaterknoten erfolgt direkt über einen Aufruf der Methode BasicApp::process_data(int size, AppData* data). Die Methode *InfrastructureApp::addQuery(int queryId,int targetId,double duration,bool permanent)* dient als Schnittstelle für Suchanfragen und kann nur auf dem Wurzel-Knoten aufgerufen werden.
Ein Koordinator ist für einen festgelegten Bereich des Dienstgebiets zuständig, welcher sich aus den Bereichen seiner Kindknoten ergibt. Dieser wird durch 4 Grenzen repräsentiert, die aufgrund der Aufteilung des Gebiets in allgemein achsenparallele Rechtecke, jeweils Parallelen zur x- bzw. y-Achse entsprechen. Um einen Kind- bzw. Vaterknoten hinzuzufügen wird der Tcl-Befehl *add-child* bzw. *add-parent* benutzt. Die generelle Gebietsaufteilung erfolgt über das Tcl-Skript und wird zunächst für die Zell-Knoten, also der untersten Ebene, durchgeführt. Ihre Grenzen werden dann rekursiv nach oben propagiert. Dazu wird der Befehl *init-service-area* bei der Wurzel aufgerufen. Der Aufruf *is-root* wird benutzt um dem Koordinator mitzuteilen, dass er der Wurzelknoten ist.
Für den hierarchischen Ansatz besitzen die Koordinatoren außerdem noch eine Instanz der Klasse *ObjectCache*, welche den Objekt-Positions-Cache realisiert und somit die letzte bekannte Positions eines Objekts speichert.

5.6.2 Zell-Knoten

Diese Knoten sind fast genauso wie die im letzten Abschnitt beschriebenen, allgemeinen Koordinatoren aufgebaut, was die Anwendungs-Komponente betrifft. Sie besitzen jedoch zusätzlich eien Agenten und eine physische Komponente.

physische Komponente

Hierfür wird wieder eine Instanz der Klasse *MobileNode* verwendet. Im Unterschied zu mobilen Knoten und Objekten bewegen sich diese Knoten nicht, jedoch wird zur Vereinheitlichung die Position in der physischen Komponente gespeichert. Die Koordinaten entsprechen dem Mittelpunkt der Zelle für den der Knoten zuständig ist. Der Mittelpunkt wird bei der Unterteilung des Dienstgebiets im Tcl-Skript mittels *$cellNode set X_* bzw. *$cellNode set Y_* initialisiert.

Agent

```
class CellAgent : public BasicAgent {
public:

        void sendmsg(int nbytes, AppData* data, const char *flags){
                if(locM!=NULL && linkToOwnNode!=NULL){
                        locM->locdb->sendToMobileNodes(
                                                linkToOwnNode,
                                                nbytes,data);
                }
        }

        void unicastToMobileNodeId(int nbytes, AppData* data,
                                   const char *flags, int nodeId){
            if(locM!=NULL && linkToOwnNode!=NULL){
                locM->unicastToMobileNodeId(linkToOwnNode,
                                            nbytes,data,nodeId);
            }
        }

private:

        LocationManager* locM;
};
```

Listing 5.10: Aufbau der Klasse CellAgent

Um den Agent bereitzustellen wird die Klasse *CellAgent* verwendet, die von BasicAgent abgeleitet wird. Diese stellt der Anwendung die Methoden *CellAgent::sendmsg(int nby-*

tes, AppData data, const char *flags)* und *CellAgent::unicastToMobileNodeId(int nbytes, AppData* data, const char *flags, int nodeId)* zur Verfügung, mittels derer die Kommunikation mit den mobilen Knoten, die sich in der dem Knoten zugeordneten Zelle befinden, ermöglich wird. Wie in Auflistung 5.10 wird dabei auf die vom Lokations-Verwalter bereitgestellten Methoden zurückgegriffen.

Anwendung

```
class CellApp : public BasicApp {

        ...
private:

        ...
        double borderLeft;
        double borderRight;
        double borderTop;
        double borderBottom;

        BasicApp* neighborTop;
        BasicApp* neighborBottom;
        BasicApp* neighborLeft;
        BasicApp* neighborRight;
        map<int, BaseStationApp*> neighborIdMap;

        map<int, set<int> > mobileNodeObjectIdList;

        BasicApp* parent;

        LocationManager* locM;
        CellAgent* agent_;
};
```

Listing 5.11: Aufbau der Klasse CellApp

Die Anwendungs-Komponente eines Zell-Knotens wird durch die Klasse *CellApp* realisiert. Die Kommunikation mit dem Vaterknoten erfolgt wie bei einem gewöhnlichen Koordinator direkt über den Aufruf von *BasicApp::process_data(int size, AppData* data)*. Für die Kommunikation mit den mobilen Knoten wird eine Instanz der Klasse *CellAgent* verwendet.

Zusätzlich ist es nötig, dass ein Zell-Knoten alle an ihn angrenzenden Nachbar-Zell-Knoten kennt. Der Grund hierfür ergibt sich aus dem Zellwechsel eines mobilen Knoten. Dabei fragt der mobile Knoten die Objekts-IDs der Zelle ab, in der er sich nun befindet. Der Zell-Knoten schickt jedoch zur Minimierung der übertragenen Daten an den mobilen Knoten nur die IDs der Objekte, die nun, im Vergleich zur vorherigen Zelle,

zusätzlich gesucht werden müssen. Außerdem werden nur die Objekt-IDs geschickt nach dem der mobile Knoten nicht schon sucht. Um diese Information zu erhalten, speichert ein Zell-Knoten die an einen mobilen Knoten gesendeten Objekt-IDs. Beim Zellwechsel fragt dann ein Zell-Knoten diese Liste vom Nachbar-Zell-Knoten ab, von dem der mobile Knoten gewechselt ist.
Außerdem werden zur Realisierung der dynamischen Suchbereichsanpassung beim Gitter-Ansatz die horizontalen und vertikalen Nachbarn-Zell-Knoten gesondert gespeichert, um eine Ausbreitung analog zu [RFH+01] zu ermöglichen.

5.7 Nachrichten

```
class AppData {
private:
        AppDataType type_;        // ADU type
public:
        AppData(AppDataType type) { type_ = type; }
        AppDataType type() const { return type_; }
        virtual int size() const { return sizeof(AppData); }
        ...
};
```

Listing 5.12: Auszug aus der ns2 Klasse AppData

```
class TrackingDataTargetReport : public AppData {
struct targetReportHeader{
        int senderId;
        double senderPosX;
        double senderPosY;
        double timestamp;
};
public:
        TrackingDataTargetReport() : AppData(TRACKING_TARGET_REPORT){}
        vector<int>* getTargets();
        virtual int size() const;
        ...
protected:
        targetReportHeader header;
        vector<int> objectIdList;
};
```

Listing 5.13: beispielhafter Aufbau einer von AppData abgeleiteten Nachricht

Eine Nachricht wird allgemein wie in Auflistung 5.12 modelliert. Von ns2 wird hierfür die Klasse *AppData* bereitgestellt, die wir, um unsere verwendeten Nachrichten zu reali-

sieren, wie in Auflistung 5.13 beispielhaft dargestellt, erweitern. Dabei wird die Funktion *AppData::size()* so angepasst, dass ihr Wert der Größe der Nachricht in Bytes entspricht. Dabei wird die Größe des Headers und des Inhalts betrachtet. Das Typ-Feld wird benutzt, um die unterschiedlichen Nachrichtentypen zu identifizieren. In der Anwendungs-Komponente erfolgt dann eine typspezifische Verarbeitung der Nachricht.
Um die verschiedenen Nachrichtentypen bereitzustellen erweitern wir den Aufzählungstyp AppDataTyp, der von ns2 in *ns-process.h* definiert wird (vergleiche Auflistung 5.14).

```
// Application-level data unit types
enum AppDataType {
        // Illegal type
        ADU_ILLEGAL,
        ...
        //mobile target tracking ADU
        TRACKING_TARGET_REPORT, //Objektfund Nachricht
        TRACKING_TASK_LIST_DELTA,   //Suchauftrag für mobilen Knoten
        TRACKING_GRID_BROADCAST_TASK_LIST_REMOVE, //Gitter Broadcast
        TRACKING_GRID_TASK_LIST_TIMEOUT, //Suchbereichsanpassung
        ...
        // Last ADU
        ADU_LAST
};
```

Listing 5.14: Erweiterung von AppData Type um weitere Nachrichtentypen

5.8 Simulationsparameter

Die Paramter für die Simulation befinden sich in der Klasse *SimulationParameters*. Jeder dort definierte Parameter ist eine statische Variable und somit für alle Klassen mittels *SimulationParameters::parameterName* zugänglich. Jeder Parameter wird über einen Aufruf von *bind("parameterNameInTcl",&variablenName)* im Tcl-Skript zugänglich gemacht. Im Skript selbst sorgt eine Instanz der gleichnamigen Tcl-Klasse *SimulationParameters* dafür, dass die Parameter gesetzt werden können. Diese wird mittels *set param [new SimulationParameters]* angelegt. Das setzen eines Parameters erfolgt durch *$param set <ParameterName> <Wert>*.

serviceAreaX bestimmt den maximalen x-Wert des Dienstgebiets, dabei ist der minimale Wert stets 0

serviceAreaY bestimmt analog zum vorherigen Paramter den maximalen y-Wert des Dienstgebiets, da wir nur quadratische Bereiche

betrachten ist dieser Parameter gleich seviceAreaX

levelOfBaseStations gibt die Tiefe des Baums beim hierarchischen Ansatz an

gridSize bestimmt die Anzahl der horizontalen bzw. vertikalen Zellen beim Gitter-Ansatz

baseStationSquareSize gibt sowohl die Länge als auch die Breite einer Zelle an, Parameter wird für den hierarchischen Ansatz und Gitter-Ansatze auf Grundlage des Parameters *levelOfBaseStations* bzw. *gridSize* vom Tcl-Skript berechnet

objectsMaximumSpeed gibt die maximale Geschwindigkeit eines Objekts an, mittels derer die Objektposition abgeschätzt wird

mobileNodeGPSInterval bestimmt den Abstand zwischen zwei Positionsaktualisierungen des Agenten

mobileNodeSensingRange bestimmt Sensorreichweite eines mobilen Knotens in Metern

mobileNodeReportTargetsInterval bestimmt den Abstand zwischen der Suche nach mobilen Objekten und der anschließenden Meldung des Objektfunds in der Umgebung

mobileNodeReportTargetsSeedRange bestimmt das Intervall innerhalb dessen die Timer zum Melden eines Objektsfunds zu Anfang der Simulation desynchronisiert werden

queryGeneratorMeanDurationOfQuery gibt die durchschnittliche Dauer einer Suchanfrage in Sekunden an

queryGeneratorMeanTimeBetweenQueries gibt den durchschnittlichen Abstand zwischen zwei Suchanfragen an

queryGeneratorMaxTimeBetweenQueries bestimmt die maximale Zeit in Sekunden zwischen zwei Suchanfragen

strategyInfrastructureNodes bestimmt welchen der vorgestellten Ansätz die Infrastruktur-Knoten verfolgen

5.9 Verknüpfung der Infrastruktur-Knoten

Die Verknüpfung der Infrastruktur-Knoten erfolgt durch ein Tcl-Skript. Dabei werden nacheinander die einzelnen Infrastrukturkoten mit ihren Komponenten angelegt und anschließend nach dem Schema wie in Abbildung 5.4 und Abbildung 5.3 dargestellt verbunden.
Dabei wird das durch die Parameter *serviceAreaX* und *serviceAreaY* vorgegebene Dienstgebiet unterteilt und jedem Bereich ein Zell-Knoten zugewiesen.

5.9.1 hierarchischer Ansatz

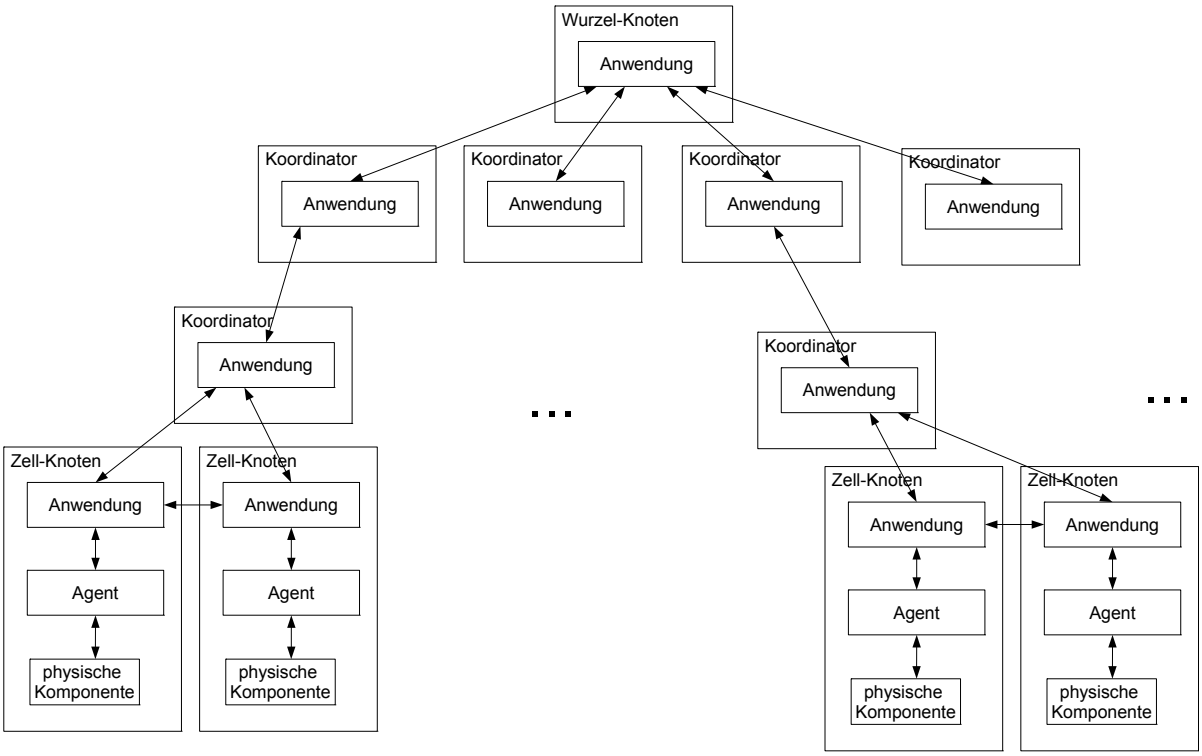

Abbildung 5.3: Schematischer Aufbau der Verknüpfung der Komponenten beim Hierarchischen-Ansatz

Wie in Abbildung 5.3 dargestellt, bilden die Infrastruktur-Knoten beim hierarchischen Ansatz durch die Verknüpfung der einzelnen Anwendungs-Komponenten eine Baumstruktur. Der Parameter *levelOfBaseStations* bestimmt dabei die Tiefe des Baums.

Zur Unterteilung des Dienstgebiets verwenden wir eine Tcl-Funktion (siehe Auflistung 5.15), welche eine rekursive Aufteilung in jeweils 4 gleich große Rechtecke bewerkstelligt. Dabei übergeben wir in jedem Schritt das Minimum und Maximum der x- und y-Werte, wodurch der (Unter-)Bereich festgelegt wird. Die Tiefe dient als Abbruchkriterium der Rekursion und bewirkt, dass dem nächsten Zell-Knoten, dem bisher noch kein Gebiet zugewiesen wurde, der Mittelpunkt eines Gebiets zugewiesen wird. Dieser Mittelpunkt ergibt sich aus den Werten der Funktionsparameter beim Rekursionsabbruch. Den so entstehenden Vierergruppen von Zell-Knoten deren Gebiete aneinander grenzen, wird dann ein gemeinsamer Koordinator zugewiesen. Die Größe einer Zelle gibt der Parameter *baseStationSquareSize* an, welcher sich bei diesem Ansatz zu $baseStationSquareSize = \frac{serviceAreaX}{2^{levelOfBaseStations}}$ berechnet, da wir nur quadratische Zellen betrachten.

```
set BSCounter 0
proc placeBS {xMin xMax yMin yMax level} {
global BSCounter BS
 if {$level==0} {
    $BS($BSCounter) set Z_ 0.0
    $BS($BSCounter) set Y_ [expr ($xMin+$xMax)/2.0]
    $BS($BSCounter) set X_ [expr ($yMin+$yMax)/2.0]
    set BSCounter [expr $BSCounter+1]
 } else {
    placeBS      $xMin [expr ($xMin+$xMax)/2.0] $yMin
                 [expr ($yMin+$yMax)/2.0] [expr $level - 1]

    placeBS      [expr ($xMin+$xMax)/2.0] $xMax $yMin
                 [expr ($yMin+$yMax)/2.0] [expr $level - 1]

    placeBS      [expr ($xMin+$xMax)/2.0] $xMax
                 [expr ($yMin+$yMax)/2.0] $yMax [expr $level - 1]

    placeBS      $xMin [expr ($xMin+$xMax)/2.0]
                 [expr ($yMin+$yMax)/2.0] $yMax [expr $level - 1]
 }
}
```

Listing 5.15: Tcl-Funktion zur Gebietsunterteilung beim hierarchischen Ansatz

Um den Zellwechsel eines mobilen Knoten zu realisieren sind die Zell-Knoten außerdem mit ihren direkt angrenzenden Nachbarn-Zell-Knoten verbunden. Diese Verbindungen sind für die Funktionalität des gesamten Systems wichtig, da es nicht sehr effizient wäre einem wechselnden Knoten die gesamte Suchauftragsliste zu schicken. Für die

dynamische Suchbereichsanpassung kommunizieren die Koordinatoren, also insbesondere auch die Zell-Knoten, nur mit ihrem Vater- bzw. ihren Kindknoten.

5.9.2 Gitter-Ansatz

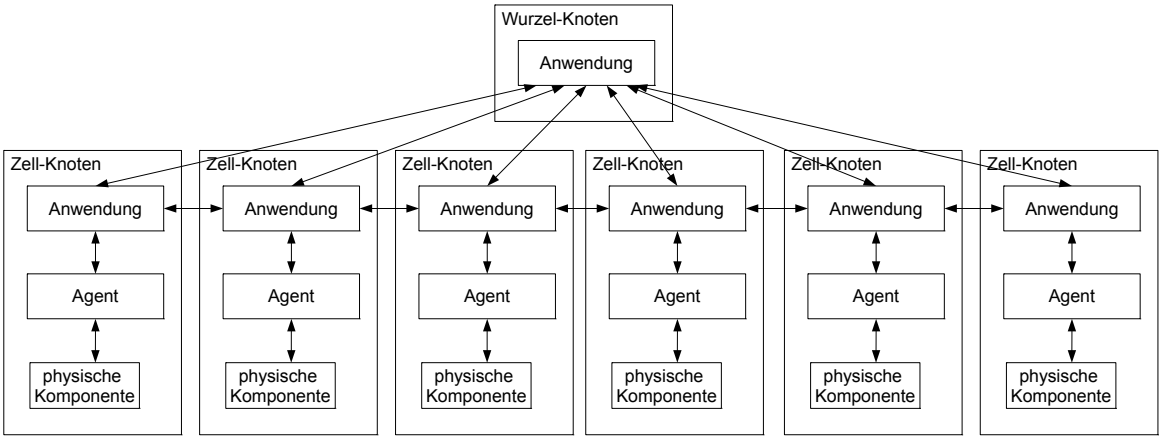

Abbildung 5.4: Schematischer Aufbau der Verknüpfung der Komponenten beim Gitter-Ansatz

```
set BSCounter 0
proc placeBSGrid {sizeX sizeY gridSize} {
global BSCounter BS
 for { set i 0 } { $i < $gridSize } {incr i} {
   for { set j 0 } { $j < $gridSize } {incr j} {
      $BS([expr ($i*$gridSize)+$j]) set X_ \
      [expr ($sizeX/(2.0*double($gridSize))) \
      +(double($j) * ($sizeX/double($gridSize)))]

      $BS([expr ($i*$gridSize)+$j]) set Y_ \
      [expr ($sizeY/(2.0*double($gridSize)))\
      +(double($i) * ($sizeY/double($gridSize)))]

      $BS([expr ($i*$gridSize)+$j]) set Z_ 0.0
   }
 }
}
```

Listing 5.16: Tcl-Funktion zur Gebietsunterteilung beim Gitter-Ansatz

Beim Gitter-Ansatz bilden die Knoten eine flache Hierarchie. Zusätzlich sind jeweils die horizontalen und vertikalen Nachbarn auf Zell-Knoten-Ebene miteinander verbunden und bilden so das Gitter. Hierüber läuft der Algorithmus zur dynamischen Suchbereichs-

anpassung mittels sich schrittweise verbreitenden Timeout-Nachrichten.
Die Zuweisung einer Zelle zu einem Zell-Knoten geschieht wie beim hierarchischen Ansatz durch das Setzen der x- und y-Koordinate des Zell-Knotens auf den Mittelpunkt der Zelle. Die Größe der Zelle bestimmt hier der Parameter *gridSize*. Damit ergibt sich die Zellgröße zu $baseStationSquareSize = \frac{serviceAreaX}{gridSize}$. Die Bestimmung und das setzen des Mittelpunkts geschieht wieder über eine Tcl-Funktion, welche in Auflistung 5.16 dargestellt ist.
Genauso wie beim hierarchischen Ansatz kann ein Zell-Knoten mit all seinen direkt angrenzenden Nachbarn kommunizieren. Dies tut er jedoch genauso wie beim hierarchischen Ansatz nur im Rahem des Zellwechsels eines mobilen Knotens.

6 Evaluierung

Die in Kapitel 3.1 und 5 vorgestellten Ansätze und Implementierungen unseres Systems wurden zusammen in einer Simulationsumgebung eingesetzt. Wir stellen zunächst den Simulationsaufbau und die zu untersuchenden Strategien, zusammen mit den verwendeten Metriken dar. Danach erläutern wir die einzelnen Szenarien und diskutieren hierfür jeweils die Ergebnisse. Am Ende Vergleichen wir dann die einzelnen Ergebnisse untereinander für ein Gesamtfazit.

6.1 Simulationsaufbau

Im Folgenden beschreiben wir den Simulationsaufbau. Hierzu werden zunächst die evaluierten Szenarien bestehend aus verschiedenen Straßentopologien und Mobilitätsmodellen beschrieben, gefolgt von einer Beschreibung der evaluierten Ansätze.

6.1.1 Simulationsszenario

Ein Simulationsszenario besteht aus einem bestimmten zugrundegelegten Straßenmodell und den daraus erzeugten Mobilitätsdaten für sowohl die mobilen Knoten als auch die Mobilen Objekte. Die Anzahl der mobilen Knoten lag bei allen Simulationsszenarien bei 1173. Diese Suchen periodisch im Abstand von einer Sekunde nach mobilen Objekten, sind jedoch untereinander desynchronisert. Das heißt, dass jeder Knoten zufällig einen Startzeitpunkt innerhalb der ersten fünf Sekunden der Simulation wählt und von diesem Zeitpunkt an jeweils im Abstand von einer Sekunde nach mobilen Objekten sucht und meldet. Die Zahl der mobilen Objekte beträgt 3033. Die Werte für die Anzahl der mobilen Knoten und Objekte ergaben sich aus den generierten Mobilitätsdaten und konnten trotz Parameter nicht exakt eingestellt werden. Die Szenarien werden außerdem durch das Muster der Suchanfragen charakterisiert. Da wir für zwei feste Mobilitätsdatensätze verschiedene Anfragemuster betrachten, teilen wir die Szenarien nach den Anfragemustern ein.
Die Größe des gewählten Kartenausschnitts beträgt bei beiden Mobilitätsgenerator jeweils 2x2 Kilometer.

Mobilität

Für die Generierung der Mobilitätsdaten verwenden wir kein einfaches Random-Waypoint Modell, sondern ein graphbasiertes Modell, da sich damit die Bewegungen von Fußgängern realistischer modellieren lassen. Als Grundlagen werden hierfür Straßenkarten verwendet, aus denen dann ein Graphmodell erstellt wird. Auf diesem wiederum

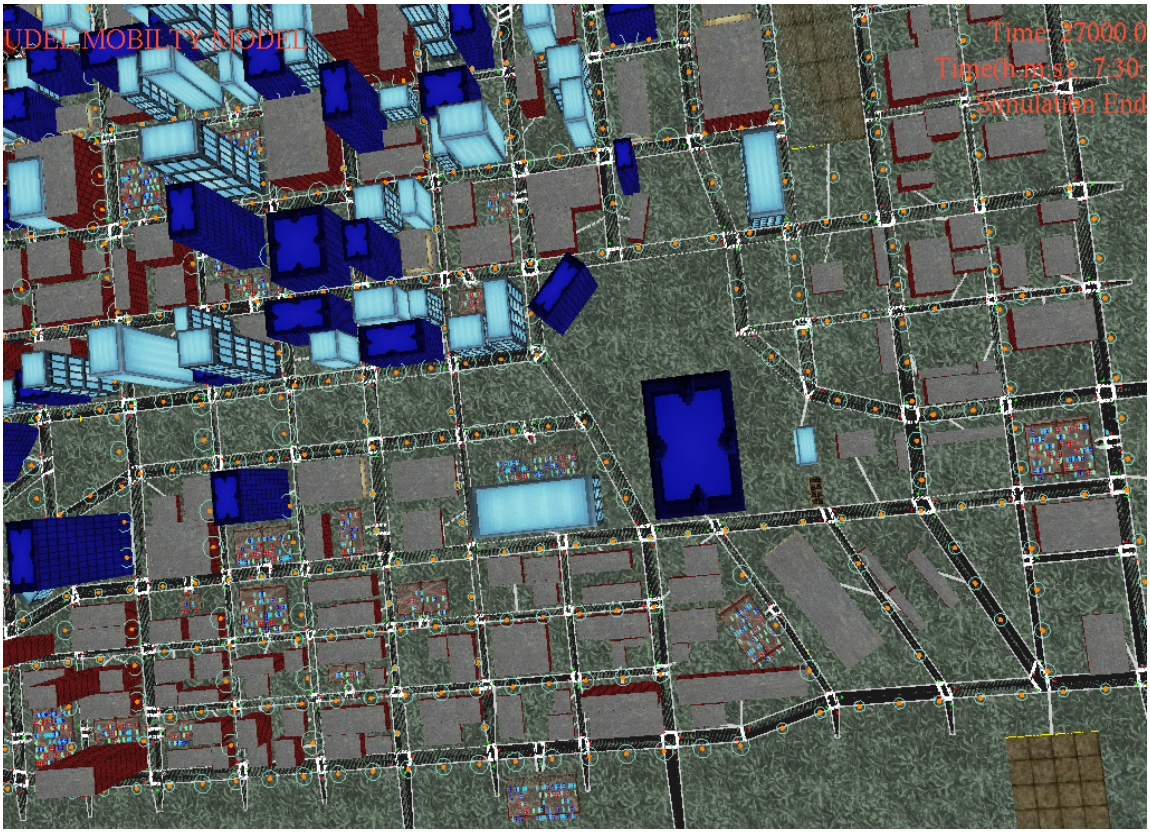

Abbildung 6.1: Verwendete Kartendaten aus dem UDel Modell

wird dann die Mobilität der Knoten simuliert. Die daraus entstehenden Mobilitätsdaten spiegeln so unser Umgebungsszenarion, bei dem es sich um städtische Gebiete handelt, sehr gut wieder. Für die Erzeugung der Mobilitätsdaten verwenden wir zum einen UDel Mobility Simulation[BSI+], zusammen mit einem Ausschnitt aus Chicago als Straßendaten. Dieser ist in Abbildung 6.1 dargestellt. Wir erzeugen mit dem Simulator jeweils für mobile Knoten und mobile Objekte separate Bewegungsdaten, die sich in ns2 einbinden lassen. Dabei modellieren wir sowohl mobile Knoten als auch mobile Objekte als Fußgänger, wobei jeweils mit verschiedenen Parametern. Die maximale Objektegeschwindigkeit wurde durch Messen ermittelt, da man diese Parameter zwar angeben kann, wie sich jedoch gezeigt hat ist dieser wirkungslos, denn es traten innerhalb des Gebiets Knoten mit höheren Geschwindigkeiten auf. Die Geschwindigkeit mobiler Objekte liegt zwischen 1 $\frac{m}{s}$ und 7 $\frac{m}{s}$. Die erzeugten Bewegungsdaten modellieren hier insbesondere das Gruppenverhalten von Personen. Für den Fall mobiler Objekte wurde das Gruppenverhalten dabei schwächer gewählt, so dass ein möglichst realistisches Verhalten abgebildet wird, da wir annehmen, dass in der Realität sich Objekte nicht so stark in Gruppen bewegen wie dies Personen tun.

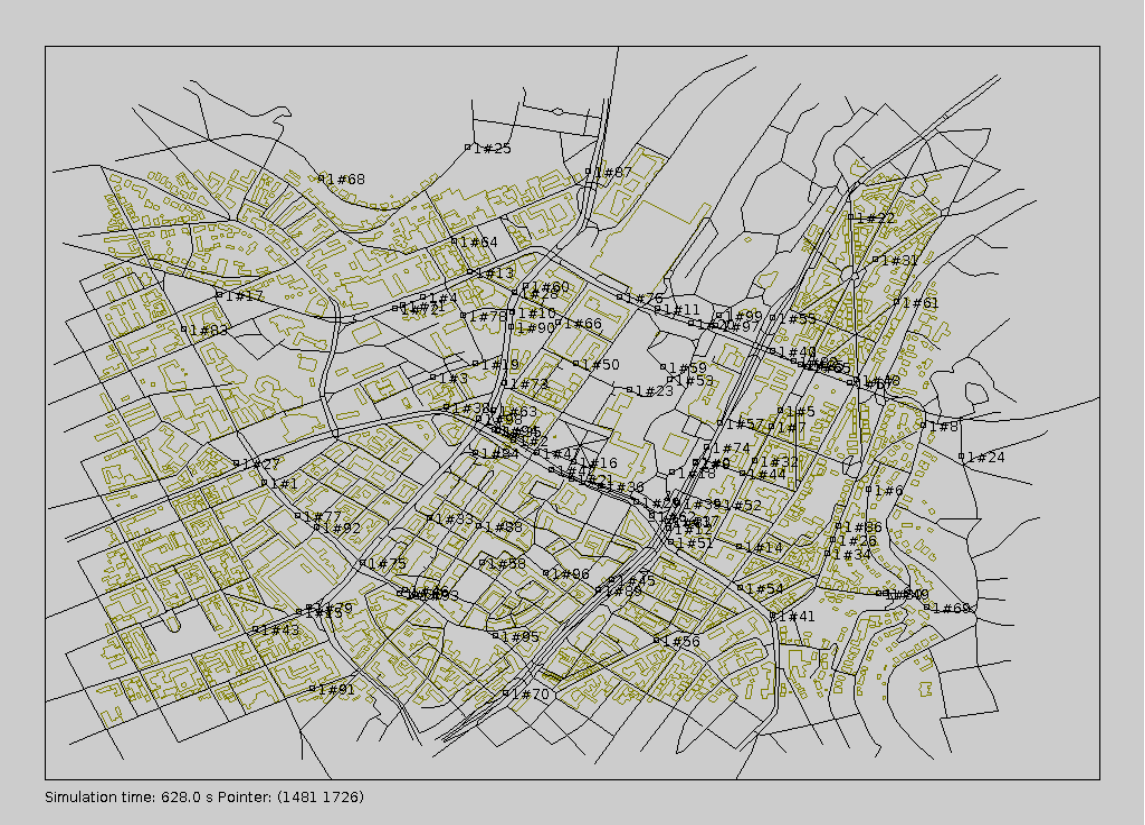

Abbildung 6.2: Auszug aus der Stuttgarter Innenstadt, dient als Eingabedaten für CanuMobiSim

Des Weiteren verwenden wir CanuMobiSim[Ste] um eine zufällige Knotenbewegung, die jedoch an Straßen gebunden ist, zu simulieren. Beim UDel Simulator bewegen sich die Knoten zwar auch prinzipiell zufällig, jedoch werden hierbei einzelne Aktivitäten, wie der Weg zur Arbeit, und die Umgebung, wie beispielsweise Ampeln, mit modelliert. Genauso wie beim Udel Mobility Simulator werden hier Mobilitätsdaten erzeugt, die sich direkt in ns2 einbinden lassen. Die verwendeten Kartendaten entsprechen einem Teil von Stuttgart (siehe Abbildung 6.2). Dieser zweite Simulator wurde verwendet, etwaige Auswirkungen der Straßendaten und des Gruppenverhaltens zu evaluieren. Ein wesentlich Grund war auch, dass sich hierbei vor allem die maximale Objektgeschwindigkeit der Ausgabedaten, die maßgeblich für die dynamische Suchbereichsanpassung ist, besser anpassen ließ. Die Objektgeschwindigkeit der Mobiltätsdaten von CanuMobiSim liegt zwischen 0.5 $\frac{m}{s}$ und 3 $\frac{m}{s}$

6.1.2 Anfragemuster

Wir verwenden insgesamt vier verschiedene Anfragemuster. Dabei liegt allen eine Poissonverteilung zugrunde, wobei definiert wird wie groß im Mittel der Abstand zwischen zwei Anfragen ist und welche Zeitspanne maximal zwischen zwei Anfragen liegt. Die mittlere Dauer einer Suchanfrage wird nach einer Exponentialverteilung bestimmt. Die so zeitlich festgelegten Suchanfragemuster wurden nach ihrer Genrierung in einer Datei protokolliert. So können wir garantieren, dass das jeweilige Anfragemuster bei den verschiedenen Simulationsläufen exakt gleich ist.
Wir betrachten als Anfragetyp nur permanente Suchanfragen, da dadurch die Optimierungen besser sichtbar werden.

Die eine Gruppe von Mustern repräsentieren eine gleichbleibende Anfragerate. Das erste Muster aus dieser Gruppe bezeichnen wir als *hohe Anfragerate*. Hierfür wählen wir eine Rate von 50 Anfragen pro Sekunde im Mittel. Der maximale Abstand zweier Anfragen beträgt 120 Sekunden. Im Mittel dauert eine Suchanfrage 60 Sekunden.
Das zweite Muster *niedrige Anfragerate* stellt eine im Mittel gleichbleibend niedrige Anfragerate dar. Hier liegt die Rate im Mittel bei 2 Anfragen pro Sekunde. Die maximale Zeitspanne zwischen zwei Anfragen beträgt 120 Sekunden. Die mittlere Dauer einer Suchanfrage liegt ebenfalls bei 60 Sekunden.
Das zweite Muster ergänzen wir außerdem durch einen kurzen "Bursts", der eine mittlere Rate von 200 Anfragen pro Sekunde hat 100 Sekunden lang andauert. Das so entstehende Muster bezeichnen wir als *niedrige Anfragerate mit Burst*.
Bei allen vorgestellten Mustern werden bei der Generierung die gesuchten Objekte zufällig ausgewählt. Jedoch mit der Einschränkung, dass nur solche Objekte in Betracht gezogen werden, die sich zum Zeitpunkt der Generierung im Dienstgebiet aufhalten. Diese Einschränkung betrifft nur den Udel Mobility Simulator, da sich dort die Knoten das Gebiet betreten und wieder verlassen.

6.1.3 Ansätze

Es wurden sowohl der Basis-Ansatz als auch der Gitter-Ansatz implementiert und verglichen. Aufgrund der Zeitbeschränkung dieser Arbeit konnte der hierarchische Ansatz nicht vollständig umgesetzt und in die Evaluierung einbezogen werden. Intuitiv stellt der Gitter-Ansatz jedoch die bessere Variante dar, da dieser den Suchbereich feiner anpassen kann.

Basis-Ansatz

Dieser Ansatz dient als Vergleich, da er die maximal möglichen Objektfund-Nachrichten garantiert, indem er jeden mobilen Knoten beauftrag nach einem Objekt zu suchen. Angestrebt ist die von diesem Ansatz gelieferte Anzahl an Objektfund-Nachrichten zu erhalten (Effektivität) und dabei das übertragene Datenvolumen zwischen Zell-Knoten und mobilen Knoten zu minimieren (Effizienz).

Gitter

Der wesentliche Parameter beim Gitter-Ansatz ist die Anzahl der Knoten in vertikaler und horizontaler Richtung. Da wir nur Fälle betrachten in denen horizontal und vertikal jeweils gleichviele Zellen sind, ergibt sich eine Benennung nach dem Muster "Gitter_AxA".

6.2 Messgrößen

Die nun folgenden Metriken werden verwendet, um die Leistungen der verschiedenen Ansätze vergleichen zu können.

6.2.1 Effizienz

Für die Effizienz messen wir jeweils die gesendeten Nachrichten zwischen mobilen Knoten und den Zell-Knoten. Dabei Messen wir sowohl die gesendete Bytes pro Sekunde als auch die gesamte übertragene Datenmenge. Dies geschieht für beide Richtungen, also von mobilen Knoten zu Zell-Knoten und umgekehrt, getrennt. Die Messung dieser beiden größen motiviert sich daher, dass wir hier für die Kommunikation auf Mobilfunk setzen und daher die benötigte Bandbreite so gering wie möglich halten wollen, um zum einen das System überhaupt realisierbar und skalierbar zu gestalten und zum anderen die Teilnahme am System für die Besitzer der mobilen Geräte attraktiv zu halten. Für die Kommunikation zwischen den Infrastruktur-Knoten sehen wir im Vergleich zu Mobilfunk eine nahezu unerschöpfliche Bandbreite. Außerdem bewirkt eine Reduktion der gesamt übetragenen Datenmenge auch eine Verringerung der Serverlast. Obwohl wir eine vergleichsweise extrem große Bandbreite im kabelgebundenen Netz annehmen, ist die Anzahl an zu verarbeitenden Nachrichten auf Serverseite ein wichtiges Kriterium. Dies lässt sich aus der übertragenen Datenmenge ableiten.

6.2.2 Effektivität

Für die Messung der Effektivität dient im wesentlichen die Anzahl der gemeldeten Objektpositionen durch die mobilen Knoten. Genauer betrachten wir die pro Sekunde gemeldeten Objektpositionen, also nicht die absolute Anzahl sondern das "delta" Insbesondere zeigt ein Vergleich dieser Größe mit dem Basisansatz, ob die Optimierungen der Effizient negative Effekte auf die Effektivität haben. Wie schon erwähnt ist das Ziel hierbei die gleiche Effizienz wie beim Basis-Ansatz zu erhalten und dabei die Anzahl der Nachrichten zu minimieren. Weiterhin messen wir die durchschnittliche Gebietsgröße in der nach einem mobilen Objekt gesucht wird um Aussagen über die dynamische Suchbereichsanpassung machen zu können. Dabei berechnen wir in wievielen Zellen nach einem Objekt durchschnittlich gesucht wird und multiplizieren dies mit der Zellgröße. Dieser Bereich entspricht somit dem effektiven Suchbereich, da der vom System geschätzte Bereich auf die Zellen abgebildet wird.

6.3 Ergebnisse

In diesem Kapitel zeigen wir nun die Ergebnisse der Simulation und werten diese aus. Wir vergleichen jeweils die Szenarien für beide Kartendaten, d.h. Chicago und Stuttgart. Bei jeder der aufgeführten Simulationsergebnisse lag die Anzahl der mobilen Knoten bei 1173. Die Anzahl der mobilen Objekte nach denen gesucht werden kann lag bei 3033. Die Sensorreichweite der mobilen Knoten für die gezeigten Ergebnisse lag bei 10 Metern.
Aufgrund einer zu hohen Dateigröße für die Bewegung der mobilen Objekte, wird bei der Straßenkarte für Stuttgart nur ein Zeitabschnitt von 5000 Sekunden betrachtet. Bei der Chicago Karte betrachten wir hingegen 11800 Sekunden an realer Zeit, die simuliert wird.

6.3.1 hohe Anfragerate

Wir betrachten hier den Fall einer hohen konstanten Last von 50 Anfragen pro Sekunde, die im Mittel 60 Sekunden lang andauern.
Zunächst erkennt man, dass, was die Last angeht, die Anzahl der gesendeten Daten

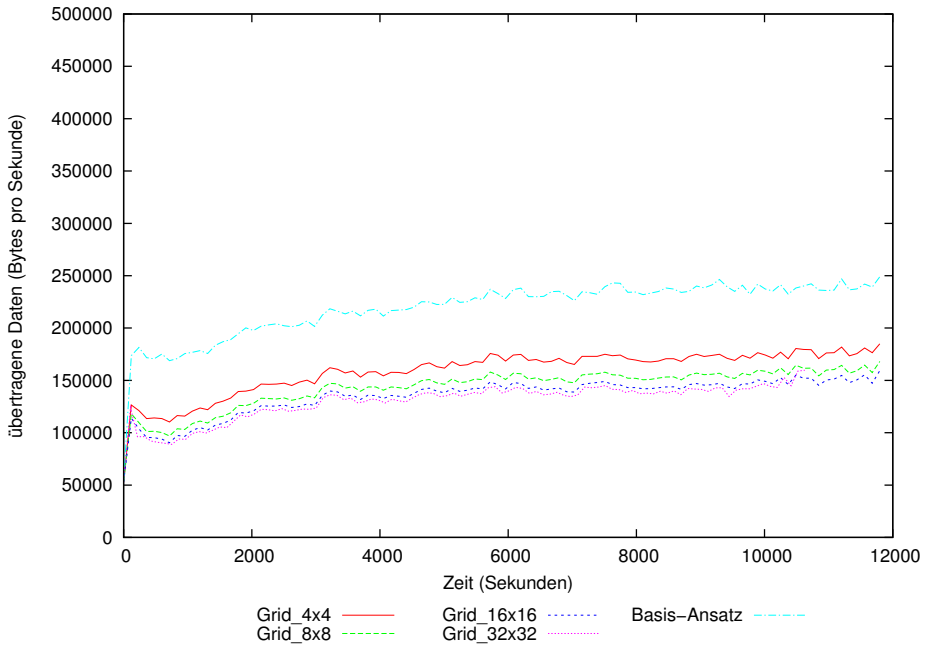

Abbildung 6.3: Messung der Effizienz, Nachrichtenlast von Zell-Knoten zu mobilen Knoten, hohe Anfragerate, Chicago Kartendaten

von den Zell-Knoten zu den mobilen Knoten beim Gitter-Ansatz niedriger als beim Basis-Ansatz ist. Die Einsparung beträgt pro Sekunde ungefähr 75.000 Bytes. (siehe Abbildung 6.3). Betrachtet man nun die verschiedenen Gittergrößen untereinander, so erkennt man einen Trend dahingehen, dass um so höher die Anzahl an Zellen ist, desto niedriger wird die Anzahl der übertragenen Daten pro Sekunde. Dies lässt sich damit

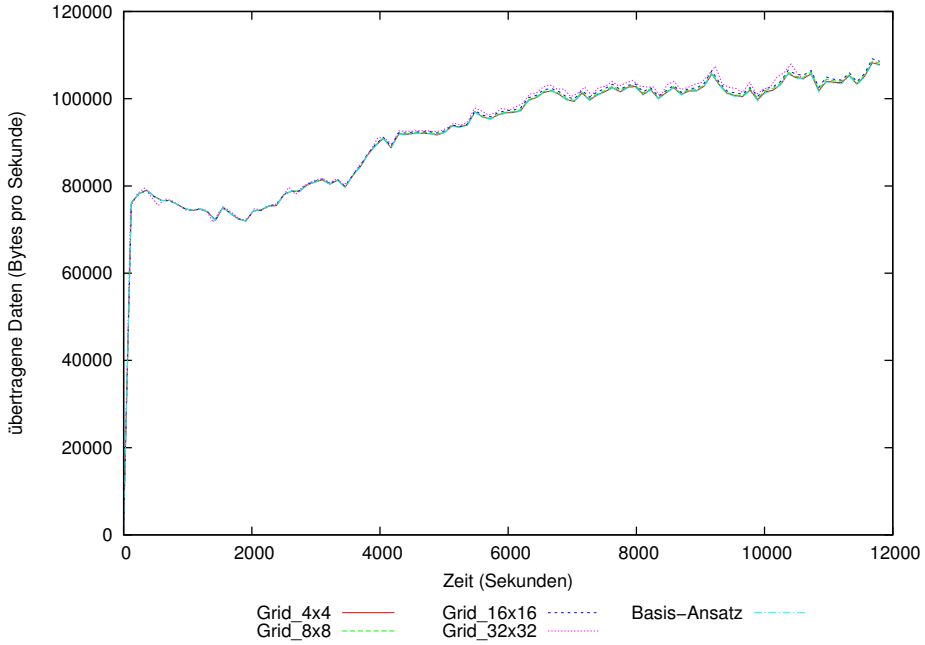

Abbildung 6.4: Messung der Effizienz, Nachrichtenlast von mobilen Knoten zu Zell-Knoten, hohe Anfragerate, Chicago Kartendaten

begründen, dass der Suchbereich um so besser durch die einzelnen Zellen abgebildet werden kann, je kleiner die einzelnen Zellen sind. Jedoch stagniert dieser Wert um so mehr, je höher die Anzahl an Zellen ist. Eine Begründung dafür ist, dass ab einer bestimmten Anzahl an Zellen, der vom System abgeschätzte Suchbereich in den Bereich der Zellgrößen kommt und so kleinere Zellen immer weniger Effekt haben.

Betrachtet man die Nachrichtenlast in Richtung Zell-Knoten, so erkennt man, dass diese sowohl im Vergleich zwischen Basis-Ansatz und Gitter-Ansatz, als auch unter den verschiedenen Gittergrößen nahezu identisch ist (siehe Abbildung 6.4). Um sicherzustellen, dass dies kein Overhead ist, der durch die Zell-Wechsel erzeugt wird, sehen wir uns, um die Effektivität zu beurteilen, die Anzahl der empfangenen Objektpositionen an.
Vergleicht man die gesamt übertragenen Datenmengen, so ergibt sich bei einer Gittergröße von 16x16 eine Einsparung bei der Richtung Zell-Knoten zu mobilen Knoten von rund 39%.

Wie in Abbildung 6.5 zu erkennen, ist die Anzahl der erhaltenen Objekpositionen nahezu identisch. Schaut man genauer in die Daten, so liegt die Anzahl der gemeldeten Objektpositionen im 10 Millionen Bereich, die Anzahl der ausbleibenden Objektfunde bewegt sich jedoch im zehntausender Bereich. Prozentual ausgedrückt liegen wir beim Grid-Ansatz im Vergleich zum Basis-Ansatz bei 99% der gemeldeten Objektpositionen des Basis-Ansatzes. Dies spricht dafür, dass der Overhead durch den Zell-Wechsel, den

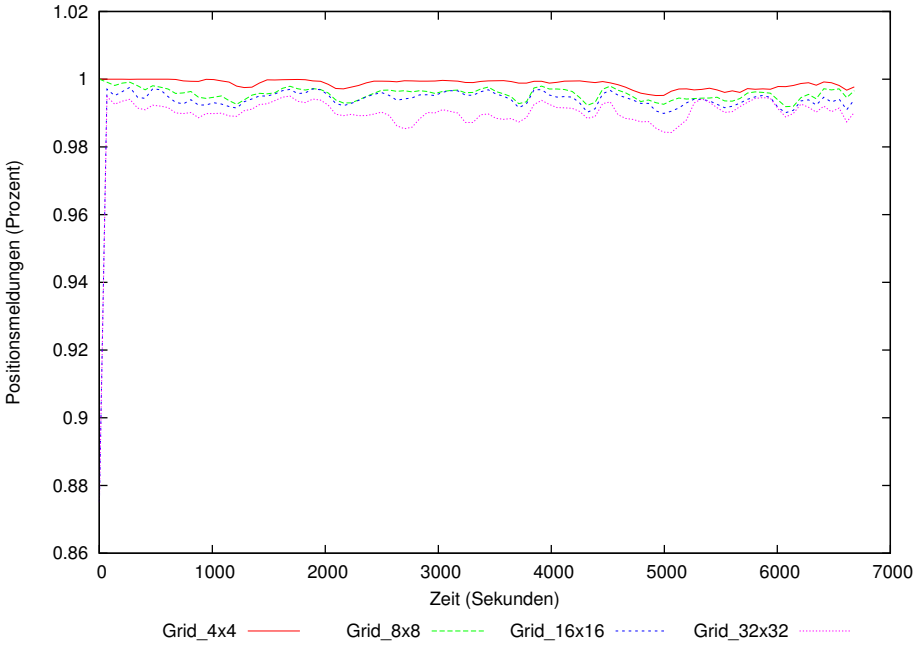

Abbildung 6.5: Messung der Effektivität, Anteil der gemeldeten Objektpositionen pro Sekunde im Vergleich zum Basis-Ansatz in Prozent, hohe Anfragerate, Chicago Kartendaten

wir beim Basis-Ansatz nicht haben, entsprechend niedrig ist.

Betrachtet man nun die durchschnittliche effektive Gebietsgröße in der nach einem Objekt gesucht wird (siehe Abbildung 6.6) , so sieht man, dass diese beim Gitter-Ansatz weniger als die Hälfte des Dienstgebiets beträgt. Beim Basis-Ansatz entspricht sie konstant dem gesamten Dienstgebiet, da dort jedes Objekt, für das eine Suchanfrage existiert, überall gesucht wird. Unter den verschiedenen Gittergrößen zeigt sich ebenfalls der Trend, dass um so höher die Anzahl der Zellen ist, desto kleiner wird das Suchgebiet. Dies erklärt sich dadurch, dass mit der Anzahl der Zellen die Granularität steigt und sich so der effektive Suchbereich feiner anpassen kann. Der Anstieg der Gebietsgröße zwischen 2000 und 4000 Sekunden ist mit einer Verringerung der gemeldeten Objektpositionen zu erklären, da in diesem Zeitraum der Graph für die von den mobilen Knoten gesendeten Daten an die Zell-Knoten abfällt. Der Anstieg der Objektfunde zwischen 6000 und 8000 Sekunden spiegelt sich analog in einem kleineren effektiven Suchbereich wieder.

Betrachtet man das gleiche Anfragemuster für die Mobilitätsdaten für den Ausschnitt aus Stuttgart, so sind die Beobachtungen im wesentlichen gleich. Das heißt, dass die Anzahl der übertragenen Nachrichten von Zell-Knoten zu mobilen Knoten mit der Anzahl der Zellen sinkt. Auch die übertragenen Daten von mobilen Knoten zu Zell-Knoten zeigen ein vergleichbares Bild zur Chicago Karte, sprich insgesamt ist die

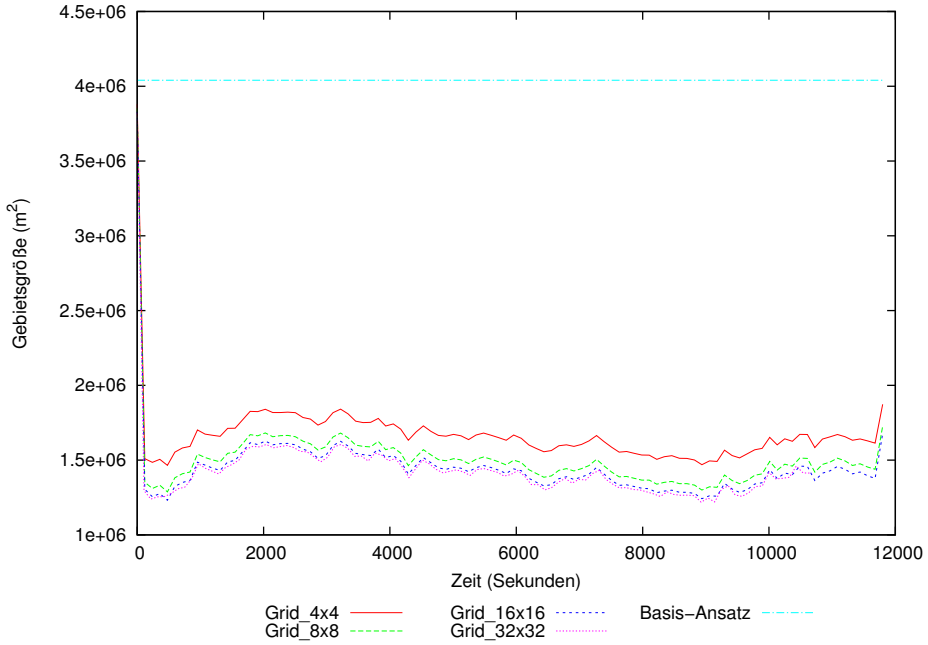

Abbildung 6.6: Messung der Effektivität, mittlere Größe des effektiven Suchbereichs in dem ein Objekt gesucht wird, minimale Größe ist dabei eine Zelle, hohe Anfragerate, Chicago Kartendaten

Effizienz vergleichbar. Lediglich der Suchbereich ist hier deutlich kleiner und beträgt bei einem 4x4 Gitter etwas 500 000 m^2. Dies lässt sich damit begründen, dass bei den Mobilitätsdaten von CanuMobiSim die maximale Objektgeschwindigkeit nur $3\frac{m}{s}$ beträgt, im Gegensatz zu $7\frac{m}{s}$ bei den Chicago Kartendaten. Da dieser Wert entscheidend für die dynamische Suchbereichsanpassung ist, und er bei CanuMobiSim etwas weniger als die Hälfte beträgt, macht es Sinn, dass der Suchbereich ca. 40% kleiner ist als bei den Chicago Kartendaten.
Was die Effektivität anbelangt, so ist die Anzahl der Objektfunde niedriger als bei der Chicago Karte. Sie liegt im schlechtesten Fall bei 90%. Eine mögliche Erklärung ist der verwendete Objekt-Positions-Cache. Dieser verwendet selbst einen Timer um zu bestimmen, wann auch in den Nachbarzellen nach einem Objekt gesucht werden muss. Das heißt der Timeout Wert für einen Eintrag wird alle 0,01 Sekunden erniedrigt. Betrachtet man die 1173 mobilen Knoten, die desynchronisiert jeweils periodisch alle Sekunde nach mobilen Objekten suchen, so beträgt der Abstand zweier verschiedene Objekfund-Nachrichten im Mittel 0,00085. Das Problem wurde zu spät erkannt, da der Großteil der Zeit zu sehr mit den Mobilitätsdaten von UDel mit einer hohen Afragerate gearbeitet wurde und dort nicht zu jedem Zeitpunkt alle mobilen Knoten aktiv sind, woraus ein höherer zeitlicher Abstand zwischen zwei Objektfundnachrichten resultiert und der eingestellt Parameter in Ordnung erschien. Insgesamt ergibt sich daraus bei einer höheren Anzahl an Zellen eine gewissen Verzögerung was hier negative Auswirkungen,

mit dem Ansatz an sich aber nichts zu tun hat.

6.3.2 niedrige Anfragerate

Wir betrachten hier wieder die beiden verschiedenen Mobilitätsdaten, jedoch bei einer niedrigen Anfragerate von im Mittel 2 Suchanfragen pro Sekunde.

Bei der Menge der übertragenen Bytes von Zell-Knoten (siehe Abbildung 6.7) zu

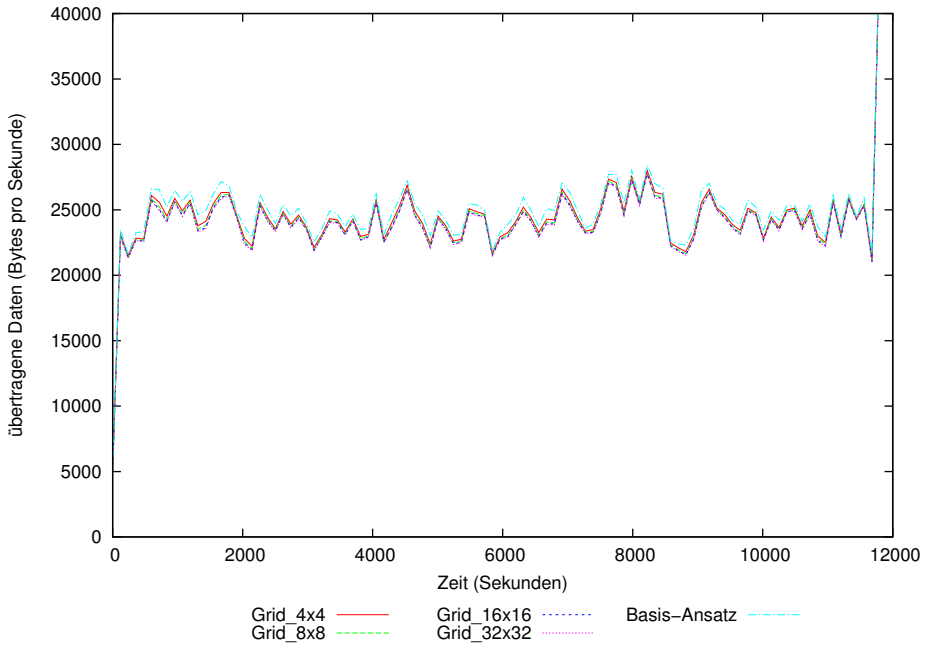

Abbildung 6.7: Messung der Effizienz, Nachrichtenlast von Zell-Knoten zu mobilen Knoten, niedrige Anfragerate, Chicago Kartendaten

mobilen Knoten erkennt man, dass hier die Optimierungen nicht mehr gut greifen und die übertragenen Daten zwischen Zell-Knoten und mobilen Knoten nahezu gleich mit dem Basis-Ansatz sind. Dieser ist jedoch immer noch minimal schlechter. Die Ursache dafür liegt in der niedrigen Anfragerate, denn dadurch werden schon allgemein sehr wenig Nachrichten von Zell-Knoten zu mobilen Knoten übertragen. Daher kristallisieren sich die Optimierungen nicht so sehr heraus. Betrachtet man zusätzlich die übertragenen Daten zwischen mobilen Knoten und den Zell-Knoten (Abbildung 6.8) so wird hier deutlich, dass der Overhead durch die Zell-Wechsel-Nachrichten überwiegt. Dieser nimmt mit der Anzahl der Zellen zu. Bei einem 16x16 Gitter ist jedoch insgesamt noch kein negativer Effekt bezüglich der Effizienz bemerkbar, was die Gesamtheit der übertragenen Daten angeht. Es werden rund 98,99% der Gesamtdaten des Basis-Ansatz bei einem 16x16-Gitter übertragen. Dabei werden zwischen Zell-Knoten und mobilen Knoten rínsgesamt rund 3% weniger übertragen. Bei einer Gittergröße von 32x32 ergibt sich das Verhältnis der gesamt übertragenen Daten zwischen mobilen Knoten und Zell-Knoten

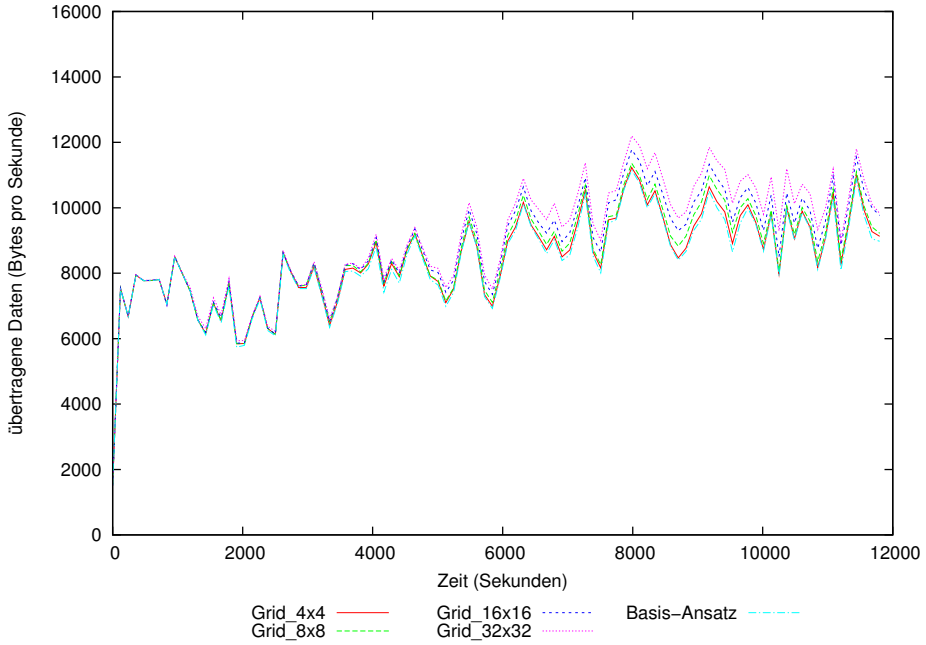

Abbildung 6.8: Messung der Effizienz, Nachrichtenlast von mobilen Knoten zu Zell-Knoten, niedrige Anfragerate, Chicago Kartendaten

zu 99,57%. Betrachtet man die von Zell-Knoten zu mobilen Knoten übertragenen Daten, so werden bei einem 32x32 Gitter im Vergleich zum Basis-Ansatz 96,76%, d.h. ungefähr 2,24%, weniger übertragen. Hier wird deutlich, dass bei einer niedrigen Anfragerate die Optimierungen kaum greifen.

Bei der Effektivität (vergleiche Abbildung 6.9). macht sich hier der im vorherigen Kapitel zur hohen Anfragerate im Abschnitt Effektivität zu niedrige gewählter Wert für den Timer des Objekt-Position-Caches bemerkbar. Trotzdem liegt die Effizienz im schlechtesten Fall bei 95% im Vergleich zum Basis-Ansatz, das heißt es werden insgesamt 5% weniger Objektpositionen gemeldet.

Das Gebiet in dem durchschnittlich nach einem Objekt gesucht wird, ist vergleichbar mit der der Gebietsgröße bei hoher Anfragerate (siehe Abbildung 6.10). Die insgesamte Verringerung des durchschnittlichen Suchgebiets zwischen 2000 und 8000 Sekunden lässt sich mit einem Anstieg der gesendeten Daten zwischen mobilen Knoten und Zell-Knoten erklären. Bei ungefähr 6800 Sekunden ist ein deutlicher Knick bezüglich der Reduzierung der Gebietsgröße zu sehen. Dieser findet sich bei den übertragenen Daten zwischen mobilen Knoten und Zell-Knoten als sprunghafter Anstieg wieder. Die mittlere Gebietsgröße bei eine 16x16 und 32x32 Gitter sind hierbei nahezu identisch,

Die Graphen zu den Kartendaten für Stuttgart weisen keine signifikanten Abweichungen

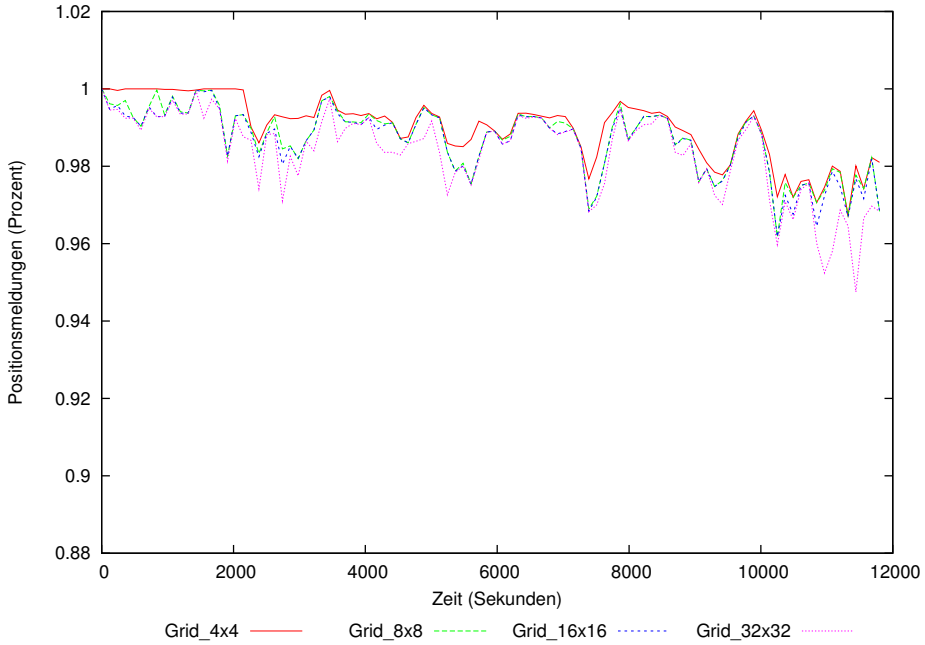

Abbildung 6.9: Messung der Effektivität, Anteil der gemeldeten Objektpositionen pro Sekunde im Vergleich zum Basis-Ansatz in Prozent, niedrige Anfragerate, Chicago Kartendaten

hierzu auf. Lediglich der effektive Suchbereich ist ein wenig kleiner, was sich analog zum Fall der hohen Suchanfragen durch die geringere maximale Objektgeschwindigkeit begründet.

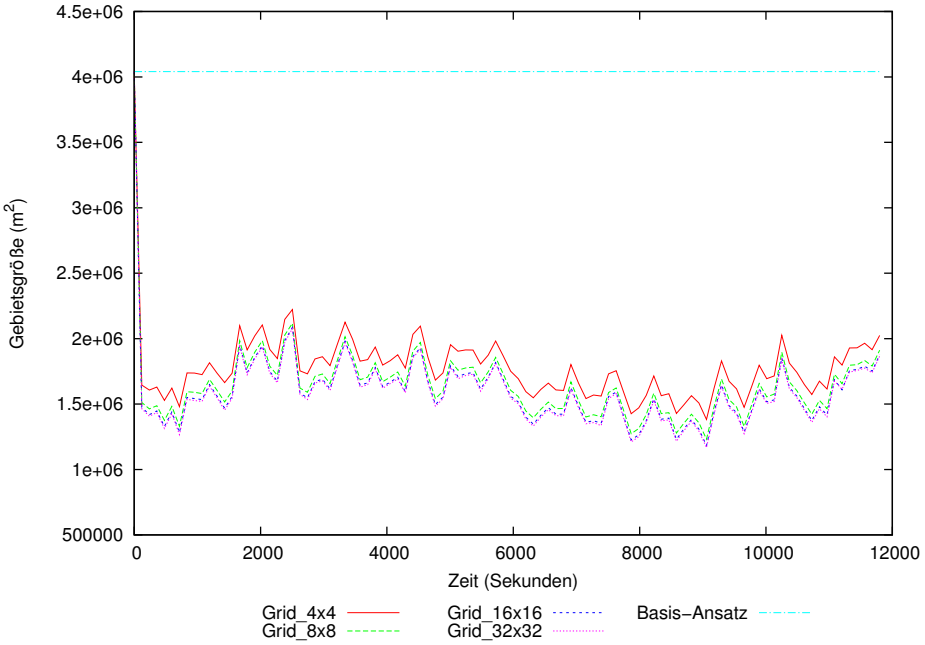

Abbildung 6.10: Messung der Effektivität, mittlere Größe des effektiven Suchbereichs in dem ein Objekt gesucht wird, minimale Größe ist dabei eine Zelle, niedrige Anfragerate, Chicago Kartendaten

6.3.3 niedrige Anfragerate mit Burst

Wir verwenden hier als Grundlage das Muster für die niedrige Anfragerate mit im Mittel 2 Anfragen pro Sekunde und einer Dauer von 60 Sekunden im Mittel. Jedoch gibt es zusätzlich eine kurze Spitzen der Anfragerate (Burst). Das Muster ist in Abbildung 6.11 zu sehen. Hierbei bedeutet ein Burst ein Anstieg der Anfragerate auf im Mittel 200 Anfragen pro Sekunde. Der Burst hat eine Dauer von 100 Sekunden. Für die Evaluierung benutzten wir wieder wie bisher die beiden Karten mit den Mobilitätsdaten der vorherigen beiden Szenarion. Hierbei betrachten wir die Ergebnisse von CanuMobiSim, also den Kartendaten für Stuttgart.

Betrachtet man Abbildung 6.13, so ist der plötzliche Anstieg der Anfragerate deutlich durch einen sprunghaften Anstieg der gesendeten Daten von den Zell-Knoten zu den mobilen Knoten zu erkennen. Vergleicht man dabei den Spitzenwert der übertragenen Daten pro Sekunde, so fällt auf, dass dieser beim Gitter-Ansatz stets niedriger ist. Ein Erklärung hierfür ist die aktive Pflegen und die Weitergabe der Objekt-Positions-Cache Einträge, auch nachdem keine Suchanfrage mehr für ein Objekt existiert. Dadurch sinkt die Größe des Gebiets in dem initial nach einem Objekt gesucht wird. Beim Basis-Ansatz wird hingegen initial stets im komplette Dienstgebiet gesucht. Der Unterschied der Übertragungsrate beträgt ca. 150 000 Bytes pro Sekunde, die beim Gitter-Ansatz weniger übertragen werden. Dies entsprich 12% weniger an übertragenen Daten zwischen Zell-Knoten und mobilen Knoten. Vergleicht man die verschiedenen Gittergrößen, so ist

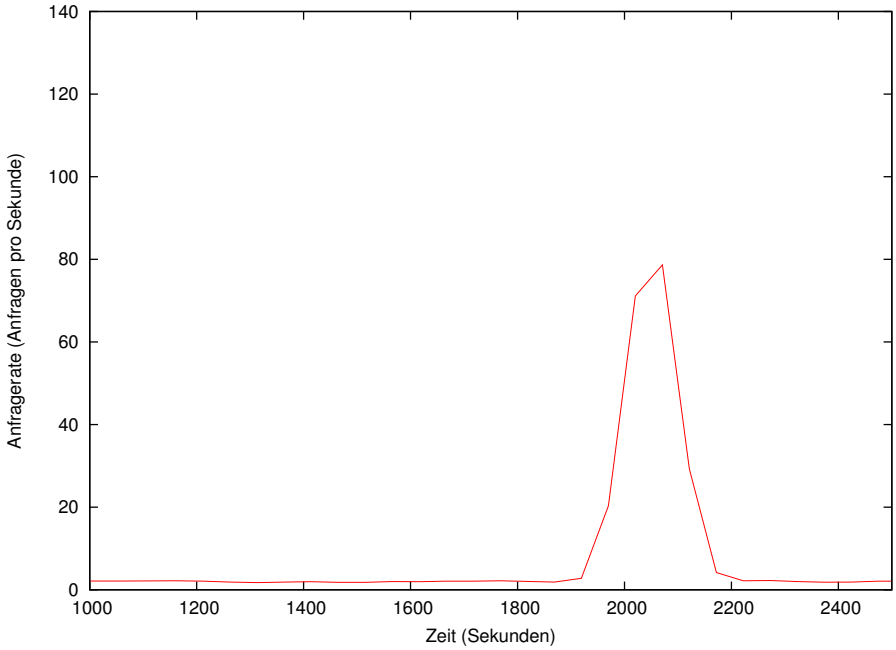

Abbildung 6.11: Anfragemuster welches eine niedrige Anfragerate kombiniert einem sprunghaften Anstieg der Anfragerate (Burst) zeigt

die Spitze um so kleiner, desto mehr Zellen vorhanden sind. Dies begründet sich wieder durch eine bessere Anpassungsfähigkeit des Suchbereichs aufgrund kleinerer Zellen.
Als Unterschied zwischen CanuMobiSim und UDel sei hier erwähnt, dass der Spitzenwert bei UDel bei 440 000 Bytes pro Sekunde lag, bei CanuMobiSim degegen lag er bei ca. 710 000 Bytes pro Sekunde. Dies begründet sich darin, dass zwar bei beiden Mobilitätsmodellen die gleiche Anzahl an mobilen Knoten verwendet wurde, jedoch sind bei UDel nicht immer alle Knoten im Dienstgebiet, daher erhalten im Gegensatz zu CanuMobiSim, grundsätzlich nicht alle mobilen Knoten einen Suchauftrag. Bei den Udel Mobilitätsdaten beträgt beim Gitter-Ansatz an der Stelle des Burst die Datenrate rund 337 000 Bytes pro Sekunde bei einem 4x4 Gitter, was 76,6% im Vergleich zum Basis-Ansatz entspricht.

Für die übertragenen Daten von mobilen Knoten zu den Zell-Knoten, lässt sich zunächst wie schon zuvor bei einer niedrigen konsten Anfragerate ein leichter Overhead erkennen, der von Zellwechseln der mobilen Knoten kommt und schon im Abschnitt zuvor bei eienr niedrigen Anfragelast beobachtet wurde. Bezieht man jedoch den Burst mit ein und betrachtet die absolute Datenmenge, die von Zell-Knoten zu mobilen Knoten übertragen wird, so beträgt diese im Vergleich zum Basis-Ansatz bei einem Gitter der Größe 16x16 rund 77,77%. Bei einer Gittergröße von 32x32 werden von Zell-Knoten zu mobilen Knoten insgesamt 76,26% des Datenvolumens im Vergleich zum Basis-Ansatz übertragen.

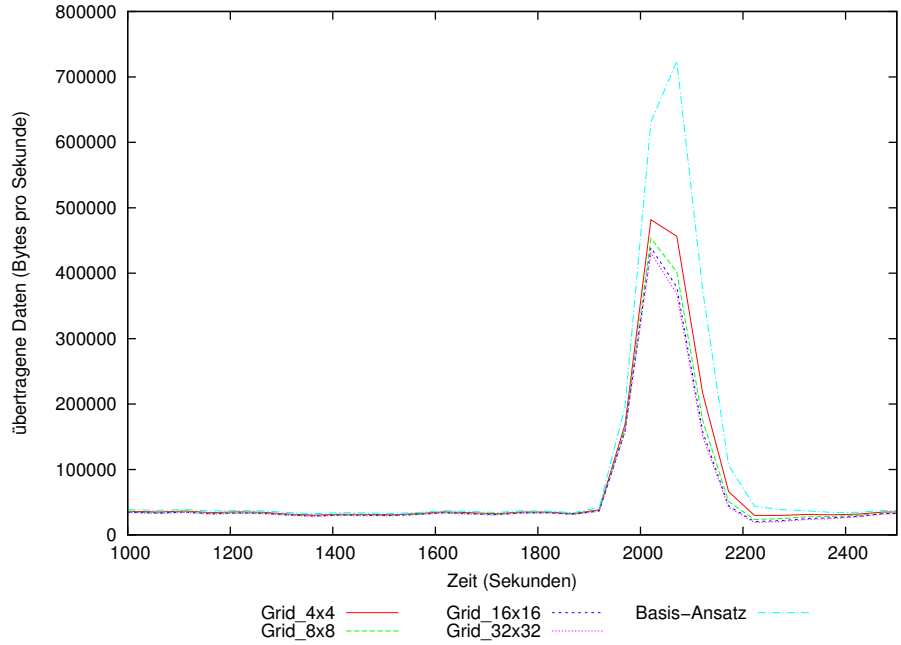

Abbildung 6.12: Messung der Effizienz, Nachrichtenlast von Zell-Knoten zu mobilen Knoten, niedrige Anfragerate mit Bursts, Stuttgart Kartendaten

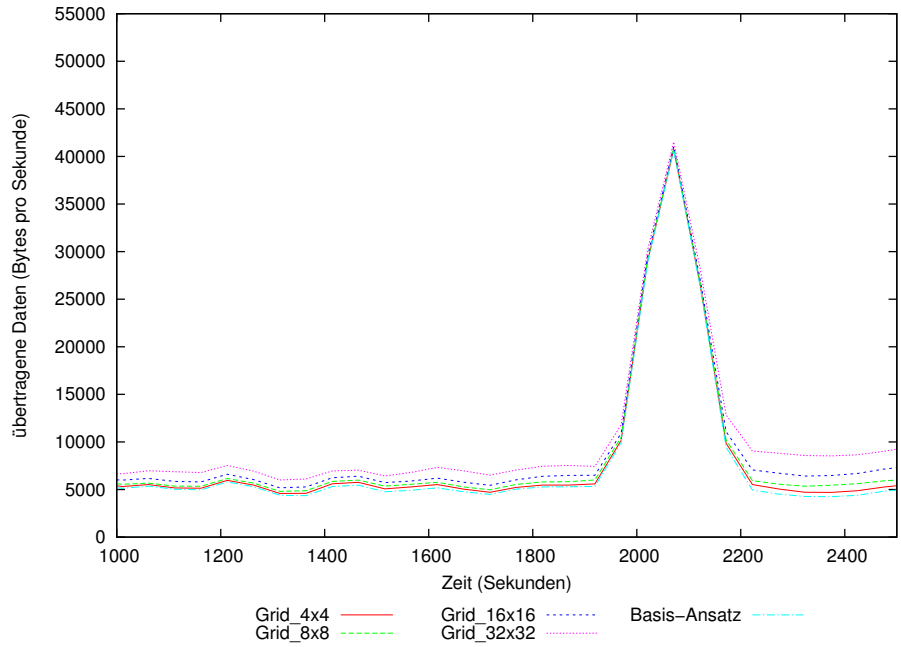

Abbildung 6.13: Messung der Effizienz, Nachrichtenlast von mobilen Knoten zu Zell-Knoten, niedrige Anfragerate mit Bursts, Stuttgart Kartendaten

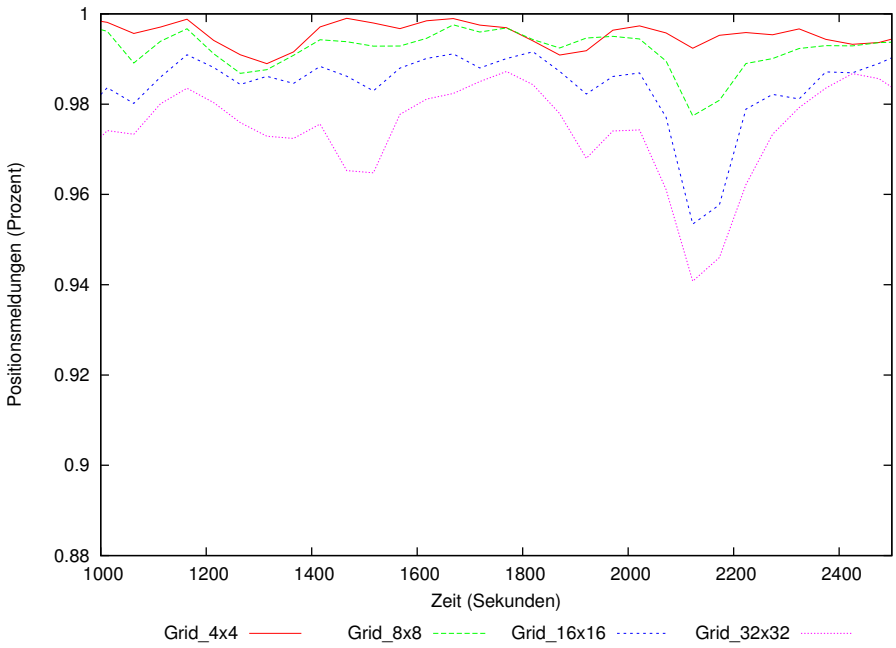

Abbildung 6.14: Messung der Effektivität, Anteil der gemeldeten Objektpositionen pro Sekunde im Vergleich zum Basis-Ansatz in Prozent, niedrige Anfragerate mit Bursts, Stuttgart Kartendaten

Betrachtete man nun die Effektivität, so liegt diese im schlechtesten Fall bei 94% im Vergleich zum Basis-Ansatz. Dabei liegt der schlechteste Wert genau beim sprunghaften Anstieg der Anfragerate, bei einem 32x32 Gitter mit 1024 Zellen. Daher liegt auch hier wieder der Schluss nahe, dass sich der Suchbereich zu langsam ausbreitet und in Folge dessen Objektfund-Nachrichten verloren gehen.
Betrachtet man den effektiven Suchbereich in Abbildung 6.15, so ist hier auch der Burst bei ca. 2000 Sekunden durch eine sprunghafte Verringerung des Gebiets sichtbar. Weiterhin bleibt die Tendenz, dass der effektive Suchbereich um so kleiner ist, desto besser sich der vom System abgeschätzte Bereich auf die Zellen abbilden lässt. Dieser Effekt nimmt mit der Anzahl der Zellen bzw. der Zunahme der Granularität des Gitters zu.

6.4 Fazit

Betrachtet man nun die vorgestellten Daten, so lässt sich generell sagen, dass der Gitter-Ansatz wo es möglich ist das insgesamt übertragene Datenvolumen zwischen Zell-Knoten und mobilen Knoten im Vergleich zum Basis-Ansatz reduziert, dabei jedoch die Anzahl der gemeldeten Objektpositionen so gut wie identisch bleibt. Sprich er realisiert die gewünschte Minimierung des Übertragenen Datenvolumens bei gleichbleibender Effektivität.

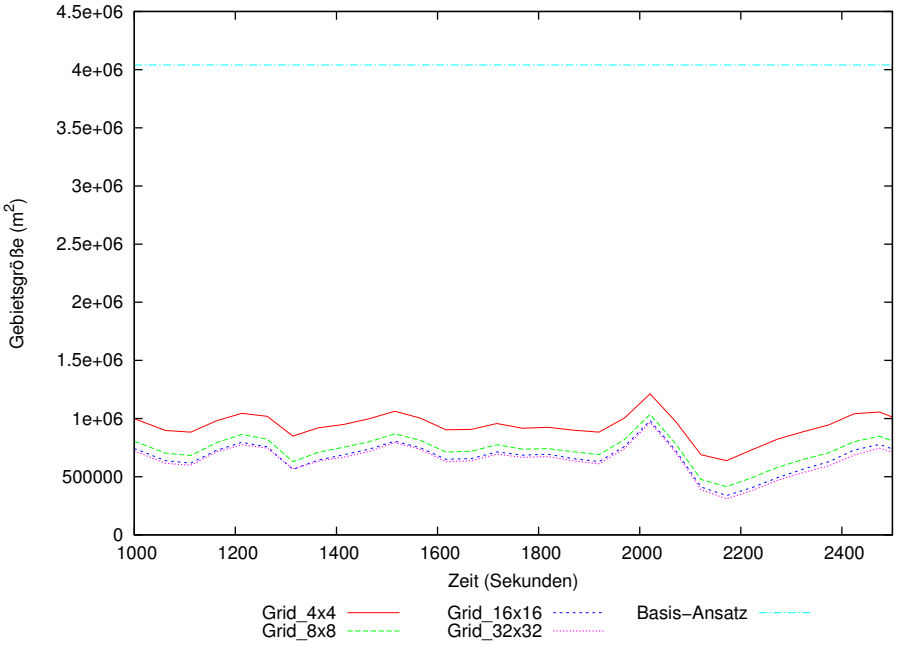

Abbildung 6.15: Messung der Effektivität, mittlere Größe des effektiven Suchbereichs in dem ein Objekt gesucht wird, minimale Größe ist dabei eine Zelle, niedrige Anfragerate mit Bursts, Stuttgart Kartendaten

Dieser Effekt kommt vor allem bei hohen Anfrageraten zum tragen, da dort zum einen das grundsätzliche Datenvolumen hoch ist und damit auch das Potenzial vorhanden ist dies durch Optimierung zu minimieren. Zum anderen trägt eine hohen Anzahl von Objektfunden dazu bei die mögliche Position eines mobilen Objekts besser einzuschätzen und in der Summe den genannten Einsparungseffekt der übertragenen Daten von Zell-Knoten zu mobilen Knoten bewirkt.
Bei niedrigen Anfrageraten macht sich der Overhead durch die Zellwechseln-Nachrichten der mobilen Knoten an die Zell-Knoten bemerkbar. Kombiniert mit dem wenigen Einsparungspotenzial der zu übertragenden Daten von Zell-Knoten zu mobilen Knoten, ergibt sich was die insgesamt übertragene Datenmenge angeht kaum eine Minimierung.
Vergleicht man beim Gitter-Ansatz die verschiedenen Gittergrößen, womit die Anzahl der Zelle gemeint ist, so sind die Einsparungseffekte was die übertragenen Daten zwischen Zell-Knoten und mobilen angeht um so stärker, desto mehr Zellen vorhanden sind. Jedoch zeigt sich, dass bei niedrigen Anfrageraten durch zu viele Zellen der Overhead durch die Abfrage des Unterschieds der gesuchten Objekte durch die mobilen Knoten überwiegt. Dies lässt sich jedoch damit begründen, dass bei einer niedrigen Anfragerate insgesamt weniger Objektfundnachrichen im Vergleich zu einer hohen Anfragerate gesendet werden. Um dies verdeutlich: Die Anzahl der Objektfundnachrichten bei der hohen Anfragerate für die UDel Mobilitätsdaten lag im Bereich von $7{,}2 \cdot 10^6$. Dagegen lag die Anzahl der Objektfundnachrichten für die gleichen Mobilitätsdaten bei der

niedrigen Anfragerate im Bereich von $1{,}53 \cdot 10^6$. Bei den Mobilitätsdaten für Stuttgart ergibt sich noch ein drastischeres Bild von $3{,}3 \cdot 10^6$ zu $403 \cdot 10^3$ was die gesamte Anzahl der erhaltenen Objektfundnachrichen betrifft.

Abschließend lässt sich sagen, dass der Gitter-Ansatz generell sehr gut funktioniert was die Minimierung der Anzahl der übetragenen Daten von Zell-Knoten zu mobilen Knoten betrifft und dies mit steigender Anzahl generell um so besser. Jedoch gibt es einen Punkt an dem die Verkleinerung der Zellgröße nur noch wenig Auswirkung hat, da der abgeschätzte Suchbereich bereits gut abgebildet werden kann. In der Simulation liegt diese Grenze bei einem Gebiet von 2x2 Kilometer bei einer Zellgröße zwischen 65 und 125 m^2, d.h. bei einer Gittergröße zwischen 16x16 und 32x32.
Ein weiterer Faktor sind die zu erwartenden Objektfundnachrichen. Gibt es davon wenige, so tritt bei einer Gittergröße, die hier ebenfalls zwischen 32x32 und 16x16 liegt, ein spürbarer Overhead durch Zellwechsel-Nachrichten der mobilen Knoten auf.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Diese Arbeit beschäftigte sich mit dem Entwurf eines Public-Sensing-Systems zur Verfolgung mobiler Objekte, bei dem Smartphones als mobile Sensorknoten dienen. Hierbei lag der Fokus auf der Minimierung des Aufwands bezüglich der Kommunikationslast bei gleichbleibender maximaler Effizienz bezüglich der gemeldeten Objektpositionen. Eine wichtige Annahme war hierbei die Gegebenheit, dass das Verhältnis der gesuchten Objekte zu allen Objekten in einem bestimmten Dienstgebiet relativ klein ist und so ein trivialer Ansatz, bei dem jedes gefunden Objekt gemeldet wird nicht den Anforderungen entspricht. Der Schlüssel zur Lösung des Problems lag bei der effizienten Verteilung von Suchaufträgen and die mobilen Knoten.

Da es in diesem Bereich bisher wenige Arbeiten gibt, wurden zunächst mögliches Systemmodell zur Verwaltung und Verteilung von Suchaufträgen entworfen. Basierende darauf wurden Methoden gesucht um die Verteilung möglichst effizient zu gestalten.

Hierbei hat sich als grundsätzlicher Mechanismus die selektive Verteilung von Suchaufträgen herauskristallisiert. Hierbei ist das Prinzip nur die mobilen Knoten mit der Suche nach einem Objekt zu beauftragen, die aus Sicht des Systems überhaupt die Möglichkeit haben das Objekt zu finden. Um Suchaufträge überhaupt selektiv verteilen zu können, wurde das Dienstgebiet in Zellen partitioniert. Dabei wurde jeder Zelle ein für sie verantwortlicher Knoten zugewiesen.

Ein weitere grundlegender Mechanismus war die Abschätzung der Objektposition auf Grundlage der bisher bekannten Positionen eines Objekts. Hierbei hat sich die Approximierung der Position durch ein achsenparalleles Rechteck als hilfreich erwiesen. Um die zeitliche Komponente einer zuletzt bekannten Position mit in die Abschätzung einzubeziehen, wurde eine einfach Form von Dead Reckoning verwendet. Dabei vergrößert sich das Rechteck proportional zur vergangenen Zeit und der maximalen Objektgeschwindigkeit.

Um die von einem mobilen Knoten gemeldete Objektposition in diesen Mechanismus einzufügen und der Tatsache gerecht zu werden, dass ein mobiler Knoten im Allgemeinen nur seine eigene Position, jedoch nicht die genau Position des gefundenen Objekts kennt, werden die gemeldeten Koordinaten durch einen mobilen Knoten zu einem Quadrat umgerechnet, wobei die Sensorreichweite mit einbezogen wird. So war es nun möglich zusätzlich den Verlauf der eintreffenden Objektpositionen auszunutzen, in

dem jeweils der Schnitt mit der bisher bekannten Position berechnet wird.

Insgesamt ergab sich hieraus eine systeminterne Abschätzung der Objektposition. Diese geschätzte Position galt es nun auf das in Zellen unterteilte Gebiet dynamisch abzubilden. Dabei bedeutet eine Abbildung auf eine Zelle die Verteilung von Suchaufträge für diese Objekt. Um diese Abbildung zu realisieren wurde der Mechanismus der dynamischen Suchbereichsanpassung entwickelt. Über diesen werden abhängig vom Aufbau der Infrastruktur die Nachbarzellen informiert entweder nun auch nach einem Objekt zu suchen oder die Suche einzustellen.

Als mögliche Strukturen wurde sowohl ein hierarchische Ansatz als auch ein Gitter Ansatz betrachtet. Für beide wurden Algorithmen entworfen um die beschriebene dynamische Suchbereichsanpassung zu realisieren. Zur Messung der Effektivität wurde zuvor außerdem ein Basis-Ansatz entwickelt, der ein Maximum an Objektfunden garantiert.
Als vielversprechendster Ansatz hat sich das Gitter gezeigt, da hier die Bereichsanpassung nur an die Aufteilung in Zellen gebunden ist. Beim hierarchischen Ansatz wurd ein unflexiblerer Quadtree verwendet.
Daher wurde der Gitter-Ansatz zusammen mit dem Basis-Ansatz im Netzwerksimulator ns2 implementeriert und evaluiert. Dabei hat sich herausgestellt, dass der Gitter-Ansatz eine zum Teil erhebliche Minimierung der Netzlast zwischen Infrastruktur und mobilen Knoten ermöglicht bei, zum Vergleich zum Basis-Ansatz, gleichbleibender Anzahl an gemeldeten Objektfunden. Und selbst und Situationen in denen wenig Minimierung der Netzlast möglich ist, dies trotzdem soweit wie möglich versucht. Auch der durch die definierten Kommunikationsabläufe zwischen mobilen Knoten und Infrastruktur entstehende Overhead blieb gering. Der Overhead war nur bei einer niedrigen Anzahl an Objektfunden im Zusammenhang mit einer Aufteilung in sehr viele Zellen relevant.

7.2 Ausblick

Der Fokus lag in dieser Arbeit auf der Entwicklung eines grundlegenden und vor allem effizienten Ansatzes zur Verfolgung mobiler Objekte mittels eines Public-Sensing-Systems. Hierbei wurde als Schwerpunkt die reine Nachrichtenlast betrachtet.
Generell wurde hier nicht die Zusammenarbeit zwischen den mobilen Knoten betrachtet, sondern nur die Kommunikation mit der Infrastruktur und die Verteilung der Suchaufträge durch diese. Auch wurde nicht die benötigte Energie der mobilen Knoten betrachtet, bzw. nur indirekt durch die Minimierung der übertragenen Daten.
Die Zusammenarbeit der mobilen Knoten um ein Gebiet möglichst energiesparsam abzudecken wurde in [WWDR11] beschrieben und stellt ein orthogonales Problem dar. Es wäre ohne weiteres möglich diesen Mechanismus hier zusätzlich anzuwenden.
Außerdem könnte man unter der Annahme, dass die direkte Kommunikation zwischen mobilen Knoten energiesparsamer ist als die Kommunikation über das Mobilfunknetz, die Verteilung von Suchaufträgen durch die Infrastruktur weiter minimieren, indem man

nur an wenige mobile Knoten in einer Zelle Suchaufträge verteilt. Diese wenigen mobilen Knoten übernehmen die weitere Verteilung dann duch direkte Kommunikation mit anderen mobilen Knoten.
Weiterhin könnte man die Positionsabschätzung weiter verbessern, in dem man den Bewegungspfad des mobilen Objektes (Trajektorie) mit einbezieht. Bestimmt man zusätzlich die Bewegungspfade der mobilen Objekte, so könnte man noch genauer abschätzen ob ein mobiler Knoten ein bestimmtes Objekt überhaupt finden kann und so die Suchaufträge noch selektiver verteilen.

Literaturverzeichnis

[Blo70] Bloom, Burton H.: Space/time trade-offs in hash coding with allowable errors. In: *Commun. ACM* 13 (1970), July, S. 422–426. – ISSN 0001–0782

[BM02] Broder, Andrei ; Mitzenmacher, Michael: Network Applications of Bloom Filters: A Survey. In: *Internet Mathematics* Vol. 1 (2002), Nr. 4, S. 485–509

[BSI+] Bohacek, Stephan ; Sridhara, Vinay ; Ilic, Andjela ; Kim, Jonghyun ; Shin, Hweechul: *UDel Mobility Model*. http://udelmodels.eecis.udel.edu

[CEL+06] Campbell, Andrew T. ; Eisenman, Shane B. ; Lane, Nicholas D. ; Miluzzo, Emiliano ; Peterson, Ronald A.: People-centric urban sensing. In: *Proceedings of the 2nd annual international workshop on Wireless internet*. New York, NY, USA : ACM, 2006 (WICON '06). – ISBN 1–59593–510–X

[CEL+08] Campbell, Andrew T. ; Eisenman, Shane B. ; Lane, Nicholas D. ; Miluzzo, Emiliano ; Peterson, Ronald A. ; Lu, Hong ; Zheng, Xiao ; Musolesi, Mirco ; Fodor, Kristóf ; Ahn, Gahng-Seop: The Rise of People-Centric Sensing. In: *IEEE Internet Computing* 12 (2008), July, S. 12–21. – ISSN 1089–7801

[CHK08] Cuff, Dana ; Hansen, Mark ; Kang, Jerry: Urban sensing: out of the woods. In: *Commun. ACM* 51 (2008), March, S. 24–33. – ISSN 0001–0782

[FBRK07] Frank, Christian ; Bollinger, Philipp ; Roduner, Christof ; Kellerer, Wolfgang: Objects Calling Home: Locating Objects Using Mobile Phones. In: *Lecture Notes in Computer Science* 4480/2007 (2007), S. 351–368

[FCAB00] Fan, Li ; Cao, Pei ; Almeida, Jussara ; Broder, Andrei Z.: Summary cache: a scalable wide-area web cache sharing protocol. In: *IEEE/ACM Transactions on Networking* 8 (2000), June, S. 281–293. – ISSN 1063–6692

[LZG+06] Lee, Uichin ; Zhou, Biao ; Gerla, M. ; Magistretti, E. ; Bellavista, P. ; Corradi, A.: Mobeyes: smart mobs for urban monitoring with a vehicular sensor network. In: *Wireless Communications, IEEE* 13 (2006), october, Nr. 5, S. 52–57. – ISSN 1536–1284

[RFH+01] Ratnasamy, Sylvia ; Francis, Paul ; Handley, Mark ; Karp, Richard ; Shenker, Scott: A scalable content-addressable network. In: *SIGCOMM Comput. Commun. Rev.* 31 (2001), August, S. 161–172. – ISSN 0146–4833

[Ste] Stepanov, Illya: *CANU Mobility Simulation Environment (CanuMobiSim)*. http://canu.informatik.uni-stuttgart.de/mobisim/

[WWDR11] Weinschrott, Harald ; Weisser, Julian ; Dürr, Frank ; Rothermel, Kurt: Participatory sensing algorithms for mobile object discovery in urban areas. In: *Pervasive Computing and Communications (PerCom), 2011 IEEE International Conference on*, 2011, S. 128–135

Erklärung

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

Unterschrift:

Offenau, den 8. Juni 2011