Inhaltsverzeichnis

1. Einleitung **9**
1.1. Aufgabenstellung . . . 9
1.2. Zielsetzung . . . 9

2. Verwandte Arbeiten **11**

3. Grundlagen **13**
3.1. Nexus . . . 13
3.1.1. Aufbau . . . 13
3.1.2. Argumented World Modeling Language (AWML) . . . 14
3.1.3. Argumented World Query Language (AWQL) . . . 15
3.2. Content Adressable Network . . . 17
3.2.1. Grundidee . . . 17
3.2.2. Aufbau . . . 18
3.2.3. Verlassen des Netzwerkes . . . 19
3.2.4. Optimierungsmoeglichkeiten, Vor- und Nachteile . . . 19
3.2.5. Zusammenfassung . . . 19
3.3. Cache . . . 20
3.3.1. Nutzen . . . 20
3.3.2. Aufbau und Funktionsweise . . . 21
3.3.3. Ersetzungsstrategien . . . 21

4. Konzeption **23**
4.1. Die Idee . . . 23
4.2. Lokaler Cache . . . 24
4.3. Aufbau des Netzes . . . 25
4.4. Anfragen über Queries . . . 27
4.5. Abwahl eines Knotens, Fehlerfall . . . 29

5. Realisierung **35**
5.1. Grundannahmen . . . 35
5.2. Peersim . . . 36
5.2.1. Konfiguration . . . 36
5.2.2. Knoten . . . 36

5.2.3. Kanten . . . 37
5.2.4. Netzwerk . . . 37
5.2.5. Protokoll . . . 37
5.2.6. Kontrollinstanzen . . . 37
5.2.7. Weitere Klassen . . . 38
5.3. Allgemeines . . . 38
5.3.1. Geometrie . . . 38
5.3.2. MGeometry . . . 40
5.3.3. AWMLReader, AWQL-Reader . . . 40
5.4. CAN-Realisierung . . . 41
5.4.1. CAN-Protokoll . . . 41
JOIN . . . 43
WELCOME . . . 45
ACCEPT, UPDATE . . . 45
ACKUPDATE . . . 46
QUERY, ROUTE . . . 46
5.5. Cache-Realisierung . . . 46
5.5.1. Cacheverwaltung . . . 47
QUERY . . . 48
REPLY . . . 50
PROCESSFINISH . . . 51
QUERYBLOCK . . . 52
5.5.2. Cache . . . 52
5.5.3. ProcessState . . . 54
5.5.4. Abweichungen . . . 55

6. Anwendungsszenarien 59
6.1. Vorbereitung . . . 59
6.1.1. SpaSe . . . 59
6.1.2. Queryerzeugung . . . 59
6.2. Programmdurchlauf . . . 62
6.3. Messinstrumente . . . 63

7. Messung und Analyse 65
7.1. Definitionen . . . 65
7.2. Messergebnisse . . . 66
7.2.1. Weiterleitungen . . . 66
7.2.2. Fülleffekt . . . 68
7.2.3. Serverzugriff . . . 70
7.3. Kostenfunktion . . . 71

8. Zusammenfassung und Ausblick 81

A. Anhang **83**
A.1. Begriffsdefinitionen 84
A.2. Mathematische Variablen 85
A.3. Widmung 86

Literaturverzeichnis **87**

Abbildungsverzeichnis

3.1. Schichtenmodell in Nexus 14
3.2. AWML-Schema 15
3.3. Splittvorgang im CAN 19
3.4. Pfadlängen bei mehrdimensionalen Koordinatenräumen 20
3.5. CLOCK-Ersetzungsstrategie 22

4.1. CAN-Beispiel 24
4.2. Weiterleitung einer Query 28
4.3. Query über mehrere Knoten 29
4.4. An- und Abmelden eines Knotens im CAN-Baum 32
4.5. Konzeptarchitektur 33

5.1. UTM-Koordinatensystem 39
5.2. QUERY-Versendemuster 50
5.3. REPLY-Versendemuster 51
5.4. Architektur der Realisierung 58

6.1. Zustand im CAN 64

7.1. Nachbarschaftsverhältnisse im CAN-Baum 67
7.2. Diagramm: Weiterleitungen pro Netzwerkgröße 68
7.3. Diagramm: Gebietswachstum 75
7.4. Diagramm: Theoretisches Gebietswachstum 76
7.5. Diagramm: Cachehits 77
7.6. Diagramm: Nachrichten 78
7.7. Diagramm: Objekte vom Cache 79

Verzeichnis der Listings

3.1. AWML-Beispiel (Quelle: [BDG+04]) 15
3.2. AWQL-Beispiel 16
3.3. AWQL-Beispiel mit NearestNeighborsOperator 16
3.4. AWQL-Beispiel mit Include-Operatoren 17

5.1. Expand-Methode im AWMLReader 41
5.2. Intersects-Methode im AWQLReader 42
5.3. Splittvorgang 44
5.4. Erhalt einer WELCOME-Nachricht 45
5.5. Erstellung der Lösungsmenge im Cache 53

6.1. Datenkopf der Bewegungsdatei 60
6.2. Startpunkte der Fahrzeuge 60
6.3. Bewegung der Fahrzeuge zu einem bestimmten Zeitpunkt (5.0) 60
6.4. Kontrollklasse 62

Verzeichnis der Algorithmen

4.1. Zwei-Kinder-Knoten-Suche 31

1. Einleitung

Mobile Geräte werden immer vielseitiger und leistungsfähiger, um Probleme des alltäglichen Lebens zu erleichtern. Das Nexus-Projekt der Universität Stuttgart bietet die Möglichkeit, Umgebungsdaten abzufragen. Hierfür wurde ein verteiltes, globales Umgebungsmodell entwickelt, welches auf verschiedenen Knoten gespeichert ist.

1.1. Aufgabenstellung

Der Zugriff an die Modelldaten in der Nexusplattform stellt einen nicht unerheblichen Aufwand dar, welcher minimiert werden soll. Vor allem im Hinblick auf die Verwendung von ressourcenbeschränkten Mobilgeräten ist besonders auf die Effizienz dieser Anfragen zu achten. Daher muss ein semantischer Cache in der Lage sein, häufig gestellte Anfragen aus einem Zwischenspeicher zu beantworten, ohne die entsprechende Anfrage an das Umgebungsmodell weiterleiten zu müssen. Hierdurch wird der Datentransfer zum Umgebungsmodell minimiert, bzw. im optimalen Fall sogar darauf verzichtet werden. Da sich aber das Umgebungsmodell auf verschiedenen Knoten befindet, muss der Cache ebenfalls in der Lage sein, auf verschiedenenen Knoten zu fungieren, ohne jedoch redundante Daten aufzunehmen, was die Kooperation zwischen diesen Knoten in den Vordergrund drängt. Insgesamt wird daher von einem verteilten, kooperativen Cache gesprochen.

1.2. Zielsetzung

Im Vorfeld sind aber einige Fragen zu beantworten. Um eine möglichst redundanzfreie Zwischenspeicherung in den semantischen Caches zu ermöglichen, muss eine Unterteilung des Datenraumes stattfinden. Da das Hauptaugenmerk der Nexusplattform auf eine ortsabhängige Anwendung liegt, liegt es nahe, die Datenraumunterteilung anhand eines geografischen Koordinatensystems zu vollziehen. In [RFH+01] wird auf dieses Problem eingegangen und untersucht.
Auch die eigentliche Datenspeicherung innerhalb eines Caches muss untersucht werden. Auf der einen Seite muss die Ressourcenbeschränkung, besonders im Speicherplatz, beachtet werden, auf der anderen Seite soll das Auffinden im Cache möglichst effizient erfolgen. Vorher muss aber noch die Frage geklärt werden, welcher Knoten in der Lage ist, einen

Cache aufzunehmen und zu verwalten. In diesem Zusammenhang ist es dann wichtig, zu entscheiden, in welcher Form die Daten im Cache hinterlegt werden. Auf der einen Seite muss versucht werden, dass sich im verteilen, kooperativen Cache möglichst viele Daten befinden, auf der anderen Seite muss sichergestellt werden, dass Anfragen effizient beantwortet werden müssen. [RD00] hat sich dieser Fragestellung angenommen. Wie in jedem ressourcenbeschränkten Konstrukt stellt sich die Frage nach der Ersetzungsstrategie, wenn der Cache sein Speichermaximum erreicht hat. Dabei muss beachtet werden, dass Anfragetypen unterschiedlich oft gestellt werden können. Ungewohnt in der Nexusumgebung ist, dass abhängig von der aktuellen Position des Benutzers zur eigentlich selben Anfrage unterschiedliche Ergebnisse liefern wird. [RD00] ist dieser Frage ebenfalls nachgegangen.
Ziel ist es, mit verschiedenen Anwendungsszenarien die Lösung der Aufgabenstellung zu verifzieren und die daraus resultierenden Messergebnisse zu analysen und eine Bewertung darüber abzugeben.

2. Verwandte Arbeiten

Laut [RFH+01] waren Napster ([Gre02]) und Gnutella ([Gnu10]) die ersten großen Vorreiter von Peer-to-Peer-Netzwerken. Hierbei werden Dateien nicht mehr über einen zentralen Serverpunkt ausgetauscht, sondern die Verbindungen werden zwischen den einzelnen Knoten (Peer) hergestellt. Der Unterschied zwischen Napster und Gnutella liegt darin, dass bei Napster eine zentrale Indixierung stattfindet, mit Hilfe dessen der Knoten mit den gesuchten Daten ermittelt wird. Zwar ist diese Variante gut skalierbar, da lediglich eine Indexstruktur geführt werden muss, jedoch ist dieser zentrale Server mit dieser Indexstruktur ein *single Point of Failure*. Fällt dieser zentrale Serverknoten aus, kann das Netzwerk nicht mehr betrieben werden. Gnutella verzichtet auf diesen zentralen Serverknoten und versucht durch Flooding die gesuchten Daten zu finden. Hierbei kann es aber passieren, dass vorhandene Daten jedoch nicht gefunden werden. Jedoch ist aufgrund der Flooding-Methode das mehr skalierbar, im Vergleich zu Napster aber robuster, da trotz Knotenausfälle das Netzwerk weiter betrieben werden kann. Um jedoch die Unterteilung des Datenraums noch skalierbarer zu machen, wird eine hash-ähnliche Prozedur verwendet, um den Knoten mit den entsprechenden Daten mit wenig Berechnungsaufwand ausfindig zu machen. Der vorhandene Datenraum wird unterteilt und auf die entsprechenden Knoten verteilt. Hierbei wird anhand einer Funktion die Datenraumpartition berechnet und entsprechend beim „zuständigen“ Knoten angefragt. Somit kann der Knoten mit den gesuchten Daten ohne Flooding ausfindig gemacht werden, was dadurch das Gnutella-System skalierbar macht. Somit ist es auch möglich, einfach den kürzesten Weg eines bestimmten Knotens zu dem „zuständigen“ Knoten zu ermitteln, da die Partitionen der Nachbarknoten ebenfalls bekannt sind. [LP08] spricht zudem von *DataCache* und *IndexCache*. Der Unterschied zwischen beiden Varianten liegt darin, dass im Falle von IndexCache der Knoten, welcher die Anfrage erhält, die Antwort in seinem Cache niederlegt und entsprechend so indixiert wird, dass diese von anderen Knoten lokalisiert werden kann. Im DataCache hingegen wird, wie bereits bei [RFH+01] beschrieben, die Antwort entsprechend der Datenraumparitionierung niedergelegt. Daher wird auf das IndexCache nicht weiter eingegangen. Der DataCache stellt somit eine gute Variante dar, um das verteilte, kooperative Caching effizient umzusetzen.

Da es sich in dieser Arbeit jedoch um Caches handelt, muss man auch davon ausgehen, dass die benötigten Informationen in gar keinen Knoten zu finden sind und eine Anfrage an das Umgebungsmodell unumgänglich ist. Es kann auch vorkommen, dass die benötigten Informationen einen größeren Bereich im Datenraum abdecken und somit mehr als einen Knoten in Anspruch nehmen. [RD00] geht auf dieses Problem ein. Wird eine Anfrage

vollständig von einem Cache ohne Anfrage an das Umgebungsmodell beantwortet, wird von einem *local hit* gesprochen. Ein *local miss* hingegen ist das gegenteilige Ergebnis. Wird mit Hilfe von anderen Knoten die Anfrage komplett beantwortet, wird von einem *global hit* gesprochen. Wird auch mit Hilfe benachbarter Knoten kein Ergebnis berechnet (*global miss*) ist eine Anfrage, bzw. Teilanfrage an das Umgebungsmodell nicht mehr zu vermeiden.

Des Weiteren wurden in [RD00] noch die Ersetzungsstrategien untersucht. Least Recently Used (LRU), Most Recently Used (MRU) und eine eigens entwickelte Ersetzungstrategie, nämlich *Furthest Away Replacement*(FAR). Da sich mobile Geräte meist bewegen, wird neben der aktuellen Position auch die Bewegungsrichtung bei der Ersetzungsstrategie beachtet und örtlich weitentfernte Objekte aus dem Cache entfernt. Simulationen haben ergeben, dass sich diese Ersetzungsstrategie bei örtlich gebundenen, semantischen Anfragen gegenüber den anderen Ersetzungsstrategien durchgesetzt hat. Da jedoch [RD00] davon ausgeht, dass nur ein Cache vorliegt, muss noch abgeschätzt werden, inwiefern die vorgestellten Ersetzungsstrategien für ein verteiltes, kooperatives Caching eignet.

Datenräume können als logisches, mehrdimensionales karthesisches Koordinatensystem aufgefasst werden. [RFH+01] unterteilt diesen Datenraum in Bereiche, jedoch sind vor allem mehrdimensionale Datenräume schwer zu verwalten, da auch die Anzahl der Nachbarknoten zunimmt. [KW08] versucht, dem entgegenzuwirken und führt den Methode der *Space Filling Curves* ein. Hierbei wird versucht, sich durch den Datenraum „durchzuschlängeln“ und dadurch den mehrdimensionalen Datenraum auf eine Dimension zu drücken. Durch die Eindimensionalität ist es dann auch möglich, mittels herkömmlicher Hashtabellen den Knoten mit den gesuchten Informationen zu ermitteln.

3. Grundlagen

3.1. Nexus

Nexus ist ein Projekt, welches seit 2000 in der Entwicklung steht. Im Mittelpunkt des Nexus-Projekts steht ein Umgebungsmodell, mit Hilfe dessen man Informationen über reale Objekte erlangen kann. [BDG+04] Im Folgenden werden der Aufbau, das Datenmodell und die Anfragesprache beschrieben, weiterführende Details zu Nexus sind in [Nex] zu finden.

3.1.1. Aufbau

Nexus ist in drei Schichten unterteilt, der Serviceschicht, der Föderationsschicht und der Anwendungsschicht. In der untersten Ebene ist die Serviceschicht zu finden, welche lokale Kontextdaten anbindet. Die Serviceschicht beinhaltet verschiedene Arten von Kontextinformationen, wie zum Beispiel die Repräsentation von realen Objekten, wie Häuser oder Gelände, sowie virtuelle Daten, zum Beispiel Links zu Webseiten im Internet. Die Serviceschicht beinhaltet *Kontextserver*, welche die benötigten Daten speichern. Auch das Internet an sich ist als ein derartiger Server anzusehen.
Die darüberliegende Schicht ist die Föderationsschicht, welche in der Lage ist, Daten der Serviceschicht aufzubereiten und zusammenzufassen. Es stellt auch die Verbindung zwischen dem Benutzer in der Anwendungsschicht und der Serviceschicht dar. Die Föderationsschicht hat die gleichen Schnittstellen wie ein Kontextserver in der darunterliegenden Serviceschicht, ist jedoch nicht in der Lage, Informationen zu speichern. Die Hauptaufgabe der Förderationsschicht besteht darin, Anfragen des Benutzers entgegenzunehmen, diese zu analysieren, und daraus resultierend die Anfrage an die Contextserver in der Serviceschicht zu stellen.
Die oberste Schicht ist die Anwendungsschicht. Die Anwendungsschicht ist eine Software, welche die Nexusplattform verwendet. Als Anwendung können beispielsweise mobile Endgeräte in Frage kommen. Für die Anwendungsschicht bestehen drei grundlegende Möglichkeiten, die Nexusplattform zu verwenden: zum einen können Anfragen gestellt werden, um Informationen über die Umgebung oder andere kartografischen Bereiche zu erlangen, desweiteren können neue Daten oder Ereignisse eingepflegt werden, um diese zugänglich zu machen, und letztlich können Dienstleistungen in Anspruch genommen werden, für welche die bekannten Daten zu Grunde gelegt werden, wie zum Beispiel das Navigieren. Abbildung 3.1 zeigt den schematischen Aufbau.

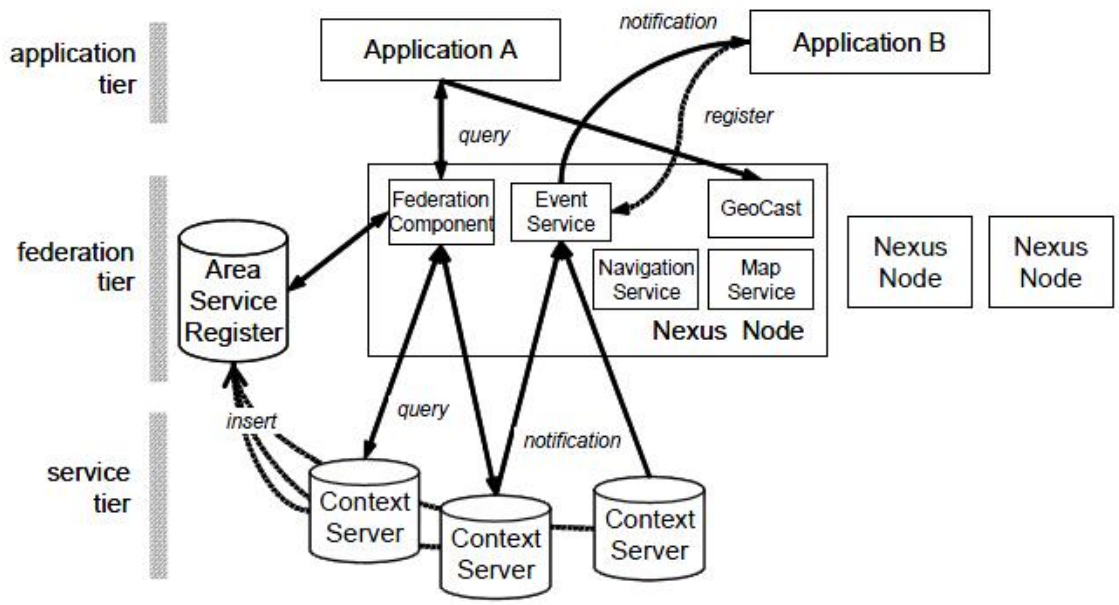


Abbildung 3.1.: Schichtenmodell in Nexus (Quelle: [BDG+04])

3.1.2. Argumented World Modeling Language (AWML)

Die Nexusplattform verwendet XML zum Austausch von Daten. Auf XML basierend ist die eigens entworfene Sprache *Argumented World Modeling Language*. AWML repräsentiert und beinhaltet die Daten der Realwelt.[1] AWML ist gekennzeichnet durch eine Vielzahl von Typen und Attributen, welche in einem vordefinierten Namensschema, dem „Nexus Standard Class Schema“ definiert sind. Abbildung 3.2 zeigt den strukturellen Aufbau eines AWML-Dokumentes. Mit Hilfe des AWML-Dokumentes können somit Informationen über ein Objekt beschrieben werden, wie zum Beispiel die Adresse eines Hauses oder die Position von Kreuzungen.

Das Beispiel 3.1 zeigt eine AWML-Beispiel. Das Beispiel beschreibt ein Gebäude in der Universitätsstraße in Stuttgart. NOL repräsentiert eine eindeutige Objekt-ID. Das beschriebene Gebäude kann mit weiteren Attributen ergänzt werden.

[1] im Original: Argumented World

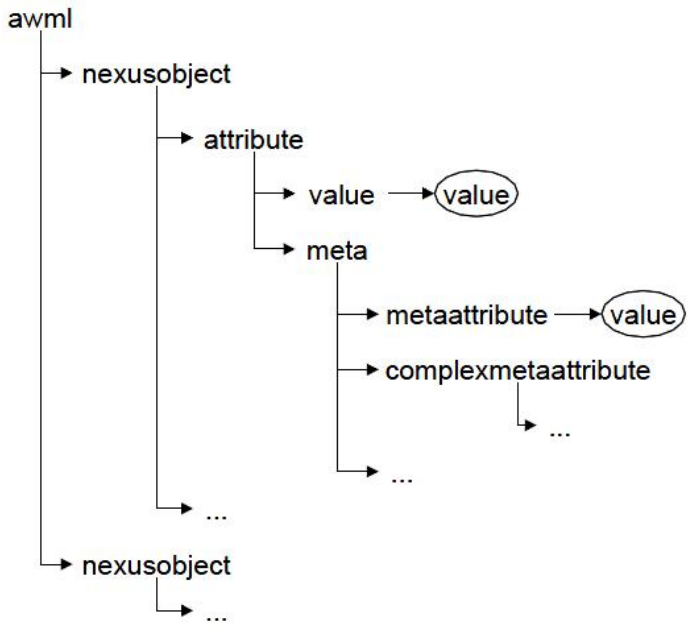


Abbildung 3.2.: AWML-Schema (Quelle: [BDG+04])

Listing 3.1 AWML-Beispiel (Quelle: [BDG+04])

```
<awml>
  <nexusobject>
    <type><value>Building</value></type>
    <NOL><nsas:value>...some NOL...</value></NOL>
    <address>
      <value>
        <street>Universitätsstrasse<street>
        <city>Stuttgart<city>
      <value>
    <address>
  <nexusobject>
</awml>
```

3.1.3. Argumented World Query Language (AWQL)

Die Anfragesprache, welche in der Nexusplattform verwendet wird, wird *Argumented World Query Language*, kurz AWQL, genannt. Auch AWQL ist auf XML aufbauend und dient dazu, Anfragen an die Kontextserver zu stellen. Da logische Operatoren verwendet werden können, kann eine AWQL-Abfrage sehr verschachtelt und daher kompliziert wirken. Eine AWQL-Anfrage beinhaltet Restriktionen, welche boolsche Ausdrucke über Prädikate sind. Zulässige Boolsche Ausdrücke sind *AND*, *OR* und *NOT*. Alle Prädikate dienen einfachen

Listing 3.2 AWQL-Beispiel

```
<awql>
 <restriction>
  <and>
   <equal>
    <target>type</target>
    <value>restaurant</value>
   </equal>
   <intersects>
    <target>extent</target>
    <value> ... Koordinaten ... </value>
   </intersects>
  </and>
 </restriction>
</awql>
```

Listing 3.3 AWQL-Beispiel mit NearestNeighborsOperator

```
<awql>
 ...
 <restriction>
  ...
 </restriction>
 <nearestNeighbors num="'5"'>
  <target>pos</target>
  <referencePoint>
   <WKT> ... WKT-Koordidaten ... </WKT>
  </referencePoint>
 </nearestNeighbors>
</awql>
```

Vergleichne, wie zum Beispiel *EQUALS* oder *INTERSECTS*.

Das Beispiel 3.2 zeigt ein unvollständiges Beispiel über eine AWQL-Abfrage. Diese Abfrage sucht nun alle Objekte, welche einem Restaurant entsprechen und in einem bestimmten Gebiet liegen.

Neben den RestrictionOperatoren gibt es noch den *NearestNeighborsOperator* , welcher die Möglichkeit bieten soll, eine Anfrage über Objekte in unmittelbarer Nähe des aktuellen Standortes zu definieren. Der NearestNeighborsOperator kann wiederum einen Restriction-Operator beinhalten, um auch diese Anfrageart sehr detailliert festzulegen. Eine Kombination beider Operatoren zeigt das Beispiel 3.3, welches auf 3.2 aufbaut.

Das Beispiel 3.3 sucht die nächsten fünf Objekte um die angegebenen WKT-Koordidaten, welche im RestrictionOperator definiert sind. Wird der RestrictionOperator aus Beispiel 3.2 verwendet, werden die fünf nächsten italienischen Restaurants gesucht.

Listing 3.4 AWQL-Beispiel mit Include-Operatoren

```
<awql>
 ...
 <restriction>
  ...
 </restriction>
 <nearestNeighbors num="'5"'>
  ...
 </nearestNeighbors>
 <include>
  <target>address</target>
 </include>
 <include>
  <target>menu</target>
 </include>
</awql>
```

In einer AWQL-Abfrage können auch include- und exclude-Definitionen festgelegt werden, also welche Informationen über ein Objekt zurückgeliefert werden sollen. Werden keine Beschränkungen gemacht, werden alle vorhandenen Daten über die Objekte zurückgeliefert. Die Übergabe eines Include-Parameters ist vergleichbar mit den Selektierung in einer SQL-Abfrage.

Das Beispiel 3.4 liefert zunächst, wenn die vorherigen Beispiele mit einbezogen werden, die Adressen und Speisekarten der naheliegendsten fünf italienischen Restaurants.

3.2. Content Adressable Network

Das Content Adressable Network (CAN) ist ein Peer-to-Peer-Netzwerk, welches Funktionalitäten einer verteilten Hashtabelle (Distributed Hashtable, DHT) bietet.

3.2.1. Grundidee

Weder der zentrale Server noch das Flooding sind gute Voraussetzungen, für ein Peer-to-Peer-Netzwerk. In [RFH+01] wird ein Peer-to-Peer-Netzwerk vorgestellt, welches auf diese zwei Elemente verzichtet. Die Grundidee besteht darin, sich die bereits bekannte und effektive Datenstruktur von Hashtabellen in verteilten Systemen zu Nutze zu machen. Der Datenraum wird mit einem logischen Koordinatensystem aufgeteilt, wobei es freigestellt wird, mit welchen Datenbestandteilen dies vorgenommen wird. Daten werden dann mit einem Hashwert in diesem Koordinatensystem abgebildet. Die Knoten, welche sich dann in diesem Peer-to-Peer-Netzwerk befinden, teilen dann diesen Datenraum unter sich auf.
Die Datenraumpartitionierung kann dadurch verfeinert werden, dass ein n-dimensionaler,

verteilter Adressraum verwendet wird. Dadurch steigt zwar der Aufwand, um das Content Adressable Network zu verwalten, jedoch wird dadurch auch erreicht, dass Objekte, welche innerhalb der gleichen Datenraumpartition sind, mehr Gemeinsamkeiten haben. Im Extremfall kann für jedes Attribut eine eigene Dimension verwendet werden, was bedeutet, dass alle Objekte, welche sich innerhalb einer Datenraumpartition befinden, die Lösungsmenge der Anfrage sind und somit keine weitere Suche innerhalb einer Datenraumpartition nötig ist.

3.2.2. Aufbau

Alle Knoten, welche sich im Netzwerk befinden, teilen sich den Datenraum untereinander auf. Bei der Initialisierung des Netzes muss somit ein Knoten den ganzen Datenraum verwalten. Dieser Zustand ist vergleichbar mit Napster. Bei einer Anmeldung muss ein Knoten gefunden werden, welcher bereits im Netzwerk integriert ist. Dieser Knoten wird Einstiegspunkt genannt. Von dort aus muss dann ein Knoten gefunden werden, welcher seinen Datenraum mit dem neuen Knoten teilt. Dieser Datenraum wird im Koordindatensystem mittig geschnitten, wobei ein Knoten immer abwechselnd über alle Dimensionen splittet. Im zweidimensionalen Raum wird somit immer abwechselnd horizontal und vertikal geschnitten. Jeder Knoten im Netzwerk kennt seinen Datenraum und auch den Datenraum seiner direkten Nachbarn. Bei einer Splittung werden dem neuen Knoten somit auch die direkten Nachbarn vom gesplitteten Knoten mitgteilt. Der neue Knoten „stelltßich dann seinen direkten Nachbarn vor, welche dann ihre Nachbarinformationen aktualisieren. Eine Nachricht, zum Beispiel eine Suchanfrage, beinhaltet immer die Koordindaten des Zielpunktes. Da sich das Content Adressable Network sich die Hashstrukturen zu Eigen macht, kann mittels einer (Hash-)Funktion immer ein Zielpunkt berechnet oder ermittelt werden. Eine Nachricht mit einem festen Zielpunkt wird somit immer zu dem Nachbar weitergeleitet, welcher dem Zielpunkt am nächsten liegt. Daher ist es auch notwendig, dass jeder Knoten Informationen über den Datenraum seiner Nachbarn haben muss. Es gibt immer mehrere mögliche Pfade zum Zielpunkt, was die Ausfallsicherheit erhöht.
Die Ermittlung des Einstiegspunktes und des Knoten, welcher gesplittet wird, kann individuell gestaltet werden. So kann das Netzwerk beispielsweise mehrere „Rendezvous Hosts“ besitzen, welcher über einen Domainnamen angesprochen wird. Wenn sich ein neuer Knoten sich mit dem Rendezvous Host verbunden hat, ermittelt dieser den Knoten, welcher dann gesplittet wird. Dies kann zum Beispiel dadurch geschehen, dass der Rendezvous Host einen zufälligen Punkt im Netzwerk bestimmt, zu welchem dann hingeroutet wird. Der Knoten, welcher diesen Punkt auf seinen Datenraum abbildet, erhält eine Anfrage des neuen Knotens, welcher zum Spitten führt. Nach dem Splitten die eigenen Routingtabellen aktualisiert und danach eine Update-Nachricht an die direkten Nachbarn geschickt. Abbildung 3.3 zeigt beispielhaft, wie ein Splittvorgang durchgeführt. Der Knoten 7 möchte sich in das Content Adressable Network einwählen und stellt eine Anfrage diesbezüglich, Knoten 1 wird für den Splittvorgang ausgewählt. Dieser teilt seinen zugewiesenen Datenraum auf und übergibt

einen Teil davon an den Knoten 7, welcher den Platz hierfür einnimmt. Die Knoten 2, 3, 4 und 5 werden über den durchgeführten Splittvorgang noch informiert.

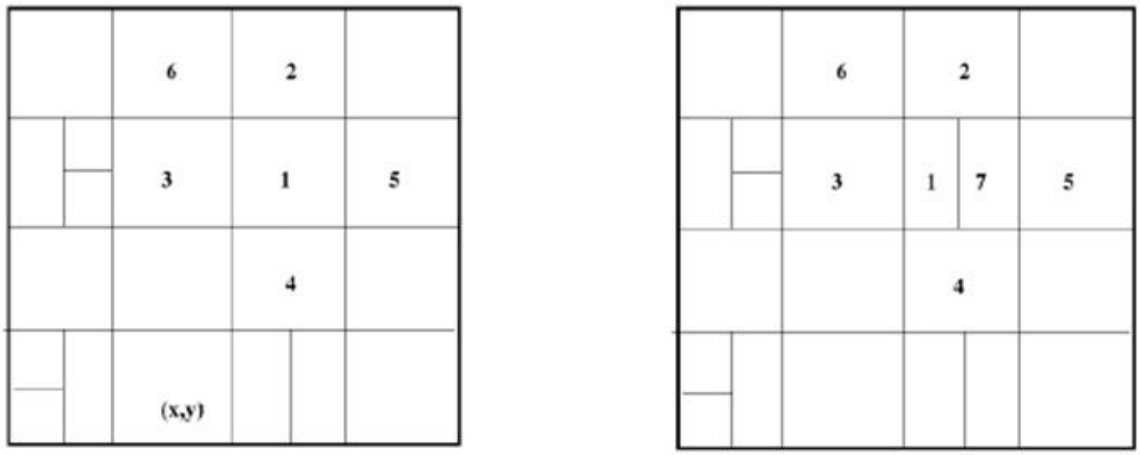


Abbildung 3.3.: Splittvorgang im CAN (Quelle: [RFH+01])

3.2.3. Verlassen des Netzwerkes

Ein Knoten kann aus verschiedenen Gründen das Netzwerk verlassen. Die zwei häufigsten Vorkommnisse, welche zu einer Abwahl führen ist das „freiwillige“ Abmelden und das Abmelden durch einen vorangegangenen Fehler. Um Fehler in einem Knoten zu erkennen, werden in regelmäßigen Abständen zwischen benachbarten Knoten *Heartbeat-Nachrichten* verschickt. Wird eine Abmeldung eines Knotens erkannt, wird der freigewordene Bereich von seinem seiner Nachbarn übernommen.

3.2.4. Optimierungsmoeglichkeiten, Vor- und Nachteile

Die durchschnittliche Pfadlänge kann verkürzt werden, wenn ein mehrdimensionaler Koordinatenraum verwendet wird. Da sich aber dadurch die Anzahl der Nachbarn erhöht, müssen in den Knoten auch größere Routingtabellen geführt werden. Durch mehr Nachbarn wird auch die Anzahl möglicher Pfade zu einem Zielpunkt erhöht, was sich positiv auf die Ausfallsicherheit auswirkt.

3.2.5. Zusammenfassung

Content Adressable Networks sind vollständig verteilt, dies bedeutet, dass der Datenraum zu jeder Zeit vollständig auf die Mengen der angemeldeten Knoten verteilt ist. Es bedarf keiner

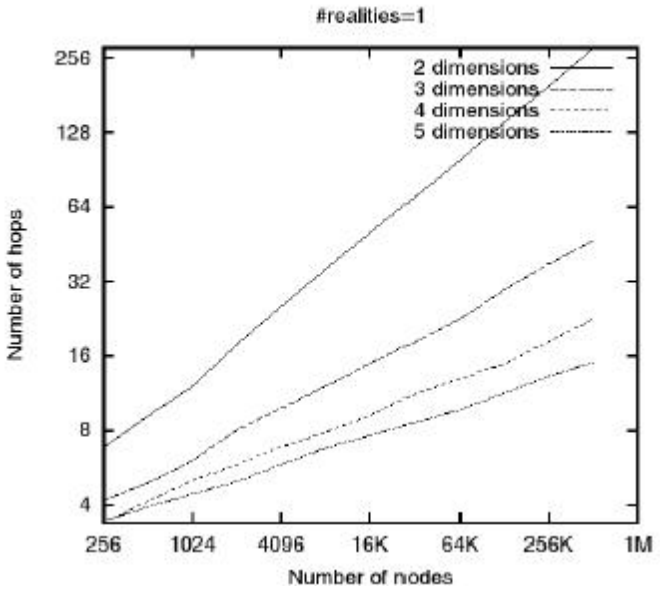


Abbildung 3.4.: Pfadlängen bei mehrdimesionalen Koordinatenräumen (Quelle: [RFH+01])

zentralen Steuerung oder Konfiguration. Das Netzwerk ist zudem skalierbar, da ein Knoten nur Informationen über seine Nachbarknoten speichert, jedoch keine globale Informationen besitzt. Dadurch kann das Netzwerk jederzeit vergrößert oder verkleiniert werden. Da es verschiedene Pfadvarianten gibt, um einen Zielpunkt zu erreichen, können fehlerhafte Knoten umgangen werden. Das Content Adressable Network ist somit fehlertolerant.

3.3. Cache

Einen Cache bezeichnet man in der elektronischen Datenverarbeitung eine Methode, um Inhalte, welche bereits einmal vorlagen, beim nächsten Zugriff schneller zur Verfügung zu stellen. Einer der bekanntesten Cache auf Heimrechner sind unter Windows die „Internet Temporary Files“, um Webseiten schneller zu laden.

3.3.1. Nutzen

Die Ziele eines Cache beruhen auf die Verkürzung der Zugriffszeit oder die Verringerung der Zugriffe auf das eigentliche Speichermedium. Somit sind Caches nur dort lohnenswert, wo die Zugriffszeit oder die Anzahl der Zugriffe sich stark auf die Gesamtleistung eines Systems auswirken. Ein weiterer Vorteil beim Cachen liegt darin, dass die Verwendung der

Schnittstelle zur nächst höheren Speicherebene in der Speicherhierarchie reduziert werden kann. Dies ist vor allem dann sinnvoll, wenn die Verwendung der Schnittstelle Kosten verursacht, wie zum Beispiel die Verwendung vom Internet.

3.3.2. Aufbau und Funktionsweise

Ein Cache hat eine meist kleine Speichergröße, um die Zugriffszeiten möglichst kurz zu halten. Wird nun eine Anfrage über Daten an ein Speichermedium gestellt, bei dem ein Cache vorgeschaltet ist, wird zuerst der Cache durchsucht, um die Daten zu lokalisieren, schlägt dies fehl, muss der *Hintergrundspeicher* dann die Anfrage entgegennehmen. Je kleiner die Cachegröße ist, desto schneller ist die Zugriffszeit, jedoch können auch weniger Daten aufgenommen werden, die gecacht werden. Auf der anderen Seite verlängert ein großer Cache die Zugriffszeit, aber die Anzahl der gespeicherten Daten nimmt zu. So ist es notwendig, einen Kompromiss zwischen diesen beiden Metriken zu finden. Um einen Cache möglichst effizient zu gestalten, werden die Lokalitätseigenschaften der Zugriffsmuster ausgenutzt. In vielen Fällen verwenden Programme nur eine kleine Anzahl von Variablen, auf welche aber immer sehr oft zugegriffen werden. Diese Variablen werden somit versucht, im Cache zu halten, um eine möglichst geringe Zugriffszeit zu verwirklichen. Durch das Zugriffsverhalten von Software können diese Lokalitätseigenschaften ermittelt werden.
Der Name des semantischen Cachings folgt aus dem Bezug zur Semantik der gespeicherten Daten, die in Verwaltung und Anfragebearbeitung einbezogen wird. Beim semantischen Caching werden Daten zusammen mit anderen Daten gespeichert, die die Cache-Inhalte semantisch beschreiben. Die Überprüfung des Cache-Inhaltes auf einen Cache-Treffer oder Cache-Fehler erfolgt anhand dieser beschreibenden Metadaten. Die eigentlichen Daten werden dafür nicht betrachtet [Sem03]. Eine Anfrage in einem semantischen Cache wird mit Hilfe von Prädikaten bearbeitet, Zwei Prädikate sind hierbei von besonderer Bedeutung, nämlich die Relationsprädikate, welche Attribute repräsentieren, die in die Ergebnismenge einfließen sollen, im Falle von Nexus somit die include-Attribute und die Selektionsprädikate, welche die Bedingung repräsentieren, die erfüllt sein müssen, damit ein Objekt in die Lösungsmenge aufgenommen wird. [HS04].

3.3.3. Ersetzungsstrategien

Es gibt eine Vielzahl verschiedener Ersetzungsstrategien. Die beste Ersetzungsstrategie ist immer abhängig von der Laufzeitumgebung, hier werden die wichtigsten Ersetzungsstrategien vorgestellt:

- *FIFO (First In, First Out):* Die ältesten Einträge werden zuerst aus dem Cache gelöscht, unabhängig davon, wie oft auf den zu löschenden Eintrag zugegriffen worden ist. Diese Ersetzungsstrategie hat den Vorteil, dass sich kein Eintrag im Cache festsetzen

kann, nachteilig ist jedoch, dass häufig verwendete Einträge auch gelöscht werden können, obwohl andere, neuere Einträge nur einmal verwendet worden sind.

- *LRU (Last Recently Used):* Diese Ersetzungsstrategie beruht darauf, dass die Einträge aus dem Cache verdrängt werden, welche am längsten nicht mehr verwendet worden sind. Somit verbleiben Einträge, welche häufig verwendet werden im Cache, während selten oder nur einmalig verwendete Einträge vorher gelöscht werden.
- *LFU (Last Frequently Used):* Diese Ersetzungsstrategie entfernt Einträge aus dem Cache, welche am seltensten verwenden worden sind. Somit können Einträge, welche nur für einen einmaligen Zugriff verwendet worden sind, schnell wieder aus dem Cache entfernt werden. Der Nachteil ist aber, dass sich Einträge im Cache festsetzen können, wenn sie zu einem früheren Zeitpunkt oft verwendet worden sind und somit die Anzahl der Zugriffe in einem Bereich liegt, welcher nicht mehr oder nur erst in geraumer Zeit, erreicht wird. Um neue Einträge vor dem Löschen zu schützen, werden nur Einträge aus dem Cache entfernt, welche ein bestimmtes, vorher definiertes, Mindestalter besitzen.
- *CLOCK (Second Chance):* Eine Verbesserung zur FIFO-Ersetzungsstrategie. Für jeden Eintrag wird ein Referenzbit geführt. Wird auf einen Eintrag zugegriffen, wird dieses Referenzbit auf 1 gesetzt. Muss ein Eintrag aus dem Cache entfernt werden, werden die Einträge durchlaufen. Wird ein Eintrag mit dem Referenzbit 1 gefunden, wird dieses Bit auf 0 gesetzt, es bekommt somit seine „zweite Chance“, ist das Referenzbit eine 0, wird dieser Eintrag ersetzt. Der Zeiger wird auf den nächsten Eintrag geschoben und verbleibt dort. Bei dem nächsten Suchlauf nach einem zu ersetzenden Eintrag beginnt sie an dieser Zeigerposition. Das Beispiel in Abbildung 3.5 zeigt die Funktionsweise.

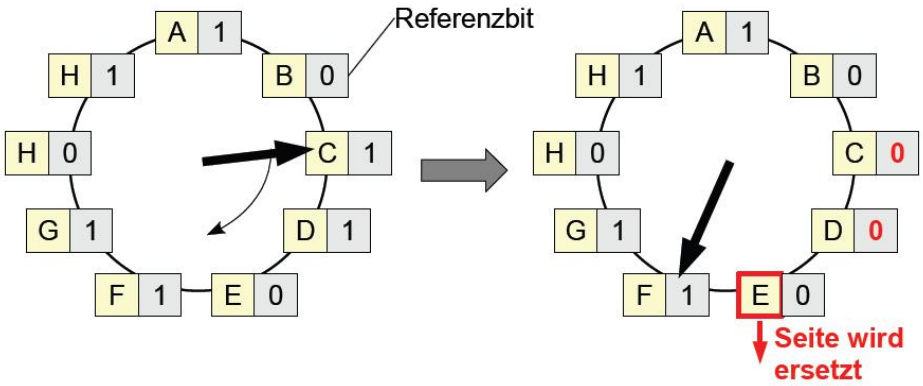


Abbildung 3.5.: CLOCK-Ersetzungsstrategie (Quelle: [Spio8])

4. Konzeption

> Plant das Schwierige da, wo es noch leicht ist. Tut das Große da, wo es noch klein ist. Alles Schwere auf Erden beginnt stets als Leichtes. Alles Große auf Erden beginnt stets als Kleines.
>
> *(Laotse - chin. Philosoph)*

In diesem Kapitel wird das Konzept vorgestellt, welches dieser Diplomarbeit zu Grunde liegt. Alle verwendeten Techniken sind im Kapitel „Grundlagen“ vorgestellt worden.

4.1. Die Idee

Ein Cache speichert eine Teilmenge eines zugrunde liegenden Speichermediums, wie beispielsweise das Internet. Werden Daten aus dem Hintergrundspeicher benötigt, wird zuerst im Cache geprüft, ob diese Daten oder Informationen bereits vorliegen, ansonsten werden die benötigten Daten aus dem Hintergrundspeicher geladen und in den Cache abgelegt. Somit bildet der Cache eine echte Teilmenge des Hintergrundspeichers. Bei einem verteilten Cache kommt noch die Schwierigkeit hinzu, die Inhalte der Caches so zu verwalten und zu speichern, dass möglichst wenig redundate Daten aufkommen.
Ein Lösungsansatz wäre, alle Knoten, welche den verteilten Cache realisieren, zu durchsuchen. Doch schon ab einer kleinen Anzahl von Knoten wird ein Skalierungsproblem festgestellt, da die Anfrage mittels Flooding an alle Knoten weitergeleitet werden müsste. Daher kann dies bereits bei kleinen Netzwerkgrößen eine längere Anfragedauer implizieren, als die direkte Bearbeitung einer Anfrage im Hintergrundspeicher. Es muss daher versucht werden, den Datenraum so zu partitionieren, dass bereits bei der Anfrage klar sein muss, welcher Knoten diese Informationen parat haben könnte. Dieses Partitionierungsprinzip des Content Adressable Network erfüllt diese Anforderungen. Da in der Nexusplattform oft Objekten gearbeitet wird, bei denen geografische Daten, zum Beispiel GPS-Koordinaten, vorliegen, kann die Unterteilung des Datenraumes anhand dieser geografischer Daten vorgenommen werden. Dies bedeutet, dass eine Anfrage, welche einen geografischen Raum betreffen auf einen bestimmten Knoten abgebildet werden kann. Knoten, welche diesen geografischen Bereich nicht auf ihrem Datenraum abbilden können, können aufgrund der räumlichen Partitionierung keine Daten diesbezüglich in ihren Caches haben. Wird in Abbildung 4.1

eine Anfrage an den Punkt $(4, 7)$ gestellt, so kann nur der Knoten D diese Informationen in seinem Cache gelagert haben. Da jeder Knoten immer über den Abbildungsbereich seines Nachbarn informiert ist, ist auch zu jedem Zeitpunkt bekannt, an welchen Knoten eine Anfrage bei Bedarf weitergeleitet werden muss, um den Knoten D zu erreichen.

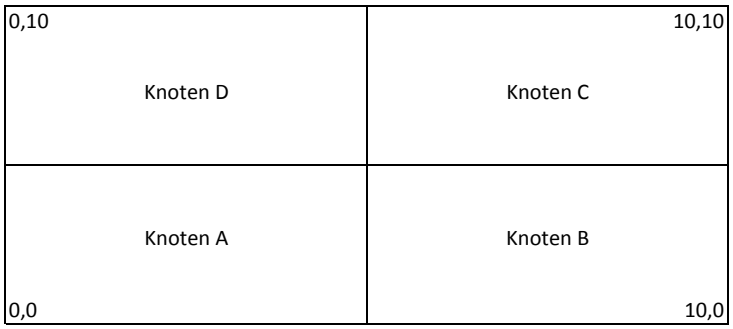


Abbildung 4.1.: CAN-Beispiel

4.2. Lokaler Cache

Jeder Knoten, welcher sich im Peer-to-Peer-Netzwerk befindet, und die entsprechenden Ressourcen zur Verfügung hat, verwaltet einen lokalen Cache. Da die Nexusplattform die Objekte und Informationen in der AWML führt, muss somit der Cache in der Lage sein, diese AWML zu verwalten. Diese Klasse enthält die Informationen, welche bei einer Anfrage an den Cache überprüft werden müssen, ob das gesuchte Objekt im Cache liegt. Ist dies nicht der Fall, wird der Hintergrundspeicher, im Fall von Nexus die Kontextserver in die Bearbeitung der Anfragen mit einbezogen.
Der lokale Cache benötigt eine sinnvolle Ersetzungsstrategie, um im Falle eines Überlaufs reagieren zu können. Um die passende Ersetzungsstrategie zu ermitteln, muss überlegt werden, wie das Anfragemuster aussieht. In der Regel werden die Anfragen von mobilen Objekten, welche sich bewegen, an die Knoten gestellt. Geht man vom menschlichen Verhaltensmuster aus, dann sucht ein Mensch meist wenige Arten von Objekten in seiner

Umgebung. Wenn man nun davon ausgeht, dass das Anfragemuster sich meist auf immer die selben gesuchten Objekte bezieht, müssen somit die zuletzt abgefragten Objekte von der Ersetzung verschont bleiben. Somit scheidet an diesem Punkt die FIFO-Ersetzungsstrategie aus. Betrachtet man nun von mehreren Menschen das Anfragemuster, so wird auch deutlich, dass dies von der Tageszeit oder dem Wochentag abhängt. An einem Wochenende steigt der Bedarf an naheliegenden Diskotheken, im gleichen Zug sinkt die Anzahl der Anfragen an behördliche Gebäude. Daher wird auf die LRU-Ersetzungsstrategie zurückgegriffen, da diese sicherstellt, das nur aktuell „im Trend liegende“ Objekte im Cache gespeichert werden. Da sich Knoten jedoch den Datenraum teilen und sich bewegende Objekte den vom Knoten abgebildeten Datenraum verlassen können, ist auch eine globale Ersetzungsstrategie notwendig. Diese globale Ersetzungsstrategie sieht vor, dass ein Knoten seine Nachbarknoten darüber informiert, dass ein bestimmtes Anfragemuster häufig gestellt wird und die Möglichkeit besteht, dass sich dieses Anfragemuster auch in den Nachbarknoten vermehrt zunimmt. Daher werden Objekte, welche diesen mitgeteilten Anfragen entsprechen, von der lokalen Ersetzungsstrategie ausgenommen. Diese Ausnahmen werden jedoch mit einem vordefinierten Lebenszeit versehen, um nicht den lokalen Cache zu blockieren.

4.3. Aufbau des Netzes

Das Peer-to-Peer-Netzwerk wird nach dem CAN-Schema aufgebaut. Da im CAN-Netzwerk eine Splittung des Datenraumes stattfindet, wenn sich ein neuer Knoten anmeldet, ist es im Vorfeld notwendig, eine Strategie zu entwickeln, welcher Knoten für die Splittung ausgewählt wird. Folgende Szenarien wurden betrachtet:

- *Größe des zugewiesenen Datenraumes:* Der erste Ansatz besteht darin, den Knoten mit dem flächenmäßig größten Datenraum zu splitten. Da sich das Content Adressable Network ein kartesisches Koordinatensystem zu Eigen macht, kann eine Funktion aufgestellt werden, um zu dem gewählten Einheiten die Größe des Datenraums zu berechnen. In konkreten Fall können mit den gegebenen GPS-Koordinaten die darin liegende Fläche berechnet werden. Der Vorteil an dieser Methode liegt darin, das unabhängig von den abgerufenen Daten eine Splitreihenfolge im Vorfeld erstellt werden kann. Es muss daher noch nicht mal eine Berechnung über die Größe gemacht werden, da die Anzahl der vorherigen Splitts gezählt werden kann. Die Knoten verteilen sich somit gleichmäßig über den Datenraum. Nachteilig ist hingegen, dass der Datenraum unterschiedlich stark abgefragt werden kann, somit werden viele Knoten häufiger mit Anfragen belastet als andere. Es findet somit ein Lastenungleichgewicht zwischen den Knoten statt.
- *Anfragelast:* Ein weiterer Ansatz liegt darin, Knoten, die verhältnismäßig oft Anfragen erhalten, zu entlasten. So wird sichergestellt, dass in Bereichen des Datenraumes, in der viele Anfragen getätigt werden, eine höhere Dichte an Knoten vorhanden sind. Die Größe des zugewiesenen Datenraumes eines Knotens wird nicht berücksichtigt.

Betrachtet man nun aber die Situation, dass jeder Knoten für seinen Datenraum einen Cache führt, werden häufig gestellte Anfragen auch häufig aus dem Cache beantwortet. Wenn der Cache seine Ressourcen noch nicht aufgebraucht hat, kann dieser eine noch höhere Last als die bis dahin schon bekannte aufnehmen und bearbeiten. Auf der anderen Seite können Knoten im Netzwerk vorhanden sein, die trotz einer geringen Anfragelast bereits am oberen Ende ihrer Leistungsfähigkeit liegen, welche nicht entlastet werden.

- *Cachefüllstatus:* Der dritte und hier letzte vorgestellte Ansatz ist, die Füllstaten in den Caches zu überprüfen und dabei die Knoten zu entlasten, welche einen hohen Füllstatus im Cache besitzen. Da ein Cache einen endlichen, meist kleinen Speicher besitzt, können bereits bei wenigen Anfragen der Cache zum Überlauf gezwungen werden, was zur Folge hat, dass Objekte im Cache gelöscht werden müssen. Der Vorteil ist daher, dass ein klares Verhältnis zwischen der Leistungsfähigkeit des Knotens und der Anfragelast geschaffen wird, welches zur Ermittlung des zu splittenden Knotens dient. Der Nachteil liegt jedoch am Verwaltungsaufwand, da bei jedem neu eingewählten Knoten die Füllstaten der Knoten abgefragt werden muss, was meist nur mit Flooding erfolgen kann.

Jede der drei vorgestellten Varianten hat entscheidende Vorteile und Nachteile. Während die Splittung nach der Größe des zugewiesenen Datenraums eines Knotens den kleinsten Verwaltungsaufwand darstellt, da die Splitreihenfolge bereits im Vorfeld fest vorgegeben ist. Bei Ermittlung der Füllstaten der Caches ist der Verwaltungsaufwand am höchsten, da diese mittels Flooding abgefragt werden müssen. Jedoch kann im dritten Lösungsansatz die Knoten mit ihren unterschiedlichen Ressourcen am besten entlastet werden. Dies ist dann auch der Grund, weshalb die Splittung nach dem Cachefüllstatus erfolgt.
Der Knoten, welcher für den Splittvorgang ermittelt worden ist, wird zum Splitten seines zugewiesenen Datenraums aufgefordert. Der Datenraum wird anhand einer neuen Achse mittig gesplittet. Die eine Hälfte mit den dazugehörigen Einträgen verbleibt bei dem gerade gesplitteten Knoten, die andere Hälfte wird dem neu angemeldeten Knoten zugewiesen. Dem neuen Knoten werden zudem seine Nachbarn mitgeteilt. Da ein Knoten stets Informationen über den Datenraum seiner Nachbarn hat, können diese dem neuen Knoten mitgeteilt werden. Es gilt dabei, dass der neue Knoten maximal so viele Nachbarn hat, wie der gesplittete Knoten vor der Splittung. Zum Schluss werden den Nachbarn noch die neue Situation mitgeteilt.
Das Content Adressable Network ist als ereignisbasiertes Netzwerk anzusehen, daher sind Nachrichten zwischen den Knoten notwendig. Für die Anmeldung und den darauffolgenden Splittvorgang sind somit folgende Nachrichten notwendig:

- *JOIN:* Der neue Knoten stellt eine JOIN-Anfrage an einen Knoten im Netzwerk. Der Knoten, welcher die JOIN-Message entgegennimmt muss nun den Knoten ermitteln, welcher für den Splittvorgang in Frage kommt. Diesem ermittelten Knoten wird die JOIN-Anfrage weitergeleitet. Dieser Knoten wird somit zum Splitten aufgefordert.

- *WELCOME:* Der gesplittete Knoten stellt die Informationen für den neuen Knoten zusammen, welche in der WELCOME-Message verschickt werden. In dieser WELCOME-Message ist beispielsweise der zugewiesene Datenraum enthalten und die Nachbarschaftsverhältnisse im Netzwerk.
- *UPDATE:* Die Knoten, welche zu dem gesplitteten Knoten ein direktes Nachbarschaftsverhältnis besitzen, müssen über die Veränderungen informiert werden. Darin wird der neue, verkleinerte Datenraum bekannt gegeben. Der neuangemeldete Knoten muss bei den Nachbarknoten registriert werden.

4.4. Anfragen über Queries

Das Peer-to-Peer-Netzwerk stellt mit seinen lokalen Caches einen verteilten Gesamtcache dar. Wird eine Anfrage (Query) an das Netzwerk gestellt, muss ermittelt werden, welcher Knoten für die Bearbeitung dieser Query zuständig ist. Der Knoten, welcher die Query als erster im Netzwerk entgegennimmt, wird im Folgenden als „Anfrageknoten“ bezeichnet. Da das Netzwerk nach dem CAN-Schema aufgebaut wird und geografische Koordindaten zur Partitionierung des Netzwerkes verwendet werden, sind in der Query die geografischen Zielkoordinaten zu ermitteln. Dies kann zum Beispiel ein Punkt im geografischen Gebiet sein, für welchen Informationen benötigt werden. Somit kann ein genauer Punkt ermittelt werden, zu welchen dann die Anfrage weitergeleitet werden muss. Da ein Knoten jedoch nur die zugewiesenen Bereiche seiner Nachbarn kennt, ist die Anfrage zu dem Knoten weiterzuleiten, der dem Zielpunkt am nächsten liegt. Abbildung 4.2 zeigt ein Beispiel für die Weiterleitung vom Anfrageknoten zum Zielpunkt.

Der Knoten, welcher den Zielpunkt in seinem zugewiesenen Datenraum beinhaltet, wird nachfolgend als „Initialknoten“ bezeichnet. Dieser Knoten kann nun mit der Bearbeitung der Query beginnen. Beinhaltet die Query einen geografischen Bereich, für welchen Informationen abgefragt werden, kann der Fall eintreten, dass dieser über die Grenzen des zugewiesenen Datenraums des Initialknotens hinausgeht. Somit ist der Nachbarknoten ebenfalls in die Berechnung mit einzubeziehen. Der Nachbarknoten, welcher zusätzlich zur Berechnung des Query einbezogen worden ist, schickt nach Beantwortung der Query seine Lösungsmenge an den Initialknoten zurück, welcher diese mit seiner Lösungmenge kombiniert und doppelte Objekte oder Informationen aus der Lösungsmenge entfernt. Doppelte Objekte entstehen dadurch, dass es Objekte gibt, welche nicht als Punkt, sondern als Polygon realisiert sind und daher eine Fläche einnehmen. Diese Fläche kann auf einer Grenze zwischen den Datenräumen zweier Knoten liegen. Objekte, welche mit ihrer Grundfläche den Datenraum eines Knotens schneidet, wird im Cache gespeichert. Da ein solches Randobjekt nun in allen angrenzenden Knoten gespeichert werden kann, können doppelte Objekte in der Lösunge entstehen, da alle Knoten dieses Objekt in ihre eigene Lösungsmenge aufnehmen, welche zu einem späteren Zeitpunkt erst zusammengetragen werden. Die Lösungsmenge wird vom Initialknoten an den Anfrageknoten zurückgesendet, welcher die Lösungsmenge

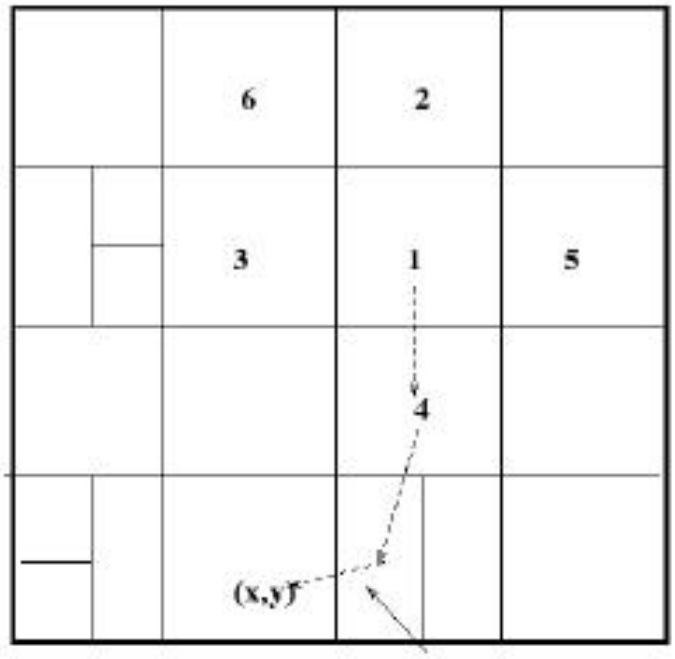


Abbildung 4.2.: Weiterleitung einer Query

dann an den Benutzer, der die Anfrage gestellt hat, weiterleitet. Für die Bearbeitung einer Query sind ebenfalls Nachrichten notwendig:

- *QUERY:* Die QUERY-Nachricht enthält die Anfrage vom Benutzer.
- *ROUTE:* Die ROUTE-Nachricht dient zur Weiterleitung der Query zum Initialknoten. Eine ROUTE-Nachricht enthält die Zielkoordinaten der Query, sowie die die unveränderte QUERY-Nachricht, welche an den Anfrageknoten gestellt worden ist.
- *REPLY:* Nachdem eine Lösungsmenge für die Query zusammengestellt worden ist, wird mit Hilfe einer REPLY-Nachricht diese an den Initialknoten, bzw. den Anfrageknoten zurückgeschickt. Falls die Bearbeitung einer Query über mehr als einen Knoten abgewickelt wird, müssen aus mehreren REPLY-Nachricht wieder eine REPLY-Nachricht zusammengestellt werden, wobei doppelte Informationen oder Objekte entfernt werden müssen.

Werden für die Beantwortung einer Query die Nachbarknoten mit einbezogen, kann der Fall eintreten, das ein Knoten von mehreren Nachbarn die gleiche Anfrage erhält. Abbildung 4.3 zeigt ein Beispiel, wann dieser Fall eintreten kann.

Der Knoten A ist der Initialknoten und beginnt die Berechnung. Es wird festgestellt, das sowohl Knoten B, als auch Knoten D im Anfragebereich der Query liegen, die Query wird somit an beide Knoten weitergeleitet. Diese berechnen nun unabhängig voneinander ihre

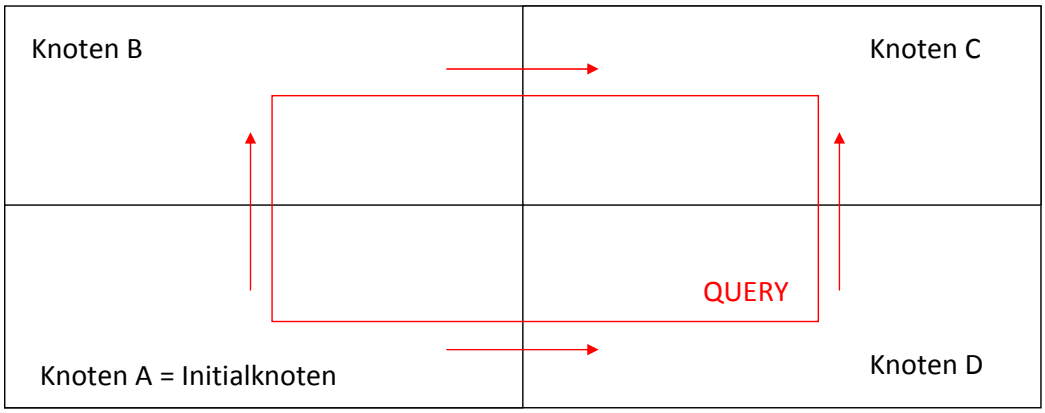


Abbildung 4.3.: Query über mehrere Knoten

Lösungsmenge, stellen aber beide fest, dass auch Knoten C für die Berechnung der Query benötigt wird. Somit schicken beide Knoten die Query an Knoten C weiter. Um nun eine Doppelberechnung im Knoten C zu vermeiden, werden Prozess-IDs eingeführt, um eine Query eindeutig zu bezeichnen. Der Initialknoten erzeugt eine Prozess-ID, welche fest in die QUERY-Nachricht eingebettet wird. Erhält Knoten C im konkreten Fall die QUERY-Nachricht, so wird überprüft, ob die beigefügte Prozess-ID bereits einmal berechnet worden ist. Wenn dies nicht der Fall ist, wird die Anfrage bearbeitet und die Lösungsmenge an den Knoten geschickt, von welchem die QUERY-Nachricht aus weitergeleitet worden ist. Wird die Prozess-ID jedoch als bereits berechnet erkannt, so wird nur eine leere Lösungsmenge an den weiterleitenden Knoten gesendet. Jeder Knoten führt somit eine Liste mit Prozess-IDs, welche bereits berechnet worden sind. Um die Liste mit den Prozess-IDs nicht unnötigerweise anwachsen zu lassen, wird vom Initialknoten, wenn alle Lösungsmengen vorliegen und zusammengefasst worden sind, eine PROCESSFINISH-Nachricht verschickt, um den Knoten zu signalisieren, dass die Prozess-ID nicht länger in der Liste gehalten werden muss.

4.5. Abwahl eines Knotens, Fehlerfall

Da der Datenraum vollständig auf die Knoten im Netzwerk aufgeteilt worden ist, bedeutet ein fehlerhafter oder abgemeldeter Knoten, dass die dazugehörige Datenraumpartition ebenfalls nicht mehr erreichbar ist. Um dem entgegenzuwirken, werden zwischen Knoten, welche ein direktes Nachbarschaftsverhältnis zueinander haben, in regelmäßigen Abständen

HEARTBEAT-Nachrichten verschickt, um die Funktionsfähigkeit des Nachbarknotens zu überprüfen. Wird auf die HEARTBEAT-Message keine Antwort empfangen, so ist von einem Fehlerfall des Nachbarknotens auszugehen. Da der Datenraum des Nachbarknotens bekannt ist, kann die dadurch entstandene Lücke im Datenraum geschlossen werden. Im konkreten Fall können aber die Einträge, welche im Cache vom fehlerhaften oder abgemeldeten Knoten sind, nicht wieder hergestellt werden.
Möchte sich ein Knoten korrekt aus dem Netzwerk abwählen, so teilt er dies einem seiner Nachbarn mit. Neben der Mitteilung werden zudem die Einträge im Cache mitgeliefert, damit der Knoten, welcher diesen Datenbereich übernehmen muss, den abgemeldeten Knoten komplett ersetzen kann. Ansonsten unterscheidet sich das korrekte Abwahlverfahren vom Fehlerfall nicht.
Wird von einem Knoten registriert, dass sich ein Knoten aus dem Netzwerk abmelden möchte, so ist eine Reorganisation des Content Adressable Network notwendig. Wie bereits vorgestellt worden ist, kann das CAN in einem Binärbaum dargestellt werden, wobei die Knoten des Netzwerkes sich immer an den Blättern des Baumes befinden. Ein Knoten im Binärbaum entsteht genau dann, wenn ein Splittvorgang vorgelegen hat. Hat ein Knoten im Baum zwei Blätter als Söhne, so sind die Knoten, welche diese Blätter darstellen, das Ergebnis des Splittvorganges. Abbildung 4.4 b zeigt, wie sich ein Splittvorgang im CAN-Baum auswirkt. Wählt sich nun ein Knoten aus dem Netzwerk ab, so befindet sich dieser Knoten in einem Blatt im dazugehörigen CAN-Baum. Da dieser CAN-Baum sogar vollständig ist, existiert immer ein zweiter Pfad vom Vaterknoten aus. Ist der zweite Pfad vom Vaterknoten aus ebenfalls ein Blatt, so übernimmt dieser gefundene Knoten den Datenraum des abgemeldeten Knoten mit, die vorhergegangene Splittung wird somit rückgängig gemacht. Das Blatt nimmt im CAN-Baum somit die Position des Vaterknotens ein und beide ausgehende Pfade werden gelöscht. Somit wird der Vaterknoten wieder zum Blatt. Führt aber der zweite Pfad vom Vaterknoten aus zu einem weiteren Knoten, so muss in diesem Zweig ein Knoten gefunden werden, welcher zwei Blätter als Kinder hat.

LEMMA: Hat im CAN-Baum ein Knoten zwei weitere Knoten als Kinder, so existiert in jedem Zweig ein Knoten, welcher zwei Blätter als Kinder hat.
Beweis: Gegeben ist ein einelementiger Baum T. T repräsentiert den Zustand im Content Adressable Network, in dem ein Knoten den kompletten Datenraum auf sich abbildet. Wird dieser Knoten gesplittet, so erhält T immer genau zwei Kinder, $t1$, und $t2$. Die Wurzel von T ist somit ein Knoten im Baum, welcher die Splittung repräsentiert, die Kinder von T repräsentieren die zwei Knoten im Netzwerk. Die Kinder $t1$ und $t2$ von T können auch als einelementige Teilbäume $T1$ und $T2$ von T aufgefasst werden. Somit führt jede Splittung im Content Adressable Network dazu, dass ein einelementiger Baum zu einem vollständigen Binärbaum mit einem Knoten und zwei Kindern wächst.

Da sichergestellt ist, dass ein Teilbaum im CAN-Baum immer einen Knoten beinhaltet, welcher zwei Kinder hat, kann mit folgendem Algorithmus dieser Knoten gesucht werden. Befinden sich mehrere Knoten mit zwei Blättern als Kinder im Teilbaum, so ist es unwichtig, welcher Knoten gefunden wird.

Algorithmus 4.1 Zwei-Kinder-Knoten-Suche

```
procedure FindNode(n)
    if GetFirstChild(n) = NODE then
        return FindNode(GetFirstChild(n))
    else if GetSecondChild(n) = NODE then
        return FindNode(GetSecondChild(n))
    else
        return n
    end if
end procedure
```

Mit dem Algorithmus 4.1 wird somit ein Knoten gefunden, welcher zwei Kinder besitzt. Ein Kind übernimmt den Datenraum des Knotens im CAN, welcher sich aus dem Netzwerk abmeldet. Der Knoten im CAN-Baum, welcher ermittelt worden ist, repräsentiert eine Splittung der beiden Kinder. Da nun ein Kind die Position im Baum wechselt, wird die Splittung zwischen diesen beiden Kindern rückgängig gemacht. Der Kindsknoten, welcher noch übrig bleibt, wird um eine Ebene im Baum nach oben versetzt und übernimmt die Position des gefundenen Knotens: der Knoten wird zum Blatt. Abbildung 4.4 c) und d) zeigen schematisch diesen Vorgang.

Nachdem die Reorganisation des Netzwerkes abgeschlossen ist, müssen noch die Nachbarschaftsverhältnisse angepasst werden. Der Knoten, welcher den Platz des abgemeldeten Knotens übernimmt, kann auch die Liste der Nachbarknoten unverändert übernehmen, es ist lediglich notwendig, dass sich dieser Knoten seinen Nachbarn vorstellt, um eine Änderung des entsprechenden Routingtabelleneintrags des Nachbarn zu erwirken. Für das Abmelden und das Registrieren von fehlerhaften Knoten sind folgende Nachrichten notwendig:

- *HEARTBEAT:* Eine HEARTBEAT-Nachricht wird regelmäßig zwischen Knoten, welche ein Nachbarschaftsverhältnis haben, verschickt. Wird eine HEARTBEAT-Nachricht empfangen, wird mit einer RESPONSE-Nachricht geantwortet.
- *RESPONSE:* Antwort auf eine HEARTBEAT-Nachricht.
- *CHECKOUT:* Möchte sich ein Knoten abmelden, wird eine CHECKOUT-Nachricht an einen seiner Nachbarn geschickt. Darin beinhaltet sind auch die Informationen, welche für den Knoten, der diese Position einnimmt, wichtig sind, darunter zum Beispiel die Elemente im Cache.
- *REORGANIZE:* eine REORGANIZE-Nachricht wird von dem Knoten erstellt, welcher die CHECKOUT-Nachricht erhalten hat, bzw. den fehlerhaften Knoten bemerkt hat.

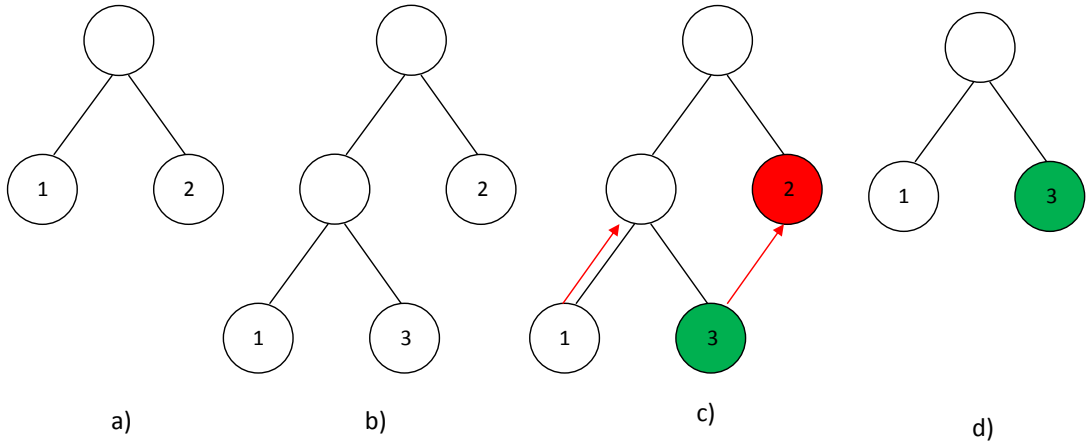


Abbildung 4.4.: An- und Abmelden eines Knotens im CAN-Baum

Die REORGANIZE-Nachricht wird mit Hilfe des CAN-Baumes an einen Knoten mit zwei Kindern weitergeleitet. Nachdem dieser Knoten gefunden worden ist, leitet dieser die Reorganisation des Netzwerkes ein. In einer REORGANIZE-Nachricht befinden sich die gleichen Informationen wie in einer CHECKOUT-Nachricht.

Das Konzept erweitert die bestehende Nexusarchitektur um einen *Cache-Overlay*, weiterführend als Cacheverwaltung bezeichnet, welche in der Föderationsschicht eingebettet ist. Diese Cacheverwaltung verwaltet die darunter liegenden Cacheinstanzen, welche von Nexusknoten geführt werden. Abbildung 4.5 zeigt schematisch die erweiterte Architektur, welche nachfolgend realisiert wird.

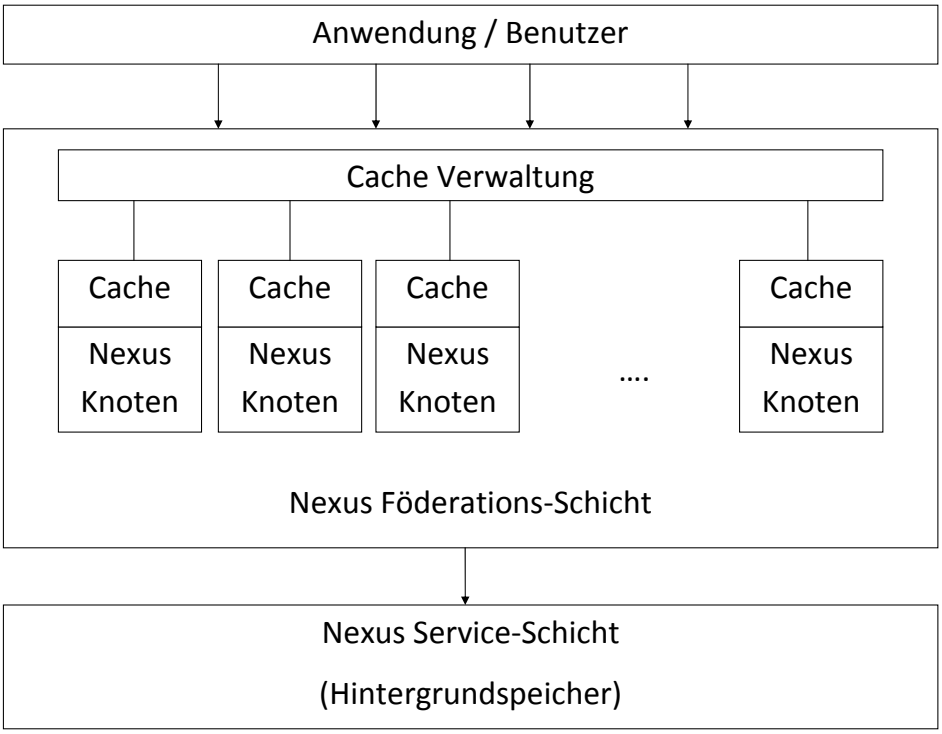


Abbildung 4.5.: Konzeptarchitektur

5. Realisierung

> Das Ziel des Künstlers ist die Erschaffung des Schönen. Was das Schöne ist, ist eine andere Frage.
>
> *(James Joyce - irischer Schriftsteller)*

Nachfolgend wird die Realisierung erläutert, wobei besonders die CAN-Realisierung im Detail erläutert wird. Zudem werden alle Nachrichten beschrieben, welche benötigt werden, um den semantischen, kooperativen Cache zu betreiben. Desweiteren wird das Simulationstool Peersim vorgestellt, mit Hilfe dessen das zugrundeliegende Peer-to-Peer-Netzwerk erzeugt wird.

5.1. Grundannahmen

Für die Realisierung in Java wurden für die Implementierung einige Voraussetzungen erarbeitet, um die Testfälle möglichst einfach zu realisieren. Da sich die Realisierung auf ein konkretes Konzept aufbaut, werden auch Begriffe verwendet, welche sich auf den konkreten Fall beziehen. Wird also von einem *Gebiet* gesprochen, so ist ein Ausschnitt aus einer Landkarte gemeint und kein abstraktes, mathematisches, geometrisches Objekt.
Bei der Realisierung wurde auf die Implementierung vom NearestNeighborsOperator verzichtet, da die Berechnung im Cache wesentlich aufwendiger ist als einer gebietsbeschränkten Anfrage. Dies rührt daher, dass eine Lösungsmenge in einer gebietsbeschränkten Anfrage nur sinngemäß in dem gesuchten Gebiet liegen kann. Bei einem NearestNeighborsOperator wird von einem bestimmten Punkt aus eine Menge von Objekten verlangt, unabhängig davon, wie weit diese von dem Anfragepunkt aus entfernt sind. Somit ist es extrem schwer, die Lösungsmenge, welche alleinig aus dem Cache geladen wird, als vollständig einzustufen. Im Allgemeinen werden stets gebietsbeschränkte Anfragen gestellt, es ist somit immer ein Gebiet angegeben, in welchem die Objekte gesucht werden sollen. Da sich das Content Adressable Network auf eine geografische Partitionierung des Datenraumes stütz, sind alle Anfragen, welche keine geografischen Daten enthalten, direkt an den Hintergrundspeicher weiterzuleiten. Ein weiterer Punkt bei der Realisierung ist, dass die zurückgelieferten AWML-Objekte vollständig sind. Wie bei der Vorstellung der AWQL-Anfragen können Anfragen so gestellt werden, dass nur bestimmte Attribute zurückgeliefert werden. Fehlende Attribute können den Cache dazu zwingen, trotz eventuell bereits vorhandener AWML-Objekte

den Hintergrundspeicher in die Berechnung mit einzubeziehen. Dies ist beispielsweise der Fall, wenn eine Anfrage über italienische Restaurants nur die Attribute Typ und Adresse zurückgeliefert haben möchte. Wird bei der nächsten Anfrage im selben Gebiet nun nach der Telefonnummer der italienischen Restaurants gefragt, muss der Hintergrundspeicher wieder einbezogen werden. Bei vollständigen AWML-Objekten wird davon ausgegangen, dass trotz einer Anfrage über die Adressen auch alle anderen Daten über das gesuchte Objekt zurückgeliefert und im Cache gespeichert werden.
Ein weiterer Punkt ist, dass das Netzwerk fehlerfrei ist und sich auch keine Knoten aus dem Netz abwählen. Somit müssen zwischen den Knoten in der Realisierung keine HEARTBEAT-Nachrichten verschickt werden.

5.2. Peersim

Peersim ist eine Peer-to-Peer-Simulation, welche frei im Internet zugänglich ist. Da Peer-to-Peer-Systeme eine sehr große Größe annehmen können und Knoten ständig sich ein- und abwählen, bedarf es eines mächtigen Simulationstool, um neue Protokolle zu verifizieren.

5.2.1. Konfiguration

Peersim wird zentral von einer Konfigurationsdatei gesteuert, in welcher man beispielsweise die verwendeten Protokolle oder den verwendeten Knoten angibt. In dieser Datei wird auch die Netzgröße definiert, wieviele Knoten sich im Peer-to-Peer-Netzwerk befinden.

5.2.2. Knoten

Peersim bietet die Möglichkeit, eigene Knoten zu verwenden. Hierfür wird das Interface *node* zur Verfügung gestellt. Das Interface gibt bestimmte Methoden als zwingend notwendig vor, beispielsweise die Vergabe einer ID oder die Anzahl der laufenden Protokolle. Oft wird jedoch ein *GeneralNode* verwendet, welcher bereits im Peersim integriert ist. Dieser wird automatisch verwendet, wenn kein eigener Knoten in der Konfigurationsdatei angegeben wird. Der *GeneralNode* wird lediglich um einen Fehlerstatus erweitert, um fehlerhafte Knoten im Netzwerk zu simulieren. Ansonsten werden nur die bereits vordefinierten Methoden des Interfaces *Node* implementiert.

5.2.3. Kanten

Der Begriff *Kanten* ist in Peersim weniger geläufig. Der Begriff *Verbindungen (Linkable)* ist hingegen passender. Nachbarschaftsverhältnisse zwischen Knoten müssen selbst festgelegt werden, hierfür bietet Peersim das Interface *Linkable* an. Die bereitgestellten Methoden soll das Verwalten der Nachbarn ermöglichen, zum Beispiel das Hinzufügen oder Entfernen eines Nachbarn oder die Anzahl der Nachbarn. Wie die Nachbarn gespeichert werden, wird jedoch nicht definiert. Sie müssen allerdings durchnummeriert werden.

5.2.4. Netzwerk

Das fertige Klasse *Network* verwaltet die angemeldeten Knoten. Dies wird von Peersim übernommen, daher ist diese Klasse nicht als Interface vorhanden. Diese Klasse verwendet ein Array, um die Knoten zu speichern und entsprechend ihrer Durchnummerierung anzusprechen. Jedoch werden keine Nachbarschaftsverhältnisse geschaffen. Die Klasse *Network* simuliert an sich nur ein Wald von einelementigen Bäumen. Die Nachbarschaftsverhältnisse werden erst mit dem Interface *Linkable* festgelegt.

5.2.5. Protokoll

Auf einem Knoten können mehrere Protokolle gleichzeitig laufen. Es gibt zwei verschiedene Protokollschnittstellen, welche zu Verfügung gestellt werden. Das erste Protokoll ist das in Peersim benannte *CDProtocol* , welches immer zyklisch angesprochen wird. Ein Zyklus wird ebenfalls in der Konfigurationsdatei definiert. In einem Zyklus wird jedes Protokoll, welches die CDProtokoll-Schnittstele verwendet, angesprochen. Die einzige Methode, die neben dem Konstruktor im Interface angegeben ist, ist die Methode *nextCycle* , welche mit jedem Zyklendurchlauf verwendet wird.
Auf der anderen Seite stellt Peersim das Protokoll-Interface *EDProtocol*zur Verfügung, welches eventbasierend ist. Wird ein Event an einem Knoten ausgelöst, so wird jedes Protokoll, das das *EDProtocol* verwendet, angesprochen. Hierfür muss die Methode *processEvent* implementiert werden.

5.2.6. Kontrollinstanzen

Eine Kontrollinstanz wird in Peersim nur einmal verwendet und steuert das Peer-to-Peer-Netzwerk. Eine Kontrollinstanz wird durch das Interface „Control“ definiert. Das Interface *Control* definiert lediglich die Methode *execute*, welche zyklisch aufgerufen wird. Die Methode *execute* gibt einen Booleanwert zurück, der festlegt, ob ein weiterer Durchlauf notwendig ist.

Wenn ein *true*zurückgeliefert wird, gilt die Simulation als beendet. Da eine Kontrollinstanz nur einmal initialisiert wird, können globale Variablen verwendet werden, um intern die Anzahl der Schleifendurchläufe festzulegen. Dies ist sinnvoll, wenn das Netzwerk nur langsam aufgebaut werden soll und mit jedem Schleifendurchlauf ein neuer Knoten sich einwählen soll.

5.2.7. Weitere Klassen

Peersim bietet noch eine große Anzahl weiterer Klassen und Interfaces an, beispielsweise Fehlerprotokolle, Beobachter oder Graphen. Da diese Klassen jedoch nicht für diese Diplomarbeit von Bedeutung sind, wird auf sie nicht weiter eingegangen. Das Interface *Transport*, welches das Protokoll-Interface erweitert, bedarf noch einiger Worte. Dieses Interface stellt die Methode der Kommunikation zwischen den Knoten dar. Jedoch ist dies nur bei Protokollen, welche auf dem *EDProtocol* basieren, möglich. Nachrichten werden als Events implementiert. Das *Transport-Interface* simuliert zudem zufällige Latenzzeiten zwischen den Knoten.

5.3. Allgemeines

Die Realisierung gliedert sich in zwei große Bereiche: Den Cache und den CAN. Jedoch gibt es Klassen, welche für beide Implementierungsbereiche notwendig sind.

5.3.1. Geometrie

Gebiete, in denen nach Objekten gesucht werden soll, werden als Polygone dargestellt. Ein partitionierter Datenraum im Zweidimensionalen wird als Rechteck dargestellt. Um herauszufinden, ob ein Knoten für die Berechnung einer Anfrage mit einem Gebiet einbezogen werden muss, wird das Suchgebiet und der zugewiesene Datenraum geschnitten. Wird eine Schnittmenge festgestellt, so ist der Knoten in die Berechnung mit einzubeziehen. Geometrisch werden somit die beiden Polygone geschnitten. Um eine Schnittmenge von Polygonen zu berechnen, gibt es in Java fertige Bibliotheken, wie beispielsweise das Projekt *JTS* [jts]. Die Nexusplattform verwendet ein darauf basierendes Paket, nämlich das Paket *CGeometry*. CGeometry erweitert das JTS-Paket mit einem *SpatialReferenzsystem*, um anzugeben, in welchem Koordinatensystem die JTS-Werte angegeben sind. Dies basiert auf der Tatsache, dass die Nexusplattform in der Lage ist, mit GPS-Koordinaten zu rechnen. Jedoch ist es rein mathematisch nicht möglich, mit GPS-Koordindaten qualitativ richtige Berechnungen durchzuführen. Die Erde hat die Form eines Ellipsoiden, der Abstand zweier GPS-Koordinaten verringert sich mathematisch immer mehr, je näher man in Erdpolnähe kommt, obwohl der Abstand der reinen GPS-Zahlen unverändert ist. Der Erdellipsoid wird unter anderem im

WGS84-Referenzsystem beschrieben. WGS steht hierbei für *World Geodetic System*, welches im Jahr 1984 aufgestellt worden ist. Das WGS84 stellt als geodätisches Referenzsystem eines der in Nexus realisierten *Spatialferenzsysteme*dar. Um dennoch mit den Koordindaten im WGS84-Koordinatensystem rechnen zu können, bedarf es eines Wechsels des Referenzsystems. Benötigt wird ein Referenzsystem, das in der Lage ist, den Erdellipsoiden zu „plätten", um ein Koordinatensystem auf einer zweidimensionalen Fläche zu erzeugen, was einem kartesisches Koordinatensystem entspricht. Ein Beispiel für ein kartesisches Referenzsystem ist das *Gauß-Krüger-Koordinatensystem*[gau10], welches in der Lage ist, hinreichend kleine Gebiete der Erde mit metrischen Koordindaten konform zu verorten. Das Gitternetz der geografischen Daten werden in Drei-Grad-breite Medianstreifen aufgeteilt. Innerhalb eines Streifens können mit metrischen Einheiten Flächen- und Abstandsberechnungen vorgenommen werden, eine Berechnung über die Streifengrenzen hinweg gestaltet sich komplexer. Ein weiteres, kartesisches Koordinatensystem ist das *UTM-Koordinatensystem*[utm10]. Das UTM-Koordinatensystem ist dem Gauß-Krüger-Koordinatensystem sehr ähnlich, jedoch werden die geografischen Daten nicht in Medianstreifen aufgeteilt, sondern in kleine, rechteckige Gebiete. Abbildung 5.1 zeigt die räumliche Unterteilung der Erdoberfläche. Stuttgart liegt in der Zone 32N, während Berlin in der Zone 33N liegt. Für die Flächen- und Abstandsberechnung wird im folgenden vom WGS84-Referenzsystem ins UTM-Referenzsystem gewechselt.

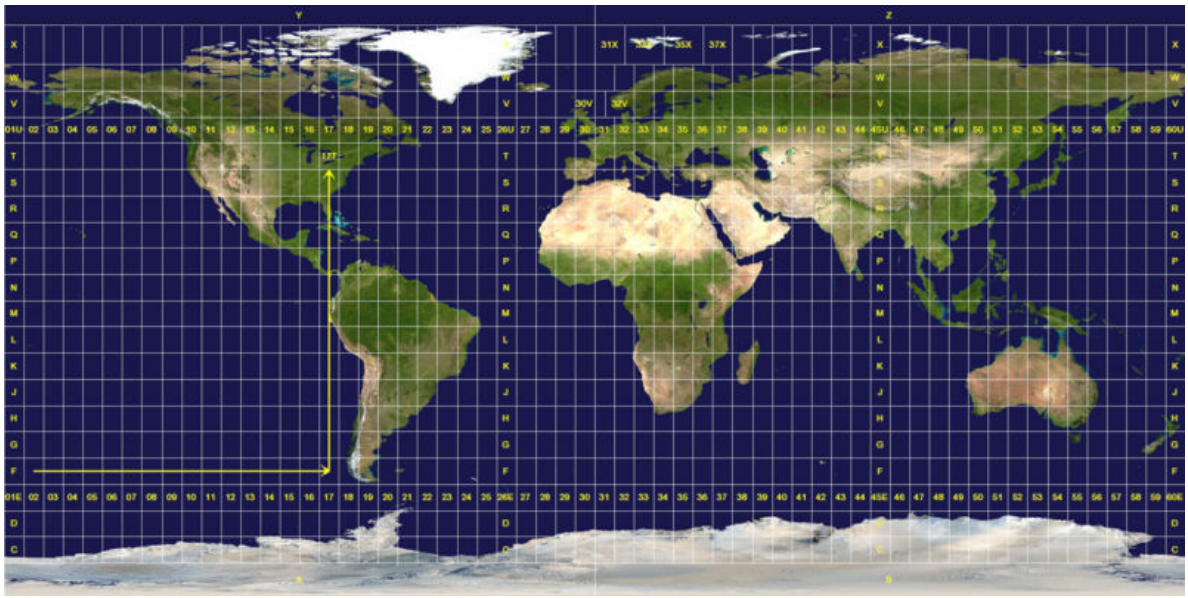


Abbildung 5.1.: UTM-Koordinatensystem (Quelle: [utm10])

5.3.2. MGeometry

Das CAN-Protokoll verwendet zum Beschränken seines zugewiesenen Datenraumes ein zweidimensionales Double-Array. Die erste Dimesion gibt an, ob es sich um horizontale oder vertikale Daten handelt, die zweite Dimension gibt den Maximal-, bzw. Minimalwert des zugewiesenen Datenraumes an. MGeometry stellt eine Schnittstelle zwischen den CGeoemetry-Klassen, den JTS-Geometrie-Klassen und dem zweidimensionalen Doublearray dar. Daher beschränken sich die Methoden meist nur auf das Konvertieren zwischen den einzelnen Klassen.

5.3.3. AWMLReader, AWQL-Reader

Eine weitere Schnittstelle zur Vereinfachung zu bestehenden Nexus-Paketen sind die AWML- und AWQL-Reader. Da die beiden Klassen wichtige Schlüsselklassen in der Nexusplattform sind und beide Klassen auf den ersten Blick sehr komplex erscheinen, wird der AWMLReader und der AWQLReader verwendet, um eine einfache und effiziente Schnittstelle zu den AWQL- und AWML-Klassen zu ermöglichen. Der AWML-Reader hat die Hauptaufgabe, von AWQL-Anfragen zurückgelieferte ResultSets zu speichern und doppelte Einträge zu lokalisieren und zu löschen.
In der Klasse „AWML-Reader“ sind nur zwei globale Variablen vorhanden, zum einen eine Variable vom Typ ResultSet, welche AWML-Objekte speichert, zum anderen eine HashSet-Liste, welche die ID's von den vorhandenen AWML-Objekten speichert. Da der Vergleich von zwei AWML-Objekten extrem komplex ist, werden von den vorhandenen AWML-Objekten die eindeutige ID entnommen und in einer separaten Liste gespeichert. Die Methode „expand“ stellt die wichtigste Methode im AWML-Reader dar. Der Programmcode 5.1 zeigt, wie das Einfügen in das ResultSet vonstatten geht.

Weitere Methoden im AWMLReader sind beispielsweise die Überprüfung auf „Leerheit“, sowie das Löschen aller Einträge.
Der AWQLReader gestaltet sich ein wenig komplexer, da dieser nicht nur in der Lage sein muss, AWQL-Queries auszulesen, sondern auch welche zu erzeugen. Wie im AWMLReader gibt es nur zwei globale Variablen, die erste ist vom Typ AwqlQuery, einer Nexusklasse, sowie eine Variable vom Typ „MAWQLGenerator“, welche eine vereinfachte Klasse des AWQLGenerators ist, in einem Nexuspaket integriert ist. Der AWQLGenerator hat die Aufgabe, die Klasse AwqlQuery zu erzeugen, bzw. deren Bestandteile.
Mit dem Konstruktor wird die AwqlQuery-Klasse mit übergeben, welche an den Cache gestellt werden soll. Der AWQLReader filtert die Geodaten in der AwqlQuery-Klasse heraus und überprüft die Schnittmenge mit dem zugewiesenen Datenraum. Wird eine Schnittmenge festgestellt, wird der RestrictionOperator der AwqlQuery-Klasse durch das eigene Gebiet erweitert. Der RestrictionOperator stellt eine boolsche Beschränkung dar, die auf Prädikaten basiert. Erfüllt ein Objekt diese boolschen Beschränkungen, so wird das Objekt in die

Listing 5.1 Expand-Methode im AWMLReader

```
public void expand(ResultSet rs) {
 // AWML-Objekt
 GenericObject gobj;
 // AWML-ID
 NOL nol;
 Iterator it = rs.iterator();
 while(it.hasNext()) {
  gobj = (GenericObject) it.next();
  nol = gobj.nolsIterator().next();
  if(!id.contains(nol)) {
   // Wenn die AWML-ID noch nicht in der ID-Liste vorhanden ist, wird die ID und das
       dazugehörige AWML-Objekt aufgenommen.
   id.add(nol);
   rs.add(gobj);
  }
 }
}
```

Lösungsmenge aufgenommen. Der Programmcode 5.2 zeigt, wie die AwqlQuery-Klasse durch ein weiteres Geoattribut erweitert wird.

Eine weitere, sehr wichtige Methode im AWQLReader ist die Erzeugung einer RemoveRestriction. Diese ist notwendig, falls Elemente aus dem Cache entfernt werden sollen. Dies ist beispielsweise der Fall, wenn der Cache gesplittet werden muss. Der Cache wird in diesem Fall zuerst kopiert und dann die Elemente des „abgeschnittenen“ Datenraumes entfernt. Die Erzeugung einer neuen AwqlQuery-Klasse ist fast identisch mit der Erweiterung einer AwqlQuery-Klasse. Zuerst wird ein AwqlTarget erzeugt, welches in einen SpatialCompOperator integriert wird, zum Schluss wird jedoch statt ein AND ein NOT-BooleanOperator erzeugt. Semantisch bedeutet dies also, dass alles gelöscht werden soll, was nicht im übergebenen Datenraum liegt.

5.4. CAN-Realisierung

Die CAN-Realisierung beruht auf Protokollen, welche innerhalb der Simulationsumgebung Peersim entwickelt wurden. Mit Hilfe von Peersim muss das zu grunde liegende Peer-to-Peer-Netzwerk nicht selbst implementiert werden.

5.4.1. CAN-Protokoll

Das CAN-Protokoll wird in der Klasse *CANProtocol* realisiert, die eine Unterklasse der Peersim-Schnittstelle *Protocol* ist. CANProtocol ist eine von zwei Klassen, welche Pakete

Listing 5.2 Intersects-Methode im AWQLReader

```
public AwqlQuery intersectQuery(Polygon polygon) {
 try {
  Collection<RestrictionOperator> ros = new ArrayList<RestrictionOperator>();
  // Erzeugung des Attributnamens
  QName qname = new QName(AWSConstants.NSAS_10_NS, "extent");
  GenericAttributePart gap =
      awqlGen.awmlFac.createAttributePart(MGeometryFactory.convert(polygon), qname,
      AWSConstants.SINGLE_VALUE_PART_NAME);
  AwqlTarget target = new AwqlTarget(awqlGen.aws, qname, true,
      AWSConstants.SINGLE_VALUE_PART_NAME);
  // Erzeugung des SpatialCompOperators, eine Spezialisierung des RestrictionOperators.
  SpatialCompOperator spo =
      awqlGen.awqlFac.createSpatialCompOperator(SpatialCompOperator.OPID_INTERSECTS, target,
      gap);
  // Speicherung des bisherigen RestrictionOperators in der AwqlQuery-Klasse
  ros.add(query.getRestriction());
  // Speicherung des neuen RestrictionOperators, welcher neu erzeugt wurde
  ros.add(spo);
  // Verknüpfung beider RestrictionOperatoren mit einem logischen AND
  BooleanOperator bo = awqlGen.awqlFac.createBooleanOperator(BooleanOperator.OPID_AND, ros);
  query.setRestriction(bo);
  return query;
 } catch(Exception e) {
  System.err.println("AWS-Error");
       return query;
 }
}
```

regelmäßig empfängt und verschickt. In dieser Klasse wird somit der Aufbau des Content Adressable Network verwaltet. Auf jedem Knoten wird ein CAN-Protokoll installiert, jedoch wird nur beim ersten initialisierten Knoten auch das CAN-Protokoll initialisiert. Auf jedem weiteren Knoten wird eine Kopie des CAN-Protokolls eingerichtet. Daher wird im Konstruktor des CAN-Protokolls der *SpaSe* gestartet. Die Klasse „SpaSe“ simuliert den Hintergrundspeicher, welcher in der Implementierung als *Server* bezeichnet wird. Desweiteren wird in dem Konstruktur der CAN-Baum initialisiert, und mit Hilfe der Peersim-Konfigurationsklasse „Configuration“ die Startwerte für die Anzahl der Dimensionen sowie die Koordinaten für den Datenraum.
Das CAN-Protokoll kann als deterministischer Automat angesehen werden, welcher verschiedene Zustände besitzt. Folgende Zustände kann das CAN-Protokoll annehmen:

- *ACTIVE:* Ist das CAN-Protokoll im Zustand ACTIVE, so sind keine laufenden Prozesse mehr vorhanden, es können neue Prozesse gestartet werden. Ein CAN-Protokoll erreicht das erste Mal den Zustand ACTIVE, wenn der zugewiesene Datenraum übernommen worden ist, sowie alle Nachbarn über den neuen Knoten informiert worden sind und die neuen Nachbarn die Änderung bestätigt haben.

- *AWAITING_WELCOME:* Dieser Zustand wird erreicht, wenn ein neuer Knoten eine JOIN-Anfrage an das Content Adressable Network gestellt hat. Dieser Knoten wartet dann auf die WELCOME-Nachricht, welche alle Informationen für die Aufnahme im Netzwerk besitzt. Während sich der Knoten in diesem Zustand befindet, werden keine weiteren Nachrichten bearbeitet.
- *AWAITING_ACCEPT:* Der Knoten, welcher gesplittet werden soll, erstellt die WELCOME-Nachricht für den neuen Knoten. Wird diese Nachricht verschickt, wird der gesplittete Knoten in den Zustand AWAITING_ACCEPT versetzt. Dies bedeutet, dass der Knoten auf die Bestätigung des neuen Knotens wartet. Werden in diesem Zustand Anfragen erhalten, so werden diese in einer Process-Queue gespeichert und dann abgearbeitet, wenn der Zustand ACTIVE wieder erreicht worden ist.
- *AWAITING_WELCOME_ACKUPDATES:* Ein Knoten wird in den Zustand AWAITING_WELCOME_ACKUPDATES versetzt, wenn noch auf die Bestätigungsnachricht der neuen Nachbarknoten gewartet werden muss. Sind alle Bestätigungsnachrichten eingegangen, wird der Knoten in den Zustand ACTIVE versetzt.
- *AWAITING_ACCEPT_ACKUPDATES:* Wenn ein Knoten Änderungen mittels einer UPDATE-Nachricht bekannt gibt, so wird dieser Knoten in den Zustand AWAITING_ACCEPT_ACKUPDATES versetzt. Dieser Knoten wartet somit auf die Bestätigungsnachrichten der Nachbarknoten, welche über eine Änderung informiert worden sind.
- *PROCESSING:* Der letztmögliche Zustand ist PROCESSING. Wird eine AwqlQuery aktiv bearbeitet, so wird der Knoten in diesen Zustand versetzt. Während der Bearbeitung einer Query kann der Knoten nicht gesplittet werden, Nachrichten können aber an die Nachbarn weiterverschickt werden. Auch kann während diesem Zustand eine weitere QUERY-Anfrage bearbeitet werden. Sind alle AwqlQuery-Berechnungen abgeschlossen, wird der Knoten wieder in den Zustand ACTIVE versetzt. Werden in diesem Zustand Nachrichten empfangen, welche nicht bearbeitet werden können, so werden diese in einer Warteschlange gespeichert und nach Rückstufung in den Zustand ACTIVE abgearbeitet.

Die Schlüsselmethode im CAN-Protokoll ist die von der Peersim-Schnittstelle vorgegebene „processEvent“-Methode. Diese Methode registriert eingehende Nachrichten, welche dann weiterverarbeitet werden. Für die Verwaltung und Bearbeitung des Content Adressable Network werden eine Reihe von Nachrichten benötigt, welche bereits im Konzept vorgestellt worden sind. Die nachfolgende Aufschlüsselung soll zeigen, wie das CAN-Protokoll auf den Erhalt der Nachrichten reagiert:

JOIN

Erhält das CAN-Protokoll eine JOIN-Nachricht über die Methode *processEvent*, so muss nun anhand der Füllstaten der Knoten ermittelt werden, welcher gesplittet werden soll. Leider ist

der Knoten mit dem absolut höchsten Füllstatus im gesamten Netzwerk nur über Flooding zu ermitteln. Um jedoch das Peer-to-Peer-Netzwerk nicht mit dem Flooding zu überfordern, wird ein lokales Flooding angewendet, um den Knoten mit dem höchsten Füllstatus „in der Nähe“ zu ermitteln. Hierfür holt der Knoten, welcher die JOIN-Nachricht erhalten hat von seinen Nachbarknoten deren Füllstaten ein. Hat ein Nachbarknoten einen höheren Füllstatus als der suchende Knoten selbst, so wird die JOIN-Anfrage an diesen Knoten weitergeschickt. Der Knoten, an welchen diese JOIN-Nachricht weitergeschickt worden ist, überprüft wiederum die Füllstaten seiner Nachbarknoten. Wird ein Knoten erreicht, welcher einen höheren Füllstatus besitzt als all seine Nachbarknoten, so wurde der zu splittende Knoten erreicht.
Ist der zu splittende Knoten im Zustand PROCESSING, so wird die JOIN-Nachricht vorerst in eine Processqueue hinterlegt und später ausgeführt, falls der Zustand ACTIVE wieder erreicht wird. Der Splittvorgang wird vorbereitet, jedoch nicht abgeschlossen. Zuerst wird der zugewiesene Datenraum geteilt, wobei die Variable *split* die Dimension beinhaltet, welche gesplittet wird. Der neue Wert für die Variable *split* wird in einer Zwischenvariable gespeichert, genau wie die zwei neuen Teilräume des vorherigen Datenraumes. Mit Hilfe der Methode *getNeighbors* werden die neuen Nachbarschaftslisten erstellt, welche an die beiden neuen Teilräume angrenzen. Auch die neuen Nachbarschaftslisten werden in einer Zwischenvariable gespeichert. Im letzten Schritt wird noch der vorhandene Cache kopiert und die Einträge des Caches, welche dem neuen Knoten zugewiesen werden sollen in sofern aufbereitet, als alle Elemente, welche außerhalb des Datenraumes des neuen Knoten liegen, gelöscht werden. Die Elemente im alten Knoten bleiben vorerst unverändert. Der Programmcode 5.3 zeigt schemenhaft, wie die notwendigen Informationen aufbereitet werden. Die Variable „intervals“ ist eine globale Variable im zu splittenden Knoten und stellt die Grenzen des bisherigen Datenraumes dar.

Listing 5.3 Splittvorgang

```
if(event instanceof Join) {
  // Die bisherigen Grenzen des zugewiesenen Datenraums werden kopiert.
  othersPendingIntervals = intervals;
  myPendingIntervals = intervals;
  // Splittung des Datenraumes, othersPendingIntervals stellt dabei den Datenraum des neuen
      Knotens dar, myPendingIntervals die Zwischenspeicherung des eigenen, neuen Datenraums.
  othersPendingIntervals[split][0] = (intervals[split][0] + intervals[split][1]) / 2.0;
  myPendingIntervals[split][1] = (intervals[split][0] + intervals[split][1]) / 2.0;
  // Aubereitung der Knotennachbarn, join.getSource() ist die ID des neuen Knotens.
  myPendingNeighbors = getNeighbors(node, candidates, myPendingIntervals, pid);
  othersPendingNeighbors = getNeighbors(join.getSource(), candidates, othersPendingIntervals,
      pid);
  myPendingNeighbors.add(join.getSource());
  // Klonen des bisherigen Caches und Bearbeitung der Elemente des neuen Knotens.
  CacheIndex cache = this.cache.getCloneofCache();
  cache.remove(awqlRd.createremoveRestriction(MGeometryFactory.createPolygon(othersPendingIntervals)));
}
```

Der gesplittete Knoten erstellt eine WELCOME-Nachricht, welche an den neuen Knoten geschickt wird. In dieser WELCOME-Nachricht werden die Variablen *othersPendingIntervals* , *othersPendingNeighbors* , *pendingSplit* und *cache* integriert. Nach dem Verschicken wird der gesplittete Knoten auf den Zustand *AWAITING_ACCEPT* gesetzt.

WELCOME

Eine WELCOME-Nachricht wird nur von einem neu angemeldeten Knoten empfangen. Diese Nachricht enthält alle Informationen, welche für den Betrieb im Content Adressable Network benötigt werden. Diese Informationen werden nach Erhalt in die eigenen, globalen Variablen abgespeichert, was im Programmcode 5.4 deutlich gemacht wird.

Listing 5.4 Erhalt einer WELCOME-Nachricht

```
if(event instanceof Welcome) {
  this.intervals = welcome.getAssignedZone();
  this.cache = new Cache(this, this.intervals);
  this.cache.setCache(welcome.getCache());
  this.split = welcome.nextSplit();
  // Speicherung der Nachbarn im CANLink-Protokoll
  ((CANLink) node.getProtocol(LINK_PID)).setNewNborSet(welcome.getNeighborSet());
  // Benachrichtigung der Nachbarn über den neuen Knoten
  updateNeighbors(node, pid);
}
```

Die Methode *updateNeighbors* verschickt eine UPDATE-Nachricht an die benachbarten Knoten. Der neue Knoten wird auf den Zustand AWAITING_WELCOME_ACKUPDATES gesetzt. Da nach Erhalt der Bestätigungsnachrichten aller Nachbarn noch eine ACCEPT-Nachricht an den zuvor gesplitteten Knoten geschickt werden muss, wird der Verweis auf den gesplitteten Knoten, welcher in der WELCOME-Nachricht als Quelle geführt wird, noch in der Variable *pendingAcceptNode* gespeichert.

ACCEPT, UPDATE

Eine ACCEPT-Nachricht wird von einem Knoten verschickt, welcher zuvor eine WELCOME-Nachricht erhalten hat. Der Empfänger einer ACCEPT-Nachricht ist der Knoten, welcher zuvor gesplittet und von dem die vorher eingehende WELCOME-Nachricht erstellt worden ist. Eine ACCEPT-Nachricht hat außer der Quelle keine weiteren Informationen. Nach dem Erhalt einer ACCEPT-Nachricht werden alle Nachbarn nun mittels UPDATE-Nachricht über den neuen Netzwerkzustand informiert. Der Zustand des Knotens wird von AWAITING_ACCEPT auf AWAITING_ACCEPT_ACKUPDATES gestellt.
Wird eine UPDATE-Nachricht erhalten, müssen lediglich die Informationen über den Knoten

in der Nachrichtenquelle geändert werden. Nach dem Update-Vorgang der Informationen wird eine ACKUPDATE-Nachricht zurückgeschickt.

ACKUPDATE

Werden UPDATE-Nachrichten an die Nachbarknoten verschickt, wird in einer Liste vermerkt, von welchen Knoten noch eine Bestätigung, also eine ACKUPDATE-Nachricht, erwartet wird. So lange nicht alle Bestätigungen eingegangen sind, wird der Zustand im Knoten nicht verändert. Wird vom letzten Nachbarknoten eine ACKUPDATE-Nachricht empfangen, muss unterschieden werden, ob der Knoten ein zuvor gesplitteter Knoten ist, oder ob der Knoten aufgrund eines zuvor gesplittenen Knoten hervorgegangen ist. Im ersten Fall werden alle Informationen, welche in Zwischenvariablen (*myPendingNeighbors* , *myPendingIntervals* , ...) vorliegen, in die eigentlich richtigen, globalen Variablen umgespeichert und der Zustand auf ACTIVE gesetzt. Da im zweiten Fall die Informationen bereits in den globalen Variablen gespeichert sind, wird der Zustand nur noch auf ACTIVE gesetzt. Der Splittvorgang ist somit in allen Punkten beendet.

QUERY, ROUTE

Das CAN-Protokoll bearbeitet Querys nicht selbst, sondern überprüft nur, ob die Query den eigenen Datenraum schneiden. Hierfür wird mit Hilfe des AWQLReaders die AwqlQuery eingelesen und die Geodaten ausgelesen. Diese ausgelesenen Geodaten werden mit dem eigenen Polygon geschnitten, welches mit Hilfe der Klasse MGeometry aus dem Datenraum erzeugt wird. Ist eine Schnittmenge vorhanden, so wird die Query an die Cacheinstanz zur weiteren Verarbeitung weitergegeben. Ist keine Schnittmenge vorhanden, so wird innerhalb des gesuchten Gebietes der AwqlQuery ein zufälliger Punkt bestimmt und die QUERY-Nachricht mit Hilfe einer ROUTE-Nachricht dorthin weitergeleitet.
Wird eine ROUTE-Nachricht empfangen, so kann der Inhalt nur eine QUERY-Nachricht sein. Es wird geprüft, ob der Zielpunkt der ROUTE-Nachricht im eigenen, zugewiesenen Datenraum liegt, wenn dies der Fall ist, wird die QUERY-Nachricht aus der ROUTE-Nachricht extrahiert und an die Cacheinstanz weitergegeben, sollte der Punkt jedoch nicht im Datenraum liegen, wird die ROUTE-Nachricht an den nächstliegenden Knoten weitergeleitet.

5.5. Cache-Realisierung

Ursprünglich sollte die Cache-Realisierung ebenfalls als ein eigenständiges Cache-Protokoll im Peersim integriert werden. Jedoch ist das Zusammenspiel zwischen dem CAN-Protokoll und dem Cache sehr ausgeprägt, oftmals werden gegenseitig Methoden aufgerufen, was bei einer Protokoll-Realisierung sehr umständlich gewesen wäre. Im Allgemeinen muss

nun unterschieden werden zwischen dem eigentlichen Cache, der Elemente aufnimmt, und der Cacheverwaltung, welche die „Vorarbeit“ leistet. Nach außen hin stellen beide Klassen eine Einheit dar, daher wurde bislang immer nur von „dem Cache“ gesprochen. Im Folgenden wird mit „Cache“ die Speicherstruktur beschrieben und mit der *Cacheverwaltung* die Verwaltung und Vorarbeit des Caches. Des Weiteren ist mit dem Begriff „Server“ der Hintergrundspeicher definiert, welcher vom Cache angefragt wird, wenn die Elemente nicht im Cache liegen.

5.5.1. Cacheverwaltung

Die Cacheverwaltung stellt die Schnittstelle zwischen dem CAN-Protokoll und dem Cache dar. Auch die Cacheverwaltung ist in der Lage, eingehende Nachrichten zu bearbeiten und auch neue Nachrichten zu erstellen. Eine QUERY-Nachricht ist die einzige Nachricht, welche vom CAN-Protokoll und der Cacheverwaltung bearbeitet wird. Bei allen anderen Nachrichten existiert eine strikte Trennung der Zuständigkeiten. Die Cacheverwaltung besitzt eine Reihe von globalen Variablen, um die Verwaltung möglichst effizient zu gestalten.

- *area:* Die Variable *area* ist vom Typ *Polygon* , einem Typ aus dem Paket JTS. Da in der Cacheverwaltung viele geometrische Operationen durchgeführt werden, wird dieses Polygon selbst geführt. Wird im CAN-Protokoll ein Schnitt durchgeführt, wird eine set-Methode in der Cacheverwaltung aufgerufen, um die neuen Grenzen zu übergeben und ein neues Polygon zu erstellen.
- *cache:* Die Variable *cache* verweist auf die Speicherstruktur, in welcher die Elemente verwaltet werden. *cache* ist vom Typ *PredicateIndex* , welche im Vorfeld zur Verfügung gestellt worden ist.
- *awqlread:* Eine Instanz des AwqlReaders wird ebenfalls global geführt, um Schnitte zwischen der AwqlQuery und dem *area* -Polygon durchführen zu können.
- *processcounter:* Wird eine QUERY-Nachricht als initial bearbeitet[1], so muss die Cacheverwaltung zunächst eine eindeutige Prozess-ID vergeben. Die Process-ID ist ein zusammengesetzter String aus der Knoten-ID des Knotens, auf welchem die Bearbeitung der AwqlQuery läuft und einer fortlaufenden Zahl, welche bei jedem Initialfall um eins erhöht wird. Die durchlaufende Nummerierung wird in der Variable *processcounter* gespeichert, welche mit dem Wert 0 initialisiert wird. So wird eine eindeutige Prozess-ID im Peer-to-Peer-Netzwerk sichergestellt.
- *processed:* Die Variable *processed* ist vom Typ *HashSet<String>* und speichert die bereits bearbeiteten Prozess-IDs, um eine Doppelberechnung zu verhindern. Eine Prozess-ID wird erst aus der *processed*-Liste gelöscht, wenn die AwqQuery im kompletten Peer-to-Peer-Netzwerk abgeschlossen ist.

[1] Der Knoten ist somit der Initialknoten

- *owner:* Die Variable *owner* ist vom Typ *CANProtocol* und verweist auf das CAN-Protokoll des Knotens. Das CAN-Protokoll hat die Cacheverwaltung initialisiert.
- *state:* Die Variable *state* ist vom Typ *ProcessState* und speichert die aktuellen Zustände bezüglich der Berechnung der AwqlQuerys.

Die Cacheverwaltung besitzt zwei Konstruktoren. Der erste Konstruktor wird nur bei der Initialisierung des ersten Knotens aufgerufen, es wird der Verweis auf die Serverinstanz übergeben und der Verweis auf das CAN-Protokoll. In diesem Konstruktor wird dann auch ein leerer Cache initialisiert, bei dem der Verweis auf den Server ebenfalls übergeben wird. Der zweite Konstruktor wird aufgerufen, wenn der Datenraum des Knotens gesplittet worden ist. Bei der Splittung wird eine Kopie des Caches vom gesplitteten Knoten übergeben, daher ist der Verweis auf den Server sowie die Neuinitialisierung des Caches nicht notwendig. Neben dem Verweis auf das CAN-Protokoll werden noch die Bereichsgrenzen des Datenraumes übergeben.
Auch die Cacheverwaltung ist zum Empfangen und Versenden von Nachrichten berechtigt, jedoch geschieht dies nur auf indirekte Weise. Da die Cacheverwaltung kein Peersim-Protokoll ist, sondern eine eigenständige Klasse, erbt diese nicht die Methode *processEvent* , wie es beim CAN-Protokoll der Fall ist. Deshalb werden alle Nachrichten über das CAN-Protokoll empfangen. Das CAN-Protokoll leitet jedoch die für die Cacheverwaltung relevanten Nachrichten, mit Ausnahme einer QUERY-Nachricht, unbearbeitet weiter. Diese Nachrichten enthalten meist Informationen, welche die aktuelle Berechnung einer AwqlQuery betreffen.

QUERY

Wird eine QUERY-Nachricht an die Cacheverwaltung weitergeleitet, so ist bereits garantiert, dass diese auch für diesen Knoten bestimmt ist. Das CAN-Protokoll überprüft die Schnittmenge zwischen dem zugewiesenen Datenraum und dem Abfragegebiet der AwqlQuery. Sollte keine Schnittmenge vorhanden sein, wird die QUERY-Nachricht an den zuständigen Knoten weitergeleitet. Eine QUERY-Nachricht enthält die Variable *processid* , um diese an die Nachbarknoten, falls notwendig, weiterzuschicken. Erhält die Cacheverwaltung eine QUERY-Nachricht, so ist anfangs nicht bekannt, ob der Knoten der Initialknoten ist, oder ob der Knoten nur an einer bereits gestarteten Berechnung teilnimmt. Erst wenn die Variable *processid* aus der QUERY-Nachricht ausgelesen wird, kann diese Frage beantwortet werden. Ist die Variable *processid* leer, so wurde die Berechnung noch nicht eingeleitet, der Knoten ist somit der Initialknoten und die Cacheverwaltung vergibt eine eindeutige Prozess-ID. Um die bereits vorgestellte, globale Ersetzungsstrategie durchzuführen, wird eine QUERYBLOCK-Nachricht an alle Nachbarknoten verschickt, um diesen zu signalisieren, welches Anfragemuster demnächst in deren Datenräumen auftauchen könnte. Da es sich bei der globalen Ersetzungsstrategie nur um eine Zukunftsvermutung handelt, wird die QUERYBLOCK-Nachricht mit einem Timer versehen, der angibt, wie lange Cacheelemente, welche dem gemeldeten Anfragemuster entsprechen, von der lokalen Ersetzungsstrategie

ausgeschlossen werden. Ist ein Knoten der Initialknoten, so wird in der Variable *state* der Absendeknoten als *firstnode* gespeichert, welcher somit der Anfrageknoten ist. An diesen Knoten muss nach Beendigung der Berechnung die Lösungsmenge zurückgeschickt werden. Da bei der Weiterleitung mit Hilfe der ROUTE-Nachrichten die Original-Nachricht unverändert weitergeschickt wird, ist somit der in QUERY-Nachricht dieser Knoten als Quelle zu finden. Für den Fall, dass der Initiialknoten und der Anfrageknoten gleich sind, wird die Variable *firstnode* in der Variablen *state* auf NULL gesetzt. Sollte der Knoten, der die QUERY-Nachricht erhalten hat, bereits eine Prozess-ID vorfinden, so nimmt dieser Knoten als zusätzlicher Knoten an dieser Berechnung teil. In diesem Fall ist als Quelle in der QUERY-Nachricht der Knoten angegeben, welcher die QUERY-Nachricht weitergeleitet hat. Dies bedeutet, dass dieser Knoten ein Nachbarknoten sein muss. Dieser Knoten wird als *owner* in der Variablen *state* gespeichert, an diesen muss die Lösungsmenge zurückgeschickt werden. Der Knoten weiß jedoch nicht, ob der *owner*-Knoten auch der Initialknoten ist. Nachdem nun die Rollenverteilung bei der Berechnung einer AwqlQuery bekannt sind, wird nun überprüft, ob die Prozess-ID bereits bearbeitet worden ist. Hierfür wird in der Liste *processed* nach der Prozess-ID gesucht. Wird diese gefunden, ist die Lösungsmenge bereits erstellt und zurückgeschickt worden, der Nachbarknoten, welcher die AwqlQuery ein zweites Mal an den Knoten geschickt hat, erhält eine leere Lösungsmenge in einer REPLY-Nachricht zurück. Abbildung 5.2 zeigt, wie die QUERY-Nachrichten zwischen den Knoten verschickt werden. *[QUERY** bedeutet, dass in dieser QUERY-Nachricht bereits eine Prozess-ID vorhanden ist, welche von Knoten A erstellt worden ist. Abbildung 5.3 zeigt die daraus resultierenden REPLY-Nachrichten. *REPLY(N+M)* bedeutet, dass die REPLY-Nachricht die Lösungsmengen der Knoten *N* und *M* enthält. In dem Beispiel erhält der Knoten C die QUERY-Nachricht von Knoten B früher als von Knoten D.

Ein Knoten, der die AwqlQuery bearbeitet, unabhängig davon, welche Rolle dieser Knoten in der Berechnung einnimmt, muss die AwqlQuery geklont und das Gebiet mit dem Polygon in der Variablen *area* geschnitten werden. Dieser Schnittbereich wird in der AwqlQuery fest gespeichert. Damit soll sichergestellt werden, dass der Cache, der den zugewiesenen Datenraum nicht kennt, nur die notwendigen Elemente ermittelt. Nachdem die AwqlQuery bearbeitet worden ist, wird diese an den Cache übergeben, welcher die Lösungsmenge ermittelt. Diese Lösungsmenge wird in der Klasse *ResultSet* gespeichert und zurückgeliefert. Diese *ResultSet* wird dann in die Klasse *AWMLReader* gespeichert, welche die Lösungsmengen verwaltet. Der AWMLReader ist in der Lage, die Lösungsmenge so zu erweitern, dass doppelte Elemente eliminiert werden. Muss die Query an die Nachbarknoten weitergeleitet werden, da diese ebenfalls im Gebiet der AwqlQuery liegen, so wird an diese Nachbarknoten die Original-AwqlQuery weitergereicht, nicht die bereits beschnittene. Diese Nachbarknoten werden in der Variable *state* gespeichert, damit sichergestellt wird, dass alle Lösungsmengen, welche auch leer sein können, eingegangen sind. Wird kein Nachbarknoten für die Berechnung benötigt, so wird, nachdem der Cache die Lösungsmenge erstellt hat, diese in den AWMLReader gepackt und mittels REPLY-Nachricht an den *owner* geschickt. Wenn die *owner*-Variable leer ist, ist der Knoten der Initialknoten und muss die REPLY-Nachricht an

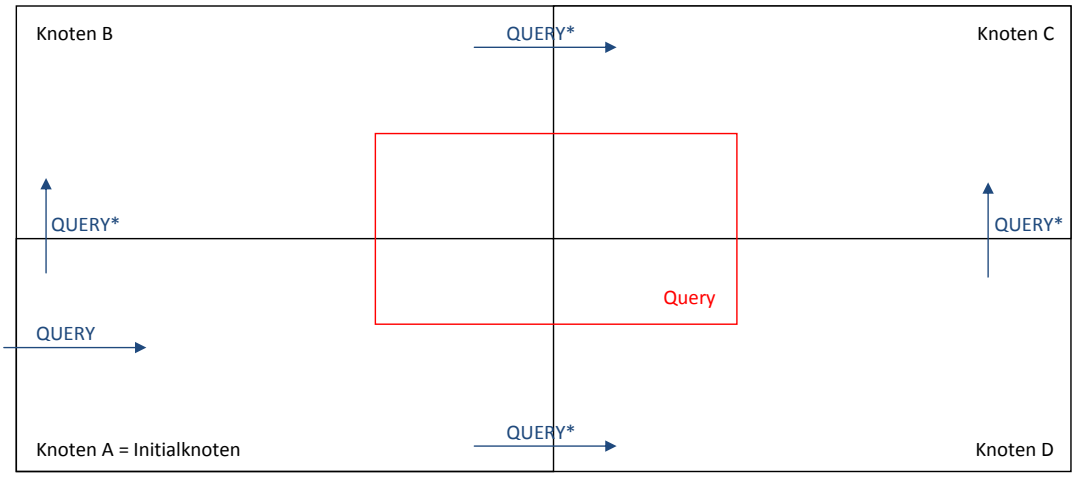


Abbildung 5.2.: QUERY-Versendemuster

den *firstnode* schicken. Ist auch diese Variable leer, so ist der Knoten neben dem Initialknoten gleichzeitig der Anfrageknoten und die Lösungsmenge wird an den Benutzer geschickt.

REPLY

Eine REPLY-Nachricht beinhaltet die Lösungsmenge eines anderen Knotens,das für die Berechnung der AwqlQuery benötigt worden ist. REPLY-Nachrichten werden auf dem selben Pfad zurückgeschickt, die QUERY-Nachrichten den Pfad erzeugt haben. In der Variablen *state* befindet sich eine Erwartungsliste, von welchen Knoten noch eine REPLY-Nachricht erwartet wird. Wird eine REPLY-Nachricht empfangen, wird der Quellknoten aus der Erwartungsliste in der Variablen *state* gelöscht. Der AWMLReader, welcher in der REPLY-Nachricht enthalten ist, wird mit dem eigenen AWMLReader vereinigt. Hierfür ist in der Klasse AWMLReader die Methode *AWMLReader.expand(AWMLReader)* vorhanden. Ist der Quellknoten der REPLY-Nachricht der letzte Knoten in der Erwartungsliste, so ist für den Knoten die Berechnung der Lösungsmenge für die AwqlQuery beendet. Die Lösungsmenge im AWMLReader wird dann an den *owner* , bzw. *firstnode* oder Benutzer geschickt. Ist der Knoten der Initialknoten der Berechnung, so ist die Berechnung der AwqlQuery komplett abgeschlossen. Um das Volllaufen der *processed*-Liste zu verhindern, wird eine PROCESSFINISH-Nachricht an die Nachbarknoten verschickt, welche neben der Prozess-ID, welche abgeschlossen ist, noch den neuen Füllstatus des Cache enthält. Der Füllstatus ist in der Hinsicht wichtig, da dieser für

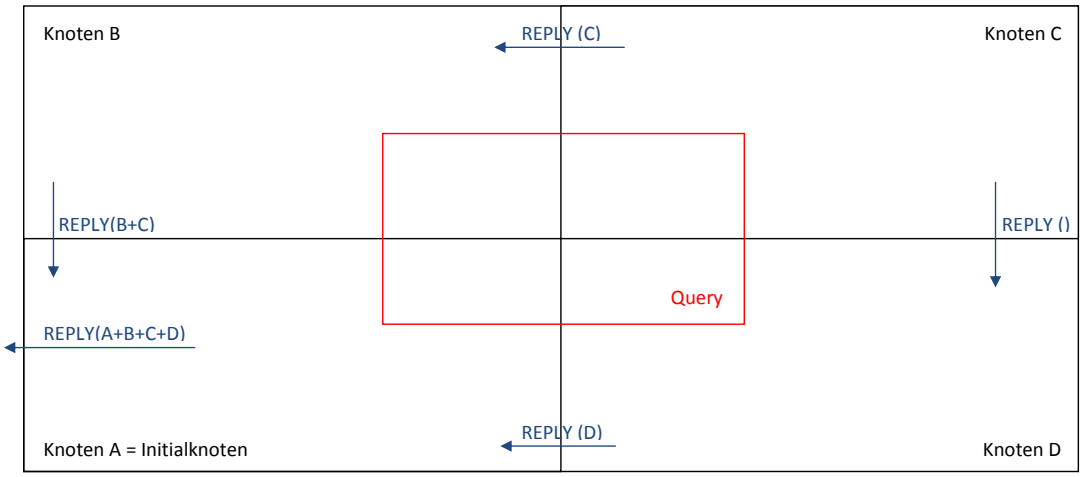


Abbildung 5.3.: REPLY-Versendemuster

die Knotenermittlung beim Splittvorgang benötigt wird. Jeder Knoten kennt den Füllstatus seiner Nachbarknoten.

PROCESSFINISH

Die PROCESSFINISH-Nachricht ist nach dem Entwurf entstanden. Während des Entwurfs wurde nicht beachtet, dass die Liste, welche die bearbeiteten Prozess-IDs speichert, eine enorme Größe annehmen kann. Um das Wachsen dieser Liste zu verhindern, wird nach Abschluss einer AwqlQuery-Berechnung eine PROCESSFINISH-Nachricht vom Initialknoten verschickt, um zu signalisieren, dass keine QUERY-Nachricht mit dieser Prozess-ID mehr vorhanden ist. Die PROCESSFINISH-Nachricht wird mit Hilfe von lokalem Flooding verschickt. Wird eine PROCESSFINISH-Nachricht erhalten, wird geprüft, ob die darin enthaltene Prozess-ID in der *processed*-Liste enthalten ist. Wenn dies der Fall ist, wird die Prozess-ID aus dieser Liste entfernt und der darin enthaltene Füllstatus des Nachbarknotens gespeichert. Die PROCESSFINISH-Nachricht wird daraufhin an die eigenen Nachbarn weitergeschickt, auch an den Knoten, von welchem die PROCESSFINISH-Nachricht kam, jedoch wird der darin enthaltene Füllstatus durch den eigenen ersetzt. Somit werden auch Änderungen, welche nur aufgrund von AwqlQueries verursacht werden können, an die Nachbarknoten vermittelt. Ist jedoch die Prozess-ID nicht in der *processed*-Liste enthalten, wird die PROCESSFINISH-Nachricht nicht weitergeleitet, endet an dieser Stelle somit das lokale Flooding.

QUERYBLOCK

Eine QUERYBLOCK-Nachricht wird verschickt, wenn eine AwqlQuery bearbeitet wird. Die QUERYBLOCK-Nachricht wird für die globale Ersetzungsstrategie benötigt. Wird eine QUERYBLOCK-Nachricht empfangen, so wird die darin befindliche AwqlQuery an den Cache weitergegeben, welcher diese in eine „Sperrliste“ aufnimmt. Elemente, welche als Lösung für eine der AwqlQuery-Anfragen in der Sperrliste in Frage kommen, können trotz lokaler Ersetzungsstrategie nicht gelöscht werden. Eine Weiterleitung von QUERYBLOCK-Nachrichten wird zu keinem Zeitpunkt durchgeführt.

5.5.2. Cache

Die Datenstruktur wurde im Zuge dieser Diplomarbeit zur Verfügung gestellt, es wurden lediglich einige Schnittstellen angepasst. Daher wird auf die Funktionsweise nicht im Detail eingegangen und es werden nur die wichtigsten Merkmale erläutert.
Der zur Verfügung gestellte Cache ist ein auf Prädikatenvergleiche basierter Cache. In diesem Cache werden Unterschiede zwischem dem Begriff „Cacheelemente“ und dem Begriff „Objekte“ gemacht. Bisher wurde davon gesprochen, dass eine AwqlQuery an den Cache gestellt wird, welcher eine Lösungsmenge erzeugt, notfalls mit Hilfe des Servers. Eine AwqlQuery beinhaltet einen oder mehrere RestrictionOperatoren, welche die Beschreibung der Lösungsmenge dient. Diese RestrictionOperatoren werden im Cache als „Cacheelemente“ bezeichnet, die Inhalte der Lösungsmenge werden „Objekte“ genannt. Jedes Objekt ist mindestens einem Element zugewiesen, da aber Objekte vielfältig sein können, können sie auch bei unterschiedlichen AwqlQueries ermittelt werden. Wird durch die *query*-Methode eine Anfrage an den Cache gestellt, so wird der darin befindliche RestrictionOperator untersucht und in den bestehenden Cacheelementen geprüft, in wiefern diese mit dem RestrictionOperator der AwqlQuery übereinstimmen. Die sich darin befindenden Objekte werden in einer vorläufigen Lösungsmenge zusammengestellt und dann in einem Feinvergleich mit der AwqlQuery verglichen. Der Vorteil an dieser Vorgehensweise ist, dass bereits im Vorfeld Objekte, welche überhaupt nicht in der Lösungsmenge enthalten sein können, von der Feinkontrolle ausgeschlossen werden. Des Weiteren kann sogar sehr schnell ermittelt werden, ob überhaupt Objekte im Cache der AwqlQuery entsprechen, um einen sofortigen Serverzugriff einzuleiten. Der Quellcode 5.5 zeigt, wie sich der Cache diese Speicherstruktur zu Eigen macht.

Ein weiterer Vorteil der Cacheelemente ist, dass bekannt ist, über welches Gebiet sich die Objekte im Cache verteilen, dies bedeutet, dass somit kein weiteres Objekt in dem bekannten Gebiet sein können. Mit Hilfe dieser Information ist es nun möglich, einen *remainder* zu erstellen, welcher die AwqlQuery so bearbeitet, dass nur noch jene Objekte vom Server geholt werden, welche sich nicht im Cache befinden. Der „remainder“ verhindert somit das Abrufen bereits vorhandener Objekte vom Server. Ist der *remainder* leer, so kann auf den

Listing 5.5 Erstellung der Lösungsmenge im Cache

```
public synchronized ResultSet query(AwqlQuery query) {
 AwqlQuery tempQuery = (AwqlQuery) query.clone()
 // Durchsuchung aller vorhandenen Cacheelemente und Vergleich mit der AwqlQuery.
 for (CacheElement element : cacheElements) {
  try {
        element.getIntersection(tempQuery);
  } catch (AWSException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
  if (!element.intersectionIsEmpty()) {
   cut.add(element);
  }
 }

 // Alle Cacheelemente, welche der AwqlQuery entsprochen haben, werden in die Feinkontrolle
     genommen. Hierfür werden die Objekte aus den Elementen ausgelesen.
 for (CacheElement element : cut) {
  Collection<GenericObject> objects = element.getObjectSet();
  for (GenericObject object : objects) {
  // Erstellen der Lösungsmenge
  if (tempQuery.getRestriction().evaluateOperator(object))
   rs.add(object);
  }
 }
}
```

Serverzugriff verzichtet werden, da sichergestellt ist, dass keine weiteren Objekte vom Server erwartet werden.
Werden Objekte vom Server geholt, werden diese im Cache hinterlegt, falls noch ausreichend Speicherkapazität vorhanden ist. Ist dies nicht der Fall, wird die lokale Ersetzungsstrategie durchgeführt. Wurden Objekte ermittelt, welche gelöscht werden sollen, werden diese noch mit der Sperrliste verglichen, in der AwqlQueries gespeichert sind, die in naher Zukunft an den Cache gestellt werden. Sind die Objekte, welche gelöscht werden sollen, in der Lösungsmenge einer dieser AwqlQueries, so können diese nicht gelöscht werden, die lokale Ersetzungsstrategie muss somit neue Objekte ermittelt, welche für das Löschen in Frage kommen. Sollten die Objekte keiner der in der Sperrliste gespeicherten AwqlQueries entsprechen, können diese gelöscht werden. Sollte der Fall eintreten, dass alle Objekte aufgrund der globalen Ersetzungsstrategie nicht gelöscht werden können, so wird die globale Ersetzungsstrategie außer Acht gelassen und nur die lokale Ersetzungsstrategie verwendet.
Der Cache stellt neben der wichtigen „query-Methode“ folgende, erwähnenswerte Methoden zur Verfügung, die nach außen hin sichtbar sind:

- *cacheFillState:* Der Füllstatus im Cache wird ausgegeben. Dieser Füllstatus liegt zwischen den Werten 0 und 1. Der Füllstatus errechnet sich anhand der Objekte im Cache und der maximalen Kapazität, welche jedoch veränderbar ist.
- *getSize:* Diese Methode gibt die aktuelle Anzahl an Objekten im Cache wieder.
- *getMaxSize:* Ausgabe der maximalen Speicherkapazität im Cache.
- *objectBasedHitMissRatio:* Ausgabe einer Metrik, wie sich die Anzahl der vom Cache geholten Objekte zu der Anzahl vom Server geholten Objekte verhält.
- *clone:* Eine Kopie des Caches, inklusive der Cachelemente und Objekte wird erstellt.

Eine weitere, sehr wichtige Methode ist die *remove*-Methode im Cache. Diese Methode wird aufgerufen, wenn sich im Content Adressable Network ein neuer Knoten anmeldet und daher ein Splittvorgang des Datenraumes eines bestehenden Knotens durchgeführt werden muss. Der Cache, welcher auf dem zu splittenden Knoten existiert, wird geklont und danach mit der *remove*-Methode bearbeitet. Die „remove"-Methode nimmt einen RestrictionOperator entgegen, also keine komplette AwqlQuery. Die Methode entfernt alle Objekte aus dem Cache, welche dem übergebenen RestrictionOperator entsprechen. Der RestrictionOperator wird mit Hilfe des AwqlReaders erstellt und als Übergabeparameter werden die neuen Bereichsgrenzen des Datenraumes verwendet. Der RestrictionOperator erhält somit die Form „alles, was den übergebenen Bereich nicht schneidet". Da die Objekte gelöscht werden, die der *Removerestriction* entsprechen, wird die Aussage „lösche alles, was den übergebenen Bereich nicht schneidet" komplettiert und der Cache wurde erfolgreich bearbeitet.
Zwar verfügt der Cache noch über weitere Methoden, doch diese werden lediglich für die interne Berechnung der Lösungsmenge, dem Einfügen neuer Objekte, welche vom Server geholt worden sind oder dem Löschen der Objekte nach der Aufforderung über die *remove*-Methode verwendet. Daher werden diese nicht weiter erläutert.

5.5.3. ProcessState

Ein eher kleines Programmstück ist der Prozesstatus, welcher in der Klasse *ProcessState* geführt wird. Da sich die AwqlQueries über mehrere Knoten erstrecken kann, ist es notwendig, die einbezogenen Knoten zu speichern, um keine Lösungsmenge außer Acht zu lassen. Da auf einem Knoten mehrere AwqlQueries gleichzeitig bearbeitet werden können, ist es notwendig, die Informationen über den Prozessstatus in Listen zu speichern, wobei die eindeutige Prozess-ID als Listenschlüssel eingesetzt wird.

- *ownerlist:* Zu jeder Prozess-ID kann es nur einen „Besitzer" geben. Der Besitzer einer AwqlQuery ist der Knoten, von welchem die AwqlQuery weitergeleitet worden ist, wobei die Weiterleitung durch ROUTE-Nachrichten nicht beachtet wird. Der Initialknoten ist der Knoten, welcher keinen Besitzer hat. Ist der Intialknoten gezwungen, die AwqlQuery an seine Nachbarknoten weiterzugeben, führen diese Nachbarknoten den Initialknoten als Besitzer. Muss ein Nachbarknoten die AwqlQuery ebenfalls an

einen seiner Nachbarknoten weiterleiten, so ist der „Besitzer“ der Nachbarknoten des Intialknotens. Die erstellte Lösungsmenge wird immer an den Besitzer zurückgeschickt.

- *firstnodelist:* Der „erste“ Knoten in einer Berechnung ist der Anfrageknoten, also der Knoten, welcher vom Benutzer die AwqlQuery entgegengenommen hat. Der erste Knoten stellt somit eine Sonderform des Besitzers dar. Ist im Prozessstatus kein Besitzer vorhanden, so wird geprüft, ob der erste Knoten vorhanden ist. Ist dies der Fall, so ist der derzeitige Knoten der Initialknoten, sind beide Einträge leer, so ist der Anfrageknoten zugleich der Initialknoten.
- *nodelist:* Die „Knotenliste“ ist die komplexeste Liste, da dies eine Liste von Listen ist. Da eine AwqlQuery an mehr als einen Knoten weitergeleitet werden kann, müssen all diese Knoten gespeichert werden. Somit gibt es zu jeder Prozess-ID eine eigene Knotenliste, während es zu jeder Prozess-ID immer genau einen Besitzer gibt, falls vorhanden, und immer genau einen ersten Knoten, falls dieser existiert. Wird eine AwqlQuery an einen Nachbarknoten weitergeleitet, so wird dieser Nachbarknoten in der Knotenliste gespeichert, wird von diesem Knoten eine REPLY-Nachricht empfangen, so wird dieser Knoten wieder aus der Knotenliste gelöscht.
- *rslist:* In der Lösungsmengenliste wird zu jeder Prozess-ID die aktuelle Lösungsmenge gespeichert, so lange die Berechnung der Query nicht abgeschlossen ist. Wird von einem Knoten in der Knotenliste eine REPLY-Nachricht empfangen, wird die sich darin liegende Lösungsmenge mit der Lösungsmenge in der Lösungsmengenliste vereint. Wird eine REPLY-Nachricht vom letzten Knoten in der Knotenliste empfangen, so wird die dann erzeugte Lösungsmenge in der Lösungsmengenliste an den Besitzer, Ersten Knoten oder Benutzer weitergeleitet.

Für jeden Knoten gibt es somit immer nur eine Prozessstatus-Klasse, in welcher eine Unterscheidung der Prozess-IDs stattfindet. Alternativ könnte man auch eine Liste von Prozessstatus-Klassen erzeugen, wobei die Prozess-ID ausgelagert wird und diese dann als Schlüssel für die Prozessstatus-Listen verwendet werden kann.

5.5.4. Abweichungen

Wie bereits eingangs erwähnt, wurden für die Realisierung einige Sonderfälle nicht implementiert, welche jedoch beachtet werden müssen. Die wohl größte Erweiterung ist die Handhabung des NearestNeighborsOperators. Dieser ist meist zusätzlich zum Restriction-Operator in einer AwqlQuery eingebettet. Der Hauptunterschied liegt darin, dass von einem bestimmten Punkt - meist der aktuelle Standpunkt des Benutzers - eine feste Anzahl von Objekten gesucht werden soll; es wird also kein festes Gebiet übergeben. Diese Handhabung gestaltet sich deswegen so schwierig, weil in den Cacheelementen das bereits bekannte Gebiet gespeichert wird. Um den NearestNeighborsOperator dennoch auf die bestehende Cachestruktur zu „trimmen“, muss ein entsprechendes Gebiet erstellt werden. Da immer die nächstgelegenen Objekte zu einem Punkt ermittelt werden sollen, kann mit Hilfe des am

weitest entfernten Objektes ein Gebiet spezifiziert werden, indem kein Objekt näher am Ausgangspunkt ist als die bisher gefundenen. Somit kann die Entfernung zwischen Standpunkt und dem entferntesten Objekt als Radius genommen werden und damit ein Kreis erstellt werden mit dem Ausgangspunkt als Mittelpunkt. In diesem Kreisgebiet sind somit keine anderen Objekte vorhanden, außer diejenigen, die mit der AwqlQuery ermittelt worden sind. Da aber der geometrische Umgang mit Kreisen sehr komplex ist, muss auch hier Abhilfe geschaffen werden. Die genaueste Variante ist es, den Kreis als n-Eck-Polygon zu realisieren, wobei das n möglichst hoch sein sollte. Jedoch ist dann auch der Speicheraufwand für groß n am höchsten, auch die Erstellung dieses n-Ecks bedarf vieler rekursiver Durchläufe. Mit diesem ermittelten Gebiet kann der NearestNeighborsOperator korollar mit dem „einfachen" AwqlQueries gleichgestellt werden.
Das Berechnen einer Lösungsmenge mit Hilfe des Caches gestaltet sich ebenfalls aufwendiger, da der Remainder nicht verwendet werden kann. Wird eine AwqlQuery mit einem NearestNeighborsOperator an den Cache übergeben, so muss wiederum ein Gebiet für die Durchsuchung im Cache erstellt werden. Dieses Gebiet errechnet sich aus der Entfernung von dem Abfragepunkts bis zu dem naheliegendsten, nicht bekannten Gebiet im Cache. In diesem, dann wieder kreisförmigen, Gebiet können die Cacheobjekte ermittelt werden, bei denen bekannt ist, dass es keine bessere Lösung vorhanden ist. Ist der Anfragepunkt bereits außerhalb eines bekannten Cacheelements, so kann der Server direkt befragt werden. Sind in dem kreisförmigen Gebiet weniger Objekte als im NearestNeighborsOperator verlangt sind, so muss der Server mit einbezogen werden.
Ein weiterer Punkt, welcher bei der Implementierung außer Acht gelassen worden ist, der sich jedoch massiv auf die Effizienz des Caches auswirken kann, sind die „vollständigen" Objekte, welche in der Realität nicht vollständig vom Cache geholt werden. Werden in einer AwqlQuery die *include*- oder *exclude*-Felder verwendet, werden nur Teilinformationen abgefragt, was dazu führt, dass trotz bekannter Objekte der Server dennoch einbezogen werden muss. Wird in einer AwqlQuery für die Objekte nur die Adresse verlangt, so muss bei einer weiteren AwqlQuery, die von den selben Objekten eine Telefonnummer möchte, der Server erneut einbezogen werden. Auf den Serverzugriff kann verzichtet werden, wenn das bekannte Gebiet im Cache ausreichend für die AwqlQuery ist, also sich die gesuchten Objekte im Cache befinden, und für die Attribute der Objekte gilt, dass die gesuchten Attribute eine Teilmenge der bereits bekannten Attribute der Objekte im Cache sind.
Ein weiterer, nicht implementierter Fall ist die Abwahl, beziehungsweise der Fehlerfall im Content Adressable Network. Zum einen ist der CAN-Baum notwendig, der die Splittvorgänge aufzeichnet, um die Knoten zu ermitteln, welche im Falle der Abwahl die Positionen ändern müssen. Wird eine kontrollierte Abwahl vollzogen, so werden die Inhalte des Caches an den Knoten weitergegeben, der den Platz des abzuwählenden Knotens übernimmt. Im Fehlerfall werden die Cacheinhalte nicht weitergegeben. Um einen Fehlerfall zu erkennen, müssen HEARTBEAT-Nachrichten in regelmäßigen Abständen an die Nachbarknoten verschickt werden.
Zu guter Letzt werden manche Informationen direkt über das von Peersim simulierte Netzwerk abgerufen, was in der Realität nicht möglich ist. Darunter fallen die Informationen

über den Datenraum der Nachbarknoten. Um diese Informationen dennoch zu haben, werden diese mit Hilfe der UPDATE-Nachrichten an die Nachbarknoten verschickt, welche diese Informationen in einer eigenen Liste speichern. Nach dem selben Prinzip funktioniert auch die Ermittlung des Cachefüllstatus, der mit einer PROCESSFINISH-Nachricht an die Nachbarknoten verteilt wird. Ein Cachefüllstatus kann sich nur ändern, wenn zuvor eine AwqlQuery bearbeitet worden ist, welche am Ende mit einer PROCESSFINISH-Nachricht abgeschlossen wird.

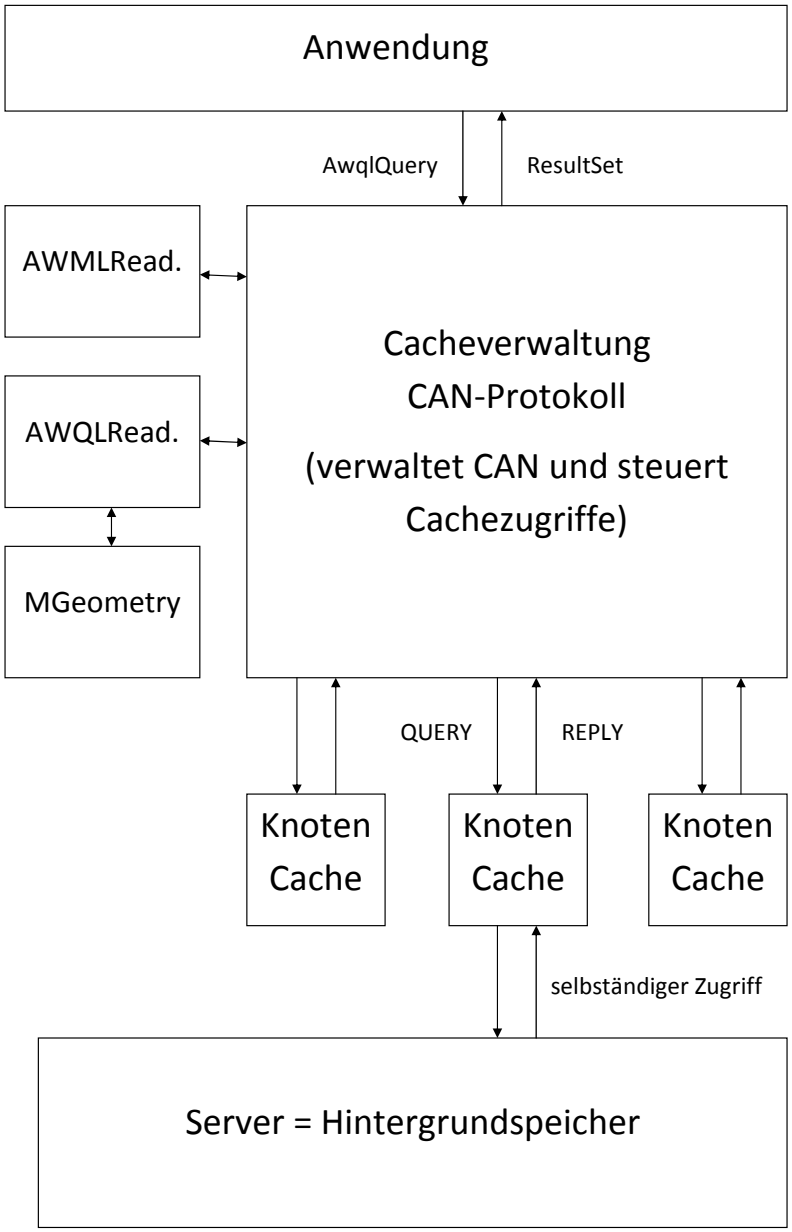


Abbildung 5.4.: Architektur der Realisierung

6. Anwendungsszenarien

Lang ist der Weg durch Lehren,
kurz und wirksam durch Beispiele.

(Marcus Lucius Annaeus Seneca - römischer Rhetoriker)

Um die vorgestellte Realisierung zu testen und entsprechende Messergebnisse zu erzeugen, wurden mehrere Anwendungsszenarien erstellt. Um diese Anwendungsszenarien erstellen zu können, sind einige Tools notwendig, welche kurz vorgestellt werden. Des Weiteren werden die Messinstrumente erklärt, welche verwendet worden sind, um die Messungen durchzuführen.

6.1. Vorbereitung

Die Vorbereitung für ein Anwendungsszenario umfasst die Simulation des Servers sowie das Erzeugen der AwqlQueries.

6.1.1. SpaSe

Der SpaSe ist eine Serversimulation, welche Anfragen entgegennimmt und eine Lösungsmenge zurückliefert. Damit die Serversimulation auch verwendet werden kann, müssen die benötigten Objekte erstmals in die Serverinstanz geladen werden. Diese Objekte befinden sich in einer Textdatei, welche ausgelesen wird.
Für die Simulation wurden für die Stadt Berlin eine Datei mit ca. 25.000 Objekten mit Hilfe von OpenStreetMap [ope04] erzeugt, welche in den SpaSe geladen werden. Da der SpaSe in seinen Grundzügen ein von der Nexusplattform bereitgestelltes Instrument ist, wird auf die Funktionsweise nicht näher eingegangen.

6.1.2. Queryerzeugung

Für die Erzeugung der AwqlQueries bedarf es einiger Hilfsklassen und Hilfsdateien. In diesem Zusammen sollen die Anfragen von sich stetig bewegenden Anfragepunkten gestellt werden. Um die Lesbarkeit zu erhöhen, werden im Nachfolgenden diese sich bewegenden

Objekte als „Fahrzeuge“ bezeichnet. Daher wurde eine „Bewegungsdatei“ erzeugt, welche für eine Anzahl von Fahrzeugen die aktuelle Position ausgibt. Diese Datei besitzt einen Kopf, in welchem die Gebietsgröße sowie die Anzahl der Fahrzeuge und deren Geschwindigkeitsbereich angegeben ist (Beispiel 6.1). Für die Erzeugung der AwqlQueries sind jedoch keine Geschwindigkeitsangaben notwendig, auch der Zeitpunkt, wann sich ein Fahrzeug wo befindet, ist irrelevant. Dem Kopf anschließend werden die Startpunkte der Fahrzeuge gesetzt (Beispiel 6.2), danach erfolgen dann die Bewegungen (Beispiel 6.3).

Listing 6.1 Datenkopf der Bewegungsdatei

```
#area: 46105.543 m x 38279.547 m\\
#100 nodes: 1.38889 (m/s) - 16.66667 (m/s)
```

Listing 6.2 Startpunkte der Fahrzeuge

```
0 36859.490484497626 23091.60168858245
1 13383.86452616268 19147.812018981203
2 11681.732079414243 21377.05381892342
3 14454.539171999553 15518.887074312195
4 10599.129974062263 22679.191656420007
5 10044.970660927007 19401.271518440917
6 32886.059470437525 21571.041816140525
7 22403.265978569398 24505.35209643282
8 20226.995256994443 20750.18527758587
9 28159.93360345828 29729.940581517294
10 15923.49677360797 16763.2461367622
```

Listing 6.3 Bewegung der Fahrzeuge zu einem bestimmten Zeitpunkt (5.0)

```
1 5.0 13407.827655748406 19155.10795537941 3.5201979
2 5.0 11648.923855919042 21413.13712416496 4.755482
5 5.0 9935.25671525154 19300.15835570637 10.1000805
7 5.0 22389.75208409631 24404.711518524215 15.813131
8 5.0 20185.822689275374 20738.336269216612 8.847631
9 5.0 15903.29053681367 16790.8921112353 6.2033086
```

Für das erste Anwendungsszenario werden 100 Fahrzeuge simuliert, die sich durch Berlin bewegen. Die angegebene Gebietsgröße besitzt eine metrische Skalierung, da aber das Anfragegebiet in einer AwqlQuery in WGS84-Koordindaten angegeben werden muss, ist es noch notwendig, den „Linksunten“-Punkt zu ermittelt, also bei welcher WGS84-Koordindate das in Metern angegebene Gebiet beginnt.
Für die Erzeugung von AwqlQueries wurde ein *AWQLGenerator* zur Verfügung gestellt, welcher mit Hilfe einer Konfigurationsdatei und mit der bereits beschriebenen Bewegungsdatei die benötigten AwqlQueries erzeugt. In der Konfigurationsdatei, welche im XML-Format geschrieben worden ist, werden folgende Einstellungsmöglichkeiten angeboten:

- *region:* Im *region* -Tag wird ein Polygon angegeben, das besagen soll, wie groß das Gebiet um den aktuellen Standpunkt ist, welches abgefragt werden soll, wobei die aktuelle Position des Fahrzeugs der Nullpunkt ist. Ein Beispiel für das *region* -Tag ist: POLYGON ((-100 -100, 100 -100, 100 100, -100 100, -100 -100)), was bedeutet, dass das Anfragegebiet eine Größe von 200 x 200 Metern hat.
- *lowerLeftCorner:* In der Konfigurationsdatein muss die WGS84-Koordindate angegeben werden, welche die linke untere Ecke des Gesamtgebietes darstellt.
- *Inputfile, Outputfile:* In der *Inputfile* wird der Pfad zu der Bewegungsdatei angegeben, im Tag *Outputfile* wird der Ausgabepfad für die Liste der AwqlQueries, welche in einer Textdatei geschrieben werden, angegeben.
- *templateMapping:* Es wird ein Template benötigt, welches das Grundgerüst einer AwqlQuery darstellen soll. Neben dem Pfad zu dem Template werden noch die IDs der Fahrzeuge angegeben, für welche diese Templates gelten. Somit ist der AWQLGenerator in der Lage, unterschiedliche AwqlQueries für verschiedene Fahrzeuge zu erzeugen. Im Anwendungsszenario suchen 20 Fahrzeuge alle Objekte im Suchgebiet, die nächsten 10 Fahrzeuge suchen Universitätsgebäude, die nächsten 10 Fahrzeuge sind auf der Suche nach Bars, und so weiter.

Ein Template, welches für die Erzeugung von AwqlQueries benötigt wird, ist an sich ebenfalls eine AwqlQuery, welche in eine Textdatei geschrieben ist. Dieses Template wird ausgelesen und das Anfragegebiet entsprechend der aktuellen Fahrzeugposition angepasst. Die Erzeugung von AwqlQueries wird aufgrund der geringeren Komplexität mit den Templates gestaltet, da die Erzeugung neuer AwqlQueries von Grund auf wesentlich komplizierter ist als das Auslesen bestehender Queries, bei dem ein Attribut geändert wird. Eine Datei, in welcher eine Vielzahl von AwqlQueries gespeichert sind, ist jedoch für das Anwendungsszenario noch nicht einsetzbar. Die in der Textdatei vorhandenen AwqlQueries müssen noch in die entsprechende AwqlQuery-Klasse geladen werden, damit die Nexusplattform und auch der Cache Anfragen bearbeiten können. Die in der Textdatei befindlichen AwqlQueries werden durch ein Trennungszeichen getrennt, um den Beginn einer neuen AwqlQuery zu verdeutlichen. Zum Auslesen wird die Klasse *MAWQLGenerator* verwendet, eine leicht abgeänderte Variante des zuvor vorgestellten *AWQLGenerators*. Der MAWQLGenerator nimmt einen String entgegen, also einen Bereich der Textdatei, und erzeugt darauf basierend eine AwqlQuery-Klasse. Die Erzeugungsmethode ist Bestandteil der Nexusplattform und musste daher nicht selbst implementiert werden. Der MAWQLGenerator liest auf diese Art die komplette Textdatei aus, und liefert eine Liste mit AwqlQuery-Klassen zurück, welche nun für die Simulation verwendet werden kann.

Für das zweite Anwendungsszenario wurde auf sich bewegende Fahrzeuge verzichtet, jedoch wurden sogenannte „Hotspots“. Ein „Hotspot“ ist ein Punkt, um den sich eine höhere Anfragedichte schart als in anderen Bereichen. Dies soll die Tauglichkeit der Realisierung untersuchen, da bekanntlich in einigen räumlichen Zentren ein höheres Menschenaufkommen ist als in eher abgelegenen Gebieten. In diesem Zusammenhang wurden die Konfigurations-

datei und der AWQLGenerator erweitert. In der Konfigurationsdatei können die Koordinaten für diese „Hotspots“ angegeben werden.
Zu guter Letzt ist es nun noch notwendig, in der Peersim-Konfigurationsdatei das Netzwerk zu bearbeiten, wie zum Beispiel die Größe des Netzwerkes. Für das Content Adressable Network werden zudem die Koordinaten für den Datenraum benötigt, die für das „Berlinszenario“ geändert werden müssen.

6.2. Programmdurchlauf

Wenn die Vorbereitungen erfolgreich abgeschlossen sind, ist eine Liste mit AwqlQueries vorhanden, sowie ein einsatzfähgier SpaSe, welcher die AWML-Datei auslesen kann und auch eine fertig eingestellte Konfigurationsdatei für die Peersim-Simulation und das Content Adressable Network. Die nachfolgenden Erläuterungen gelten für beide Anwendungsszenarien.
Peersim bietet für die Steuerung des Peer-to-Peer-Netzwerkes die Schnittstelle *Control* an, welche erweitert worden ist. Die daraus entstandene Klasse ist mit dem Namen *CANRandomJoin* bezeichnet, wird jedoch nachfolgend als „Kontrollklasse“ geführt wird. Die Kontrollklasse beginnt mit einer JOIN-Nachricht, welche an einen zufälligen Knoten im Netzwerk geschickt wird - bei der Erstinitialisierung ist jedoch nur ein Knoten vorhanden - gefolgt von 25 AwqlQueries aus der zuvor erzeugten AwqlQuery-Liste. Nach den 25 AwqlQueries wird wieder eine JOIN-Nachricht an einen zufälligen Knoten im Netzwerk übergeben. Der Quellcode 6.4) verdeutlicht die Vorgehensweise.

Listing 6.4 Kontrollklasse

```
[..]
int totalCnt = 0;
[..]

public boolean execute() {
 // Abbruchbedingung
 if(totalCnt < maxCnt) {
  if(totalCnt % 25 == 0) {
   // Jede 25. Nachricht ist ein JOIN
   join();
  } else {
   query();
  }
  totalCnt++;
  return false;
 } else {
  return true;
 }
}
```

Die Methoden *join()* und *query()* schicken QUERY- und JOIN-Nachrichten an einen zufälligen Knoten im Netzwerk .

6.3. Messinstrumente

Das wichtigste Messinstrument ist der *Monitor*, welcher während der Bearbeitung einer Awql-Query die Cache- und Serverzugriffe misst, die Anzahl der durchsuchten Cacheelemente, Cacheobjekte und Serverobjekte aufnimmt und auch die Anzahl der benötigten Pakete misst. Da eine REPLY-Nachricht immer die direkte Antwort auf eine QUERY-Nachricht ist und somit die Anzahl der beiden Nachrichten gleich ist, muss nur die Gesamtzahl der Nachrichten gezählt werden, eine Unterscheidung zwischen den Nachrichten ist nicht notwendig. Der Messmonitor nimmt somit folgende Werte auf: Anzahl Nachrichten, Anzahl Weiterleitungen (ROUTE), Anzahl durchsuchter Serverobjekte, Anzahl durchsuchter Cacheelemente, Anzahl durchsuchter Cacheobjekte, Größe des Netzwerkes, Knoten, welche an der Berechnung mitgewirkt haben, Anzahl Cachezugriffe, Anzahl Serverzugriffe, Anzahl gefundener Objekte im Cache, Anzahl gefundener Objekte im Server. Zum Schluss wird noch die Prozess-ID aufgenommen.
Als weiteres Werkzeug, welches jedoch keine Messungen aufnimmt, dient eine grafische Oberfläche, welche die Verteilung der Anfragegebiete sowie den aktuellen Zustand des Content Adressable Network zeigt. Abbildung 6.1 zeigt ein Beispiel, wie sich das Content Adressable Network nach ca. 1.500 Anfragen und knapp 60 Splittvorgängen entwickelt hat. Die roten Markieren sind die Anfragegebiete, es ist klar zu erkennen, dass die Fahrzeuge, welche die Anfragen stellen, sich langsam durch den Datenraum bewegen.

Als weiteres Messinstrument dient die Aufzeichnung der Splittvorgänge in einem Baum, welcher nachfolgend als CAN-Baum bezeichnet wird. Da die Anzahl der Weiterleitungen von QUERY-Anfragen ebenfalls die Berechnungsfunktion beeinflusst, soll eine direkte Abhängigkeit zwischen der Baumstruktur und der Anzahl der Weiterleitungen ermittelt werden.

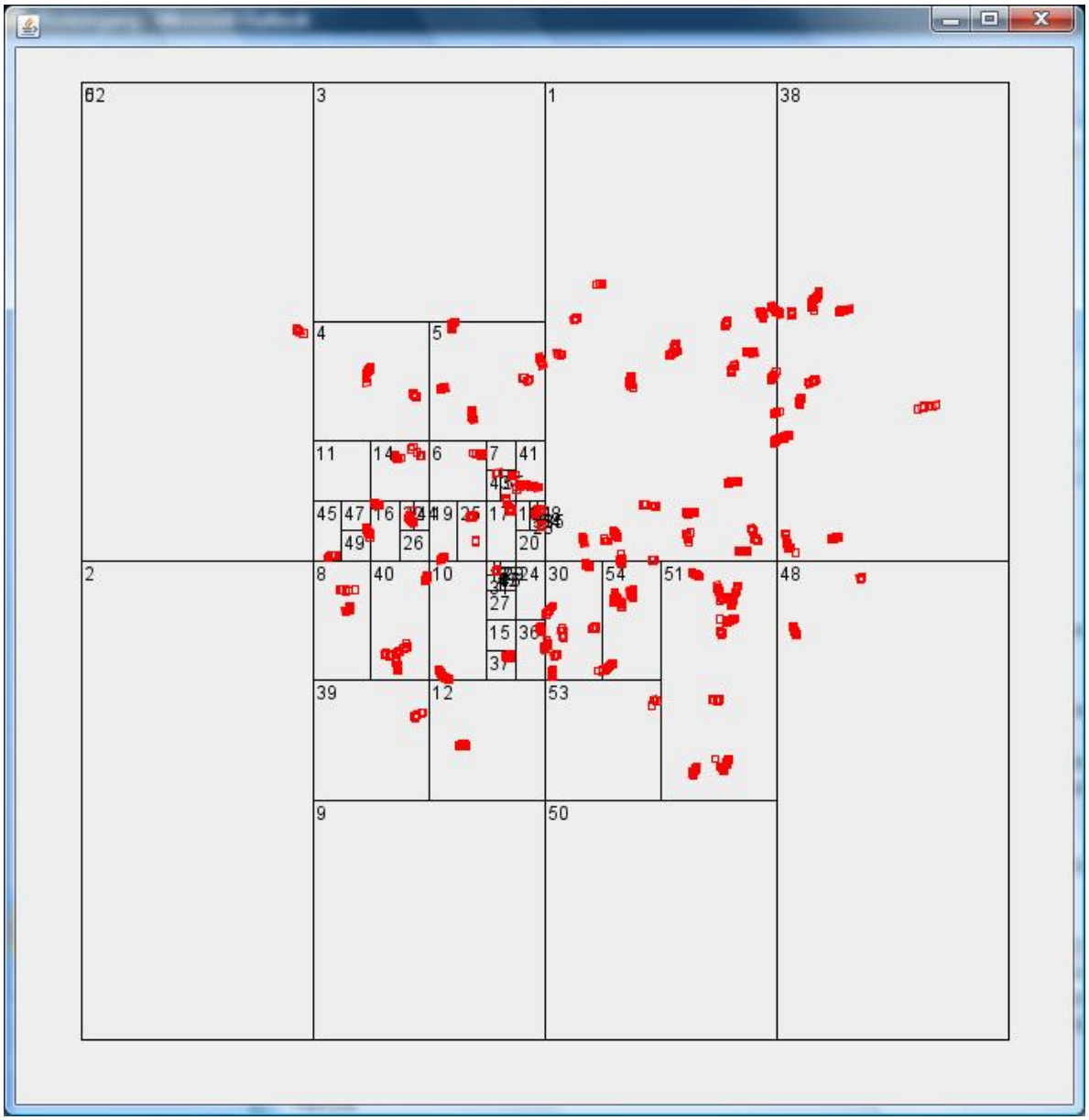


Abbildung 6.1.: Zustand im CAN

7. Messung und Analyse

Was du nicht messen kannst,
kannst du nicht lenken.

(Peter Drucker - US-Ökonom)

Primäres Ziel dieser Diplomarbeit ist es, dass die Berechnung der Lösungsmenge über den verteilten Cache effektiver ist als direkt über den Server. Der Serverzugriff bedarf lediglich einer Nachricht zum Server und einer weiteren für die Lösungsmenge. Jedoch ist der Rechenaufwand im Server höher als in einem Cache. Der verteilte Cache benötigt hingegen eine Reihe von Nachrichten, um die Berechnung einer AwqlQuery durchführen zu können. Dennoch kann es unumgänglich sein, dass zusätzlich der Server in die Berechnung mit einbezogen werden muss, wenn die Berechnung gar über mehrere Knoten geht und jeder Cache des Knotens keine Lösungsmenge generieren kann. So ist für jeden Knoten ein Serverzugriff notwendig. Daher ist es nötig, dass die Nachrichten zwischen den Caches und die Serverzugriffe minimal werden.

7.1. Definitionen

Für die Berechnung des Messergebnisses werden folgende Annahmen und Konventionen getroffen:

- *Serverzugriffe:* Der direkte Serverzugriff vom Benutzer aus und der Serverzugriff vom Cache aus sind gleichzusetzen. Es wird davon ausgegangen, dass der Server einen linearen Suchalgorithmus in Abhängigkeit der sich darin befindlichen Objekte verwendet. Somit ist der Aufwand für die Berechnung einer AwqlQuery: $T_s = c_s O_s$, wobei c_s eine Serverkonstante ist und O_s für die Anzahl der Objekte im Server.
- *Cachezugriffe:* Für den Aufwand eines Cachezugriffs wird die Annahme getroffen, dass die RestrictionOperatoren der Cacheelemente linear zur Anzahl der Cacheelemente durchsucht werden, die Objekte, welche einem Cacheelement angehören, werden ebenfalls in Linearzeit durchsucht. Da jedoch bei der Suche innerhalb eines Cacheelements nur noch ein kleiner Teil des RestrictionOperators überprüft werden muss, ist somit die Feinsuche innerhalb eines Cacheelements effizienter als im Server. Somit ergibt sich für den Aufwand eines Cachezugriffs: $T_c = c_{ce} O_e + c_{co} O_{eo}$, wobei sich die Konstante c_{ce} auf die Suche nach den Cacheelementen bezieht und c_{co} die Suchkonstante für die Suche nach Objekten innerhalb eines Cacheelements ist. Es gilt zudem: $c_{ce} > c_{co}$, da

die Suche nach den Cacheelementen aufwändiger ist als die Suche nach Cacheobjekten innerhalb eines Cacheelements. O_e ist die Anzahl der Cacheelemente im Cache, O_{eo} ist die Anzahl der Cacheobjekte innerhalb eines Cacheelements.

- *Nachrichten:* Das Versenden von Nachrichten bedarf immer dem gleichen Aufwand T_n. Für den Serverzugriff sind neben dem Aufwand T_s noch zwei weitere Nachrichten notwendig, eine für die AwqlQuery, die andere für die Lösungsmenge.
- *Bearbeitung von Nachrichten:* Unterschiedliche Nachrichten benötigen unterschiedliche Berechnungszeiten auf einem Knoten. Da eine QUERY-Nachricht nur an den Cache weitergegeben wird, ist keine zusätzliche Berechnung erforderlich. Als eher bearbeitungsintensiv werden REPLY-Nachrichten angesehen. Daher wird für die Berechnung angenommen, dass REPLY-Nachrichten den doppelten Berechnungsaufwand haben als alle anderen Nachrichten, mit Ausnahme der QUERY-Nachricht, welche einen zusätzlichen Berechnungsaufwand hat. Der Berechnungsaufwand wird mit T_b definiert. Da jedoch zu jeder QUERY-Nachricht eine REPLY-Nachricht erzeugt wird, ergibt sich im Mittel einen Berechnungsaufwand von T_n, was somit für alle Nachrichten gilt.

7.2. Messergebnisse

Das vorher vorgestellte erste Anwendungsszenario wurde mit Hilfe des Monitors gemessen.

7.2.1. Weiterleitungen

Als erstes Messergebnis wird die Anzahl der Weiterleitungen vom Anfrageknoten zum Initialknoten in Abhängigkeit der Netzwerkgröße vorgestellt. Jedoch stellt sich sehr schnell heraus, dass eine direkte Abhängigkeit zwischen der Netzwerkgröße und der Anzahl der Weiterleitungen sehr schwer herleitbar ist. Dies hängt damit zusammen, dass der CAN-Baum keine direkten Nachbarschaftsverhältnisse anzeigt. Versucht man, die Nachbarschaftsverhältnisse in den CAN-Baum einzutragen, so werden sofort Querverweise zwischen den Blättern im CAN-Baum sichtbar. Abbildung 7.1 zeigt bereits, dass bei einem kleinen Content Adressable Network scheinbar unkontrollierte Nachbarschaftsverhältnisse existieren.

Abbildung 7.2 zeigt, wie sich die Anzahl der Weiterleitungen im Verhältnis zur Netzwerkgröße verhält. Es ist zu erkennen, dass die Kurve zu Beginn schnell anwächst und danach verflacht, jedoch kein Maximum erreicht wird. Dies spricht für ein logarithmisches Anwachsen der Weiterleitungen.

Wird in einem Baum ein Blatt gesucht, so ist die durchschnittliche Pfadlänge:

$$d(T) = \frac{\sum_{l=1}^{h} (l \cdot L(n,l))}{n}$$

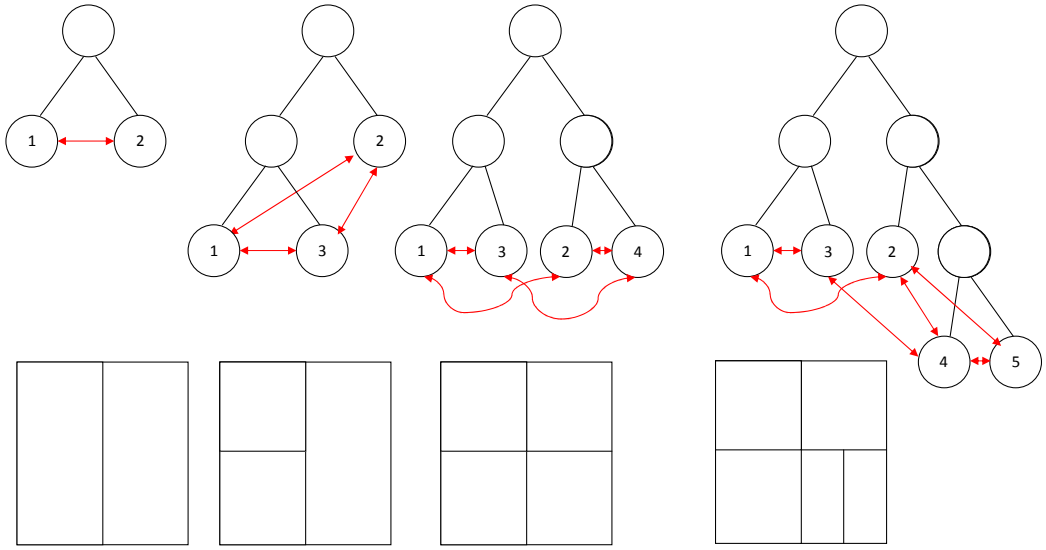


Abbildung 7.1.: Nachbarschaftsverhältnisse im CAN-Baum

wobei n für die Anzahl der Blätter im Baum steht und die Funktion $L(n, l)$ die Anzahl der Blätter zurückliefert, welche auf der Ebene l angesiedelt sind. h steht für die Höhe des Baumes und ist im Falle des Content Adressable Network davon abhängig, wie gleichverteilt die Splittungen sind. Für den Fall einer Gleichverteilung ist der daraus folgende Binärbaum vollständig und hat somit eine Höhe von

$$h = \log_2 n$$

Im Vergleich dazu zeigt der CAN-Baum eine wesentlich bessere durchschnittliche Suche, was dafür spricht, dass mit dem Content Adressable Network eine sehr effiziente Variante verwendet wird. Auch bei sehr großen Netzwerken bleibt die durchschnittliche Anzahl an Weiterleitungen weit unter der Kurve von $log_2 n$.

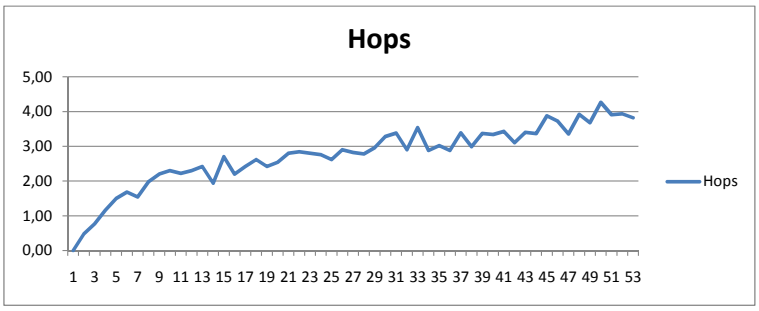


Abbildung 7.2.: Diagramm: Weiterleitungen pro Netzwerkgröße

7.2.2. Fülleffekt

Der Cache speichert zu den Cacheelemente noch das Gebiet, welches als bekannt eingestuft wird. Alle Objekte innerhalb dieses Gebietes sind somit dem Cache bekannt, der Server muss nicht einbezogen werden. Somit wird schon jetzt deutlich, dass eine direkte Abhängigkeit zwischen dem bekannten Gebiet und der Anzahl der Serveranfragen besteht. Für diese Testmessung wurden AwqlQueries erstellt, welche jedoch, mit Ausnahme des Anfragegebietes, identisch sind. Die relative Größe des Anfragegebietes zur Gesamtgebietsgröße wurde in jeder Versuchsreihe fest vorgegeben, wobei bei unterschiedlichen Anfragegebietsgrößen die durchschnittliche Gebietsgröße entscheidend ist. Das Diagramm in Abbildung 7.3 zeigt den Füllstatusverlauf für eine durchschnittliche Anfragegebietsgröße von 1,5% (oben), 3,5% (Mitte) und 5,5% (unten) jeweils in Abhängigkeit von der Anzahl gestellter Anfragen.

Somit ist zu erkennen, dass das bekannte Gebiet ein enormes Wachstum aufzeigt. Um das Wachstum mathematisch auszudrücken, ist es notwendig, zu zeigen, in wieweit die bekannte Gebietsgröße in Abhängigkeit der Anfragegebietsgröße zunimmt. Für alle Gebietsangaben werden relative Werte in Bezug auf die Gesamtgebietsgröße verwendet. Im optimalen Fall hat somit das Gebiet, welches im Cache bekannt ist, eine Größe von 1. Nimmt man nun an, dass das bekannte Gebiet $P1$ eine Größe von α besitzt und ein Anfragegebiet mit der Größe β hinzugefügt werden soll - und eine Gleichverteilung der Positionen der Anfragegebiete vorausgesetzt werden kann - so ist dies die Wahrscheinlichkeit, dass $\alpha \cdot \beta$ innerhalb des bekannten Gebietes und der Rest innerhalb des unbekannten Gebietes liegt. Somit wächst das bekannte Gebiet um diese Größe. Da eine direkte Abhängigkeit der neuen, bekannten Gebietsgröße von der alten, bekannten Gebietgröße vorhanden ist, ergibt sich somit folgende Rekursionsgleichung:

$$\alpha(n+1,\beta) = \alpha(n,\beta) + \beta - (\alpha(n,\beta) \cdot \beta)$$

wobei n für die Anzahl der Anfragen steht. Da in der Realiltät unterschiedliche Anfragen gestellt werden, ist eine Erweiterung der Formel notwendig. Diese Anfragen können ungleichmäßig gestellt werden, dies bedeutet, dass es Anfragen gibt, welche häufiger gestellt werden als andere. Daher wird noch eine Verteilungsvariable benötigt, welche das Verhältnis zwischen den Anfragetypen anzeigt. Existieren m-verschiedene Anfragetypen und liegt eine Gleichverteilung vor, so werden pro Anfragetyp insgesamt $\frac{n}{m}$ gestellt, wobei n für die Gesamtanzahl aller Anfragen steht. Um die unterschiedliche Anzahl von Anfragen pro Anfragetyp zu verdeutlichen, wird die Variable n_i eingeführt, wobei $i = \{1, 2, ...m\}$ gilt. Für die Anfrageverteilung wird die Variable a_i verwendet. Für beide Variablen gilt somit:

$$\sum_{i=1}^{m} a_i = 1$$

$$\sum_{i=1}^{m} n_i = n$$

Werden nun alle Variablen in die oben vorgestelle Gleichung eingesetzt, ergibt sich somit für die durchschnittlich bekannte Gebietsgröße:

$$A(n,\beta) = \sum_{i=1}^{m} (a_i \cdot \alpha(n_i,\beta))$$

Für den Fall, dass $m = 1$ und $\beta = 3,5\%$ ist, wird die Wachstumskurve in Abbildung 7.4 gezeigt.

Hierbei wird deutlich, dass zwar die Steigung mit den Messergebnissen übereinstimmt, die Kurve sich aber einem höheren Wert annähert als es die Messungen zeigen. Dies hängt damit zusammen, dass bei der zufälligen Erzeugung der Anfragegebiete die Eck- und Randbereiche wesentlich seltener „getroffen“ werden, als es in der theoretischen Gleichverteilung der Fall ist. Grund hierfür ist, dass für ein Polygon mindestens zwei Punkte in einem schmalen Bereich am Rand ermittelt werden müssen, was jedoch nach der Wahrscheinlichkeitsrechnung leider oft passiert. Bei der theoretischen Gleichverteilung wird dies jedoch außer Acht gelassen. Somit ist der praktische Füllstatus im Gebietszentrum fast vollständig abgedeckt, doch die Randbereiche weisen eine größere Anzahl von unbekannten Gebieten auf. In der Messung wurde bisher noch nicht bedacht, dass der Cache ein Limit hat und somit nur eine begrenze

Anzahl von Objekten aufnehmen kann. Wird dieses Maximum erreicht, ist das Löschen von Objekten mit Hilfe der Ersetzungsstrategien notwendig. In diesem Fall werden Flächen gekürzt oder gar ganz gelöscht. Daher ist es kaum möglich eine vollständige Abdeckung zwischen dem Gesamtgebiet und dem bekannten Gebiet zu erreichen. Daher wird ein neuer, für jeden Cache individuellen Grenzwert erzeugt, an welchen sich die Wachstumskurve annähert. Der Grenzwert wird nachfolgend mit der Variablen δ ausgedrückt.

7.2.3. Serverzugriff

Der Serverzugriff muss erfolgen, falls das Anfragegebiet nicht innerhalb des bereits bekannten Gebietes im Cache liegt. Daher ist eine direkte Abhängigkeit zwischen der Wahrscheinlichkeit eines Serverzugriffs und der bekannten Gebietsgröße nachzuweisen. Des Weiteren spielt auch die Größe des Anfragegebietes eine Rolle. Abbildung 7.5 zeigt, in wieviel Prozent der Fälle, in Abhängigkeit der bekannten Gebietsgröße, die Beantwortung der AwqlQueries direkt aus dem Cache erfolgt, so dass kein Serverzugriff nötig ist. Die Unterscheidung aller drei Diagramme liegt widerum in der Größe des Anfragegebietes, wobei das obere Diagramm eine durchschnittliche Anfragegebietsgröße von 1,5%, das mittlere Diagramm von 3,5% und das untere Diagramm von 5,5% besitzt.

Es ist zu erkennen, dass erst bei einem hohen, relativen bekannten Gebiet die Anzahl an Serverzugriffe spürbar zurückgeht. Die Wahrscheinlichkeitsfunktion, dass ein Anfragegebiet mit der relativen Größe p_q innerhalb des bekannten Gebietes innerhalb eines Caches mit der relativen Größe p_b liegt, wird nachfolgend mit $P(p_b, p_q)$ bezeichnet.

Je größer das bekannte Gebiet innerhalb eines Caches ist, desto höher ist die Wahrscheinlichkeit, dass der Server nicht in die Berechnung einbezogen werden muss. Die Messergebnisse zeigen, dass, wenn ca. 80% des Gebietes innerhalb eines Caches bekannt ist, der Server nur noch in der Hälfte der Fälle befragt werden muss.

7.3. Kostenfunktion

Nachdem die Testergebnisse analysiert und die verwendeten Variablen definiert worden sind, bedarf es nun einer Kostenfunktion, um den Aufwand zu beschreiben, welcher notwendig ist, um den verteilten, semantischen Cache zu verwenden. Die Kostenfunktion besteht aus vier Summanden:

$$T = T_c + T_s + T_n + T_b$$

In dieser Rechnung fehlen nur die Anzahl der Zugriffe auf den Cache und den Server sowie die Anzahl der Nachrichten, welche verschickt worden sind. Da jede Nachricht zur Folge hat, dass der Empfänger einer Nachricht eine Bearbeitungszeit benötigt, so können die hinteren beiden Summanden zusammengezogen werden. Für die weitere Erläuterung wird davon ausgegangen, dass für die Berechnung einer AwqlQuery insgesamt a Cachezugriffe und b Serverzugriffe nötig sind, wobei gilt, dass $a \geq b$ ist. Desweiteren werden insgesamt s Nachrichten verschickt, um die Kommunikation zwischen den Knoten im Netzwerk sicherzustellen. Somit ergibt sich als Zwischenlösung:

$$T = a \cdot T_c + b \cdot T_s + s \cdot (T_n + T_b)$$

Wie bereits vorgestellt, ist zwar für jeden Knoten, welcher an der Berechnung der AwqlQuery beteiligt ist, ein Cachezugriff notwendig, aber der Serverzugriff ist abhängig vom Zustand des Caches. Somit kann mit Hilfe der vorher vorgestellten Wahrscheinlichkeitsfunktion der Wert b in Abhängigkeit von a gesetzt werden.

$$T = a \cdot T_c + a \cdot P(p_b, p_q) \cdot T_s + s \cdot (T_n + T_b)$$

Der Wert p_b bezieht sich dabei auf die durchschnittliche, relative Größe des bekannten Gebietes innerhalb des Caches. Für die Gleichverteilung der Anfragen über den gesamten Datenraum ist die Lösung nun fertig, jedoch kann von einer Gleichverteilung nicht ausgegangen werden. Besonders für das Szenario, in dem mehrere Hotspots existieren, wird die Lösungsgleichung als unbrauchbar angesehen. Befindet sich ein Hotspot innerhalb des Gebietes eines Caches, so wächst ein Teilgebiet des Caches wesentlich schneller als der Rest. Dies ist dadurch zu erklären, dass bei der Wachstumsfunktion $A(n, \beta)$ das β eine relative Größenangabe ist. Wird das absolute Gebiet eines Caches verkleinert, die absolute Größe des Anfragegebietes jedoch konstant bleibt, so vergrößert sich die relative Größe des Anfragegebietes. Wie vorher gezeigt worden ist, ist somit ein schnelleres Wachstum zu verzeichnen. Es sind wesentlich weniger Anfragen notwendig, um das bekannte Gebiet innerhalb eines Caches zu füllen. Nun kann man jedoch das Gebiet innerhalb eines Caches in unterschiedliche Teilgebiete unterteilen, für welche eine unterschiedliche Anfragehäufigkeit zu verzeichnen ist, wie zum beispiel bei der Hotspot-Simulation. Für die nachfolgende Gleichung werden folgende Variablen neu definiert: r_i ist die relative Größe des i-ten Teilgebietes, $A_i(n, \beta)$ ist die Wachstumsfunktion innerhalb des i-ten Teilgebietes, k_i stellt das relative Anfrageverhältnis zwischen den Teilgebietes dar. Für die neuen Variablen gilt zudem:

$$\sum_{i=1}^{d} r_i = 1$$

$$\sum_{i=1}^{d} k_i = 1$$

wobei d für die Anzahl der Teilgebiete innerhalb des Cache steht. Für die neue, allgemeine Wachstumsfunktion ergibt sich:

$$B(n, \beta) = \sum_{i=1}^{d} \left(r_i \cdot A(n_i, \frac{\beta}{k_i} \right)$$

Falls im zugewiesenen Datenraum ein Teilgebiet vorhanden ist, welcher ein Hotspot darstellt und alle Anfragen, welche vom Cache bearbeitet werden innerhalb dieses Teilgebietes liegen, so sind alle anderen Teilgebiete nicht mehr Teil der Summe, da r_i den Wert 0 annimmt. Da sich das bekannte Gebiet im Hotspot wesentlich schneller vergrößert, wächst auch die durchschnittliche Gesamtgröße des bekannten Gebietes innerhalb eines Caches wesentlich größer. Im Extremfall ist die relative Größe des bekannten Gebietes des gesamten Caches gleich der relativen Größe des bekannten Gebietes eines Hotspots.

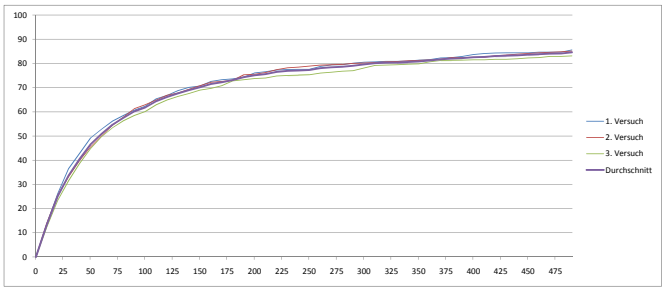
Da sich also in einem Cache, der einen Hotspot beinhaltet, die Wachstumsfunktion des bekannten Gebietes besser entwickelt als ohne einen Hotspot, ist auch die Wahrscheinlichkeit eines Serverzugriffs geringer, da diese von der Größe des bekannten Gebietes innerhalb eines Caches abhängt.
Je größer das Peer-to-Peer-Netzwerk ist, desto mehr Nachrichten sind notwendig, um eine Anfrage zu bearbeiten. Während die Anzahl der Weiterleitungen bereits vorgestellt worden sind, werden diese Weiterleitungsnachrichten nachfolgend nicht mehr einberechnet. Das Diagramm in Abbildung 7.6 zeigt, wie die Zunahme von QUERY-, REPLY-, QUERYBLOCK- und PROCESSFINISH-Nachrichten im Abhängigkeit von der Netzwerkgröße verhält.

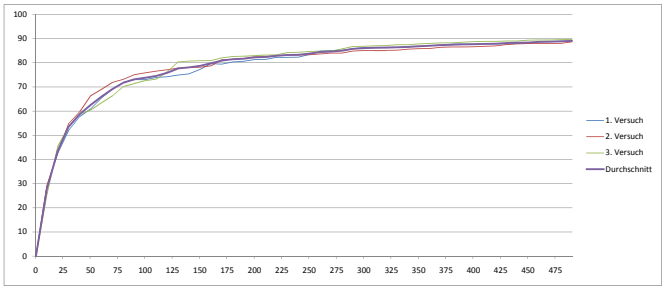
Es wird offensichtlich, dass die Anzahl der Nachrichten fast linear zur Netzwerkgröße wächst. Dies ist dadurch zu erklären, nach Nachrichten nur zwischen Nachbarknoten verschickt werden und die Anzahl der Nachbarn eines Knotens nur ganz langsam im Schnitt steigt. Wird gar eine Gleichverteilug angenommen, wobei sich das Content Adressable Network gleichmäßig aufbaut, wird gar ein fester Wert für die Anzahl der Nachbarn erreicht. Der Wachstum hängt somit nur indirekt von der Netzwerkgröße ab. Wenn man bedenkt, dass die Größe des Anfragegebietes im Verhältnis zum Gesamtgebiet sehr klein sind, so kommt man schnell zu dem Entschluss, im frühen Netzwerkstadium, also wenn sich noch wenige Knoten den gesamten Datenraum teilen, meist ein Knoten alleine das komplette Anfragegebiet in seinem zugewiesenen Datenraum abbilden kann. Da sich aber die durchschnittliche Größe des zugewiesenen Datenraums für jeden neuen Knoten im Netzwerk abnimmt, steigt die Wahrscheinlichkeit, dass ein Anfragegebiet von mehr als einem Knoten beantwortet werden muss. Da pro beteiligtem Knoten etwa die gleiche Anzahl an Nachrichten erzeugt werden, ist das lineare Wachstum der Kurve zu erklären.
Zum Schluss wurde noch untersucht, wieviele Objekte prozentual aus dem Cache geholt worden sind, ohne jedoch die Anzahl der Serverzugriffe zu berücksichtigen. Häufig wurde festgestellt, dass die Objekte, welche sich im Server befinden, nicht gleichverteilt sind. Dies kann zur Folge haben, dass der Cache gezwungen wird, aufgrund von unbekannten Gebieten in seinem Datenraum, den Server in die Berechnung mit einzubeziehen, aber dennoch alle Objekte vom Cache stammen, da in dem unbekannten Gebiet keine Objekte mehr vorhanden sind. Abbildung 7.7 zeigt, wieviel Prozent der Objekte vom Cache in Abhängigkeit der Anzahl der Anfragen geholt werden. Zu Grunde lag das Anwendungsszenario mit den Fahrzeugen, welche sich durch den Datenraum bewegen.

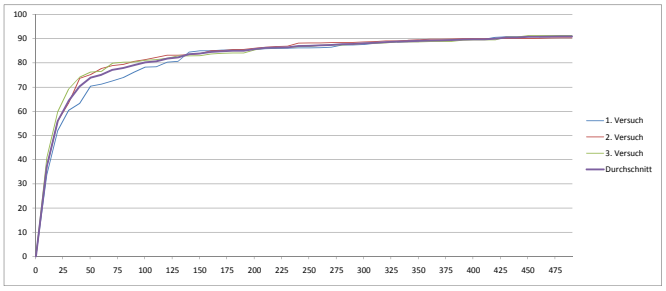
Dieses Ergebnis scheint zu überraschen, jedoch ist dies zum einen mit der Ungleichverteilung der Objekte im Datenraum zu erklären, zum anderen durch das Anwendungsszenario an sich. Im Anwendungsszenario bewegen sich die Fahrzeuge sehr langsam, damit wird ein großer Teil des bekannten Gebietes im Cache erneut abgefragt. Da sich insgesamt 100 Fahrzeuge im Anwendungssezarnio bewegen, ist ein sichtbares Ergebnis erst nach der 100. Abfrage zu erkennen. Da sich die Fahrzeuge von einem Gebiet mit hoher Objektdichte in ein Gebiet mit geringer Objektedichte bewegen können, ist es möglich, dass alle Objekte vom Cache geladen werden können, andererseits können auch Fahrzeuge von Gebiete mit niedriger Objektdichte in Gebiete mit hoher Objektdichte fahren, was dann zur Folge hat,

dass nur noch eine geringe Prozentzahl vom Cache geholt werden kann. was im Bereich der 500. Anfrage im Schaubild deutlich wird. Wird die gleiche Messung am Hotspot-Szenario durchgeführt, ist anfangs ein ähnliches Bild zu erkennen, jedoch werden keine negativen „Ausreißer“ vernommen, da sich die Anfragen immer um einen bestimmten Punkt herum sammeln. Das Anwendungsszenario verwendet zur Erzeugung der AwqlQueries die Normalverteilung nach Carl Friedrich Gauß [nor10], was dazu führt, dass bereits nach einer kleinen Anfragezahl der Kurve nahe dem Maximalwet von 100 Prozent nähert. Im Anwendungszenario wurden zwei HotSpots verwendet, daher tritt der beschriebene Effekt früher ein als bei dem Fahrzeug-Szenario. Abbildung 7.7 verdeutlicht diesen Effekt.

Wird die Gebietswachstumsfunktion $B(n, \beta)$ in der Wahrscheinlichkeitsfunktion $P(p_b, p_q)$ mit $p_b = B(n, \beta)$ verwendet, so kann bereits im Vorfeld einer Berechnung im Cache abgeschätzt werden, ob eine hohe Wahrscheinlichkeit für einen Treffer im Cache vorliegt, um so unnötigen Berechnungsaufwand bei hinreichend kleinen Wahrscheinlichkeiten zu vermeiden.







, 3,5% (Mitte), 5,5% (unten)

Abbildung 7.3.: Diagramm: Gebietswachstum in Abhängigkeit zur Anzahl der Anfragen. Anfragegrößen: 1,5% (oben

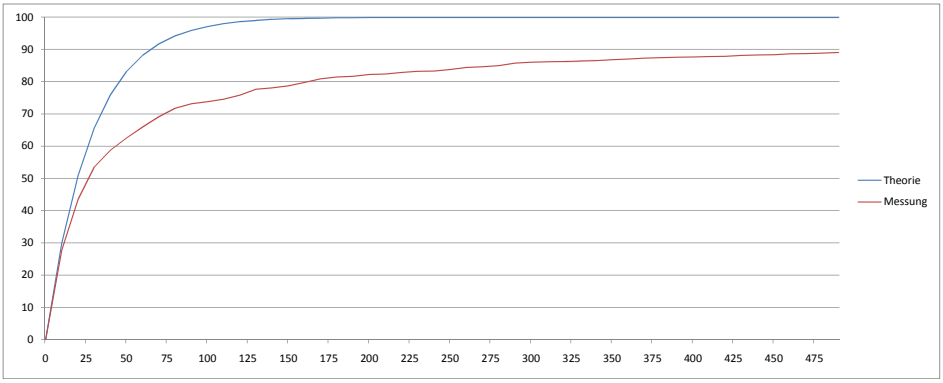


Abbildung 7.4.: Diagramm: Theoretisches Gebietswachstum, Vergleich Messung und Theorie in Abhängigkeit zur Anzahl der Anfragen mit fester Gebietsgröße

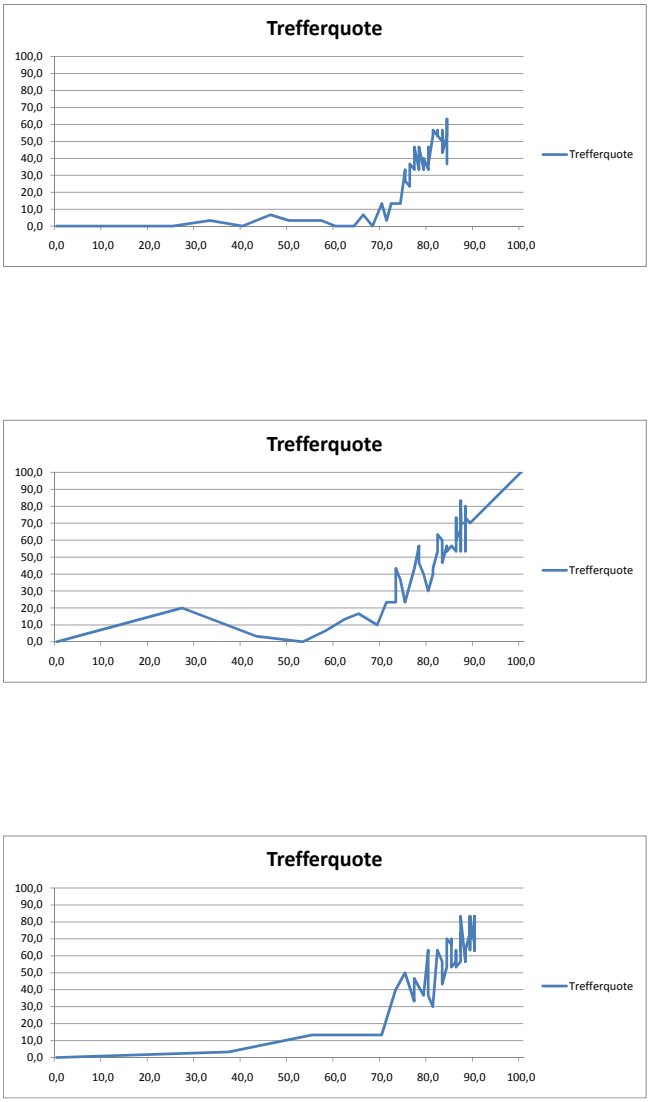


Abbildung 7.5.: Diagramm: Cachetreffer in Abhängigkeit der bekannten Gebietsgröße

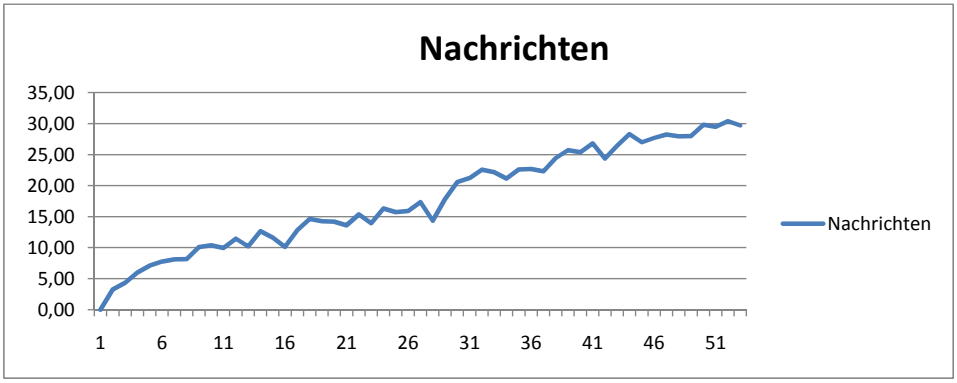


Abbildung 7.6.: Diagramm: Anzahl der Nachrichten in Abhängigkeit von der Netzwerkgröße

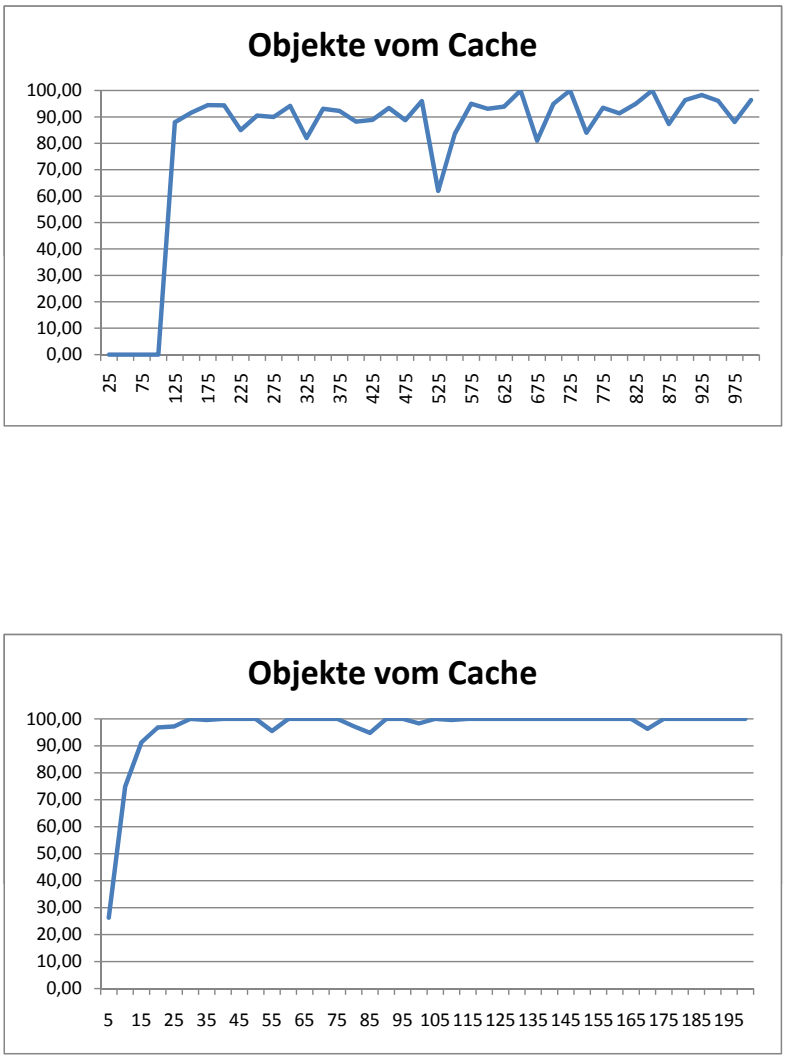


Abbildung 7.7.: Diagramm: Objekte vom Cache in Abhängigkeit zur Anzahl der Anfragen. oben: Fahrzeugszenario, unten: Hotspotszenarnio

8. Zusammenfassung und Ausblick

Wie anhand der Messergebnisse zu erkennen ist, muss in den meisten Fällen ein Mehraufwand betrieben werden, um die Anzahl der Serverzugriffe zu reduzieren. Ob sich der hier vorgestellte, verteilte, semantische Cache lohnt, ist abhängig vom Einsatzgebiet. Wenn eine Gleichverteilung der Anfragen vorliegt, verpufft fast die Effizienz des Caches, es sei denn, diese sind von ihrer maximalen Kapazität so bemessen, dass die Ersetzungsstrategien nicht zum Zuge kommen müssen und sich ein Großteil aller Objekte im Server irgendwann in den Caches befinden. Anders sieht es aus, wenn es mehrere Hotspots gibt. In diesem Fall ist die Knotendichte in und um einen Hotspot herum und somit auch die Dichte der Caches sehr viel höher als in anderen Gebieten. Eine Hohe Anzahl an Caches bedeutet somit auch, dass viele Objekte gespeichert werden können, was die Serverzugriffe in diesem Gebiet auf einen Minimalwert sinken lassen und somit der Cache seine Aufgabe erfüllt. Generell gilt, dass sich der Einsatz des Caches nur lohnt, falls gilt, dass sich die durchschnittlichen Kosten für eine Anfrage, welche über die Caches beantwortet werden, geringer sind, als die Kosten für einen direkten Serverzugriff. Dies gilt jedoch erst ab einer bestimmten Anzahl von Anfragen, da die Caches zu Beginn leer sind und daher die Anfragen immer erst vom Server beantwortet werden. Für das Anwendungsszenario mit den fahrenden Fahrzeugen kann ebenfalls als positives Beispiel betrachtet werden, aber nur dann, wenn die bereits gefahrenen Wege häufig wiederholt werden, was in der Realität der Fall sein dürfte. Zwar werden keine Hotspots generiert, jedoch aber gibt es dann einen Schwerpunkt auf verschiedenen Streckenabschnitten, welche auch als „kleine, langgezogene Hotspot“ aufgefasst werden können. Bedenkt man nun, dass sich das durchschnittlich bekannte Gebiet eines Caches durch eine Vielzahl unterschiedlicher Anfragen sehr negativ auswirken kann und mit Hinblick auf die Begrenzung der Cachekapazität die Gefahr besteht, dass nie ein ausreichend hoher relativer Wert für das bekannte Gebiet erreicht wird, um die Serverzugriffszahl signifikant sinken zu lassen. Daher wird im Zuge dieser Diplomarbeit zusammengefasst, dass der praktische Einsatz eines verteilten, semantischen Caches extrem von der eingesetzten Umgebung abhängt. Ein Ausblick in die Zukunft gestaltet sich recht schwierig. Im Zeitalter der Smartphones und tragbaren Tablecomputern wird auch der Bedarf an Informationen „on the way“ immer größer. Wenn man jedoch bedenkt, dass die mobile Kommunikationstechnik mittlerweile Geschwindigkeiten erreicht hat, welche bislang nur mit konventioneller Datenübertragungstechnik möglich war, stellt sich die Frage, worauf das Hauptaugenmerk gerichtet werden soll. Soll ein Server entlastet werden, so kann es schon sinnvoll sein, eine Cache-Overlay-Strukur zu verwenden, soll aber die durchschnittliche Anfragezeit verkürzt werden, ist dies nur dann sinnvoll, wenn dieses System zum einen sehr intensiv genutzt wird und sich zum

anderen die Anfragen nicht allzu sehr unterscheiden. Ob der verteilte, semantische Cache zum Einsatz kommt, wird bezweifelt, jedoch kann natürlich nicht ausgeschlossen werden, dass es auch effizientere Methoden gibt, um beispielsweise den Anfrageserver zu entlasten. Vielleicht können auch andere, neue Technologien verwendet werden, welche bisher nicht zur Verfügung standen.

A. Anhang

A.1. Begriffsdefinitionen

Anfrageknoten	Knoten, welcher eine Query vom Benutzer als erster entgegennimmt.
Anfragepunkt	Position, von welcher eine Query gestellt wird.
AWML	Argumented World Modeling Language: Modellbeschreibungssprache für Objekte innerhalb der Nexusplattform.
AWQL	Argumented World Query Language: Anfragesprache innerhalb der Nexusplattform.
Besitzer	Knoten, von welchem eine AwqlQuery weitergeleitet worden ist, jedoch ohne ROUTE-Nachrichten.
CAN	Content Adressable Network.
CAN-Baum	binäre Baumstruktur, welche die Splittvorgänge im Content Adressable Network aufzeichnet.
Erster Knoten	Anfrageknoten, welcher im Prozessstatus gespeichert wird.
Fahrzeug	Ein sich bewegendes Objekt, welches auf seiner festen Fahrstrecke regelmäßig Anfragen stellt.
Flooding	Versenden einer Nachricht an alle Nachbarn, bis eine Abbruchbedingung erfüllt ist.
Gauß-Krüger-Koordinatensystem	Kartesisches Koordinatensystem, mit dessen Hilfe ein geografisches Koordinatensystem berechenbar ist.
GPS	Global Positioning System, System zur Positionsbestimmung auf der Erdoberfläche.
Initialknoten	Knoten, welcher eine Query als erster aktiv bearbeitet.
Kontrollklasse	Die Klasse *CANRandomJoin]*, welche die Schnittstellen-Klasse *Control* der Peersim-Simulation verwendet.
Nexus	Plattform für Umgebungsmodelle, entwickelt von der Universität Stuttgart.
Server	Hintergrundspeicher, mit Hilfe der Klasse *SpaSe* simuliert.
UTM	Universal Transverse Mercator - kartesisches Koordinatensystem, mit dessen Hilfe ein geografisches Koordinatensystem mathematisch berechenbar ist.
WGS84	Geodätisches Referenzsystem, unter anderem Grundlage für GPS.

A.2. Mathematische Variablen

$A(n, \beta)$	Wachstumsfunktion für das bekannte Gebiet in einem Cache.
c_{ce}	Konstante für die Suche nach Elementen im Cache.
c_{co}	Konstante für die Suche nach Objekten innerhalb eines Cacheelements.
c_s	Konstante für die Suche nach Objekten im Server.
d	Anzahl der Teilgebiete eines Caches.
k_i	Relatives Anfrageverhältnis zwischen den Teilgebietes innerhalb eines Caches.
n	Summe aller Anfragen.
n_i	Anzahl der Anfragen des Typs i.
m	Anzahl der unterschiedlichen Anfragetypen.
O_e	Anzahl der Elemente im Cache.
O_{eo}	Anzahl der Objekte innerhalb eines Cacheobjekts.
O_s	Anzahl der Objekte im Server.
p_b	Relative Größe des bekannten Gebietes.
p_q	Relative Größe des Anfragegebietes.
$P(p_b, p_q)$	Wahrscheinlichkeitsfunktion, ob ein Gebiet mit der relativen Größe p_q innerhalb eines Gebietes mit der relativen Größe p_b liegt.
r_i	Relative Größe eines Teilgebietes innerhalb eines Caches.
T_b	Kostenfunktion für die Bearbeitung einer Nachricht.
T_c	Kostenfunktion für den Cachezugriff.
T_n	Kostenfunktion für das Versenden einer Nachricht.
T_s	Kostenfunktion für den Serverzugriff.
$\alpha(n, \beta)$	Relative Größe des bekannten Gebietes innerhalb eines Caches für einen Anfragetyp.
δ	Individueller Grenzwert des relativen, bekannten Gebietes in einem Cache, bei welchem die Ersetzungsstragie zum Tragen kommt.

A.3. Widmung

Diese Diplomarbeit widme ich hiermit meiner Mutter, Hildegard Richter (25.01.1957 - 16.06.2010), die während der Zeit der Erstellung meiner Arbeit im Alter von 53 Jahren verstorben ist.

Literaturverzeichnis

[BDG+04] M. Bauer, F. Dürr, J. Geiger, M. Grossmann, N. Hönle, J. Joswig, D. Nicklas, T. Schwarz. *Information Management and Exchange in the Nexus Platform*. University of Stuttgart Germany, 2004. (Zitiert auf den Seiten 7, 13, 14 und 15)

[gau10] http://de.wikipedia.org/wiki/Gauß-Krüger-Koordinatensystem. Internet, 2010. (Zitiert auf Seite 39)

[Gnu10] http://de.wikipedia.org/wiki/Gnutella. Internet, 2010. (Zitiert auf Seite 11)

[Gre02] M. Green. Napster Opens Pandoras Box: Examining How File-Sharing Services Threaten the Enforcement of Copyright on the Internet, 2002. (Zitiert auf Seite 11)

[HS04] H. Höpfner, E. Schallehn. Anfragegeneralisierung zur komprimierten Repräsentation von Indexen semantischer Caches auf mobilen Endgeräten. Technical report, Otto-von-Guericke-University Magdeburg Germany, 2004. (Zitiert auf Seite 21)

[jts] http://www.vividsolutions.com/jts/JTSHome.htm. Internet. (Zitiert auf Seite 38)

[KW08] M. Knoll, T. Weis. A P2P-Framework for Context-Based Information. Technical report, 2008. (Zitiert auf Seite 12)

[LP08] K. Lillis, E. Pitoura. Cooperative XPath Caching. Technical report, 2008. (Zitiert auf Seite 11)

[Nex] http://www.nexus.uni-stuttgart.de. Internet. (Zitiert auf Seite 13)

[nor10] http://de.wikipedia.org/wiki/Normalverteilung. Internet, 2010. (Zitiert auf Seite 74)

[ope04] http://www.openstreetmap.de. Internet, 2004. (Zitiert auf Seite 59)

[RD00] Q. Ren, M. H. Dunham. Using Semantic Caching to Manage Location Dependent Data in Mobile Computing. Technical report, Department of Computer Science and Engeneering Dallas TX USA, 2000. (Zitiert auf den Seiten 10, 11 und 12)

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. A Scalable Content Adressable Network, 2001. (Zitiert auf den Seiten 9, 11, 12, 17, 19 und 20)

[Sem03] Semantic Caching in Ontology-based Mediator Systems. Technical report, 2003. (Zitiert auf Seite 21)

[Spi08] O. Spinczyk. Virtueller Speicher, 2008. Lehrstuhl der Informatik, Technische Universität Dortmund. (Zitiert auf Seite 22)

[utm10] http://de.wikipedia.org/wiki/UTM-Koordinatensystem. Internet, 2010. (Zitiert auf Seite 39)

Alle URLs wurden zuletzt am 10.11.2010 geprüft.