



UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik

Implementierung und Evaluation eines semantischen Peer-to-Peer Netzwerks

Masterarbeit

im Rahmen des Studiengangs Informatik
zur Erlangung des Grades
Master of Science

von

Thomas Janson

Matrikelnummer: 6147966

vorgelegt bei

Prof. Dr. Friedhelm Meyer auf der Heide
und
Prof. Dr. Christian Schindelhauer

Paderborn, Juli 2009

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt habe. Alle aus fremden Quellen direkt oder indirekt übernommenen Formulierungen sind als solche gekennzeichnet und unter Literatur angegeben. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Ort, Datum

Unterschrift

Abstract

Peer-to-Peer Netzwerke werden benutzt um große Datenmengen und Ressourcen zwischen tausenden von Nutzern auszutauschen. Die meisten Netzwerke basieren auf verteilten Hashtabellen (DHT), wo sich die Suchmöglichkeit auf Punktanfragen beschränkt. Komplexere Suchen wie Bereichsanfragen stellen hier ein grundlegendes Problem dar. Mit 3nuts wird ein semantisches Peer-to-Peer Netzwerk vorgestellt, dass einen verteilten Suchbaum realisiert und dabei 3 Arten von Lokalität ermöglicht: Datenlokalität für Bereichsanfragen, Netzwerklokalität für schnelle Antwortzeiten sowie Interessenlokalität um gezielt bestimmte Anteile der Daten zu unterstützen. Zufallsnetzwerke bilden dabei mit ihrer robusten Struktur das Grundgerüst des Netzwerks. Darin wird die *Pointer-Push&Pull*-Operation zur Wartung und Aufrechterhaltung der zufälligen Struktur genutzt. Gewichtete verteilte Hashtabellen (*Distributed Heterogeneous Hash Tables*, DHHT) sorgen für die Lastbalancierung von Daten und Netzwerkverkehr auf die Peers. Das Netzwerk wurde prototypisch in der Programmiersprache *Java* implementiert und im Rahmen der Masterarbeit erweitert und verbessert. Anhand von Messungen im laufenden Betrieb des Netzwerks wird die Praxistauglichkeit untersucht.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Verwandte Arbeiten	3
1.2. Aufbau der Arbeit	4
2. Beschreibung des Netzwerks	5
2.1. Struktur	5
2.1.1. Gewichtete verteilte Hashtabellen	7
2.1.2. Zufallsnetzwerke	11
2.1.3. Verteilter Suchbaum	14
2.2. Algorithmen	23
2.2.1. Netzwerkerzeugung	23
2.2.2. Lookup	24
2.2.3. Suche	28
3. Implementierung	33
3.1. Grundgerüst	33
3.1.1. Communication	33
3.1.2. TaskScheduler	36
3.1.3. Zufallsnetzwerk-Peers	37
3.1.4. Gewichtete verteilte Hashtabellen	38
3.2. Struktur eines 3nuts Peer	39
3.2.1. Baumstruktur	39
3.2.2. Änderungen im Baum	42
3.3. Netzwerkoperationen	46
3.3.1. Lookup	47
3.3.2. Suche	48
4. Messungen	51
4.1. Grad	53
4.2. Entstehungsprozess des Netzwerks	57
4.3. Lookup	60
4.4. Netzwerklokalität	62
4.5. Lastverteilung	67
4.6. Robustheit des Netzwerks	71

Inhaltsverzeichnis

5. Zusammenfassung und Ausblick	75
Literaturverzeichnis	79
Glossar	83
A. Screenshots der Implementierung	85
B. Messparameter	89

Einleitung

In dieser Masterarbeit wird die Implementierung und Evaluation des semantischen Peer-to-Peer Netzwerks 3nuts vorgestellt. Im Rahmen des DELIS EU-Forschungsprojekts wurde bereits ein Prototyp des Netzwerks von mir als studentischer Hilfskraft implementiert. Dieser Prototyp dient als Grundlage für diese Arbeit.

Peer-to-Peer (P2P) Netzwerke werden benutzt um große Datenmengen und Ressourcen von tausenden von Nutzern gemeinsam zu nutzen. Jeder Peer bringt einen Teil der Daten oder Ressourcen ins Netzwerk ein, die über IDs identifiziert werden. In einer Lookup-Funktion können dann anhand der IDs die Peers mit den Daten und Ressourcen ermittelt werden.

In einem Client-Server-System würde man hierfür auf einem Server eine Lookup-Tabelle erstellen, die dann für jede ID einen Eintrag mit den Adressen der Peers mit den Daten enthält. In einem reinen P2P-Netzwerk hingegen wird dies verteilt gelöst und jeder Peer übernimmt einen Teil der Verwaltung. So wird jeder ID mindestens ein Peer zugeteilt, der dann die Verwaltung dieser ID übernimmt. Die Peers werden dann in einer logischen Netzwerkstruktur (strukturiertes Overlay) miteinander verbunden, die auf dem physikalischen Netzwerk (z.B. Internet) aufbaut. Diese ist dann so angelegt, dass man in der Lookup-Funktion in meist $\mathcal{O}(\log n)$ Hops (Pfadlänge im Netzwerk) zu dem Peer gelangt, der für eine ID zuständig ist und die Peers mit den Daten der ID kennt.

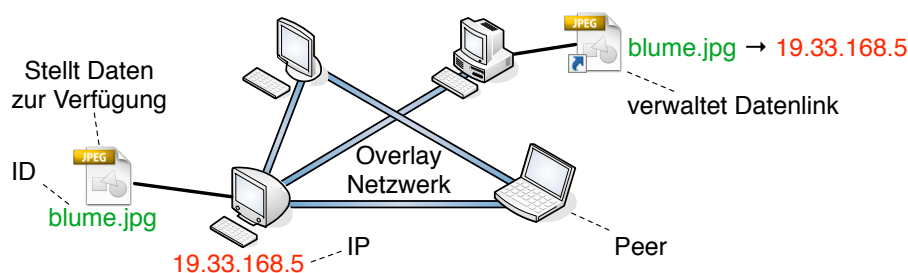


Abbildung 1.1.: Beispiel für Datenverwaltung in strukturiertem P2P-Netzwerk

P2P-Netzwerke bieten einige Vorteile gegenüber etwa Client-Server-Architekturen.

1. Einleitung

Da jeder Peer, der den Lookup-Dienst des Netzwerks nutzt auch gleichzeitig einen Teil dieses Dienstes anbietet, kann ein P2P-Netzwerk mit einer beliebigen Anzahl Nutzern skalierbar betrieben werden, ohne dass der Dienst bei zu vielen Nutzern überlastet wäre. Hierbei gibt es 2 Faktoren: die Anzahl Netzwerkverbindungen pro Peer und die Laufzeit der Lookup-Funktion. Beides steigt hier typischerweise nur logarithmisch zur Anzahl Peers im Netzwerk an. Ein weiterer Vorteil liegt in der Dynamik, Robustheit und Selbstorganisation der Netzwerke: fällt ein Teil der Peers aus, so übernehmen die anderen Peers deren Aufgaben mit. Werden neue Peers hinzugefügt, so wird ein Teil der Last an diese Peers abgetreten.

Ein Großteil der Peer-to-Peer Netzwerke basieren auf verteilten Hashtabellen (*Distributed Hash Tables*, DHT). Beispiele hierfür sind CAN [KS01], Chord [SMK⁺01], Pastry [RD01], Tapestry [HHK⁺02], Viceroy [MNR02], Distance-Halving [NW02] oder Koorde [KK03]. Dabei werden die IDs der Daten über eine Hashfunktion den Peers zugewiesen. Das Hashing sorgt hier für eine möglichst zufällige Aufteilung der IDs auf die Peers, so dass jeder Peer gleichmäßig viele IDs zugewiesen bekommt. Allerdings wird durch den Einsatz einer Hashfunktion die Ordnung der IDs zerstört, die nicht mehr mit der Ordnung der Hashwerte übereinstimmt. Suchschlüssel wie z.B. 'IPv4' und 'IPv6', die ursprünglich in einer z.B. lexikographischen Ordnung dicht beisammen liegen, können nach Eingabe in die Hashfunktion ganz verschiedene Hashwerte haben und damit an ganz verschiedenen Orten im Netzwerk liegen. So kann hier nicht einfach nach einem allgemeineren Suchmuster 'IP' gesucht werden und man kriegt 'IPv4' und 'IPv6' zurück ohne dabei die vollständige Hashtabelle durchsuchen zu müssen.

Deshalb gibt es Peer-to-Peer Netzwerke mit semantischer Speicherung, wo die Ordnung der Daten in der Verwaltung beibehalten wird. So verwaltet hier jeder Peer aus allen geordneten IDs einen zusammenhängenden Bereich. Dies ermöglicht dann effiziente Bereichsanfragen also die Suche nach allen Elementen in einem eingeschränkten Bereich (z.B. alle Elemente, die mit 'IP' beginnen). 3nuts ist ein solches Netzwerk. Hier können die Daten oder Ressourcen in einem Suchbaum verteilt verwaltet werden. Ein Suchbaum ist eine auf Bäumen basierende Datenstruktur, in der man in einer geordneten Menge von Elementen effizient suchen kann. Jede ID eines Elements ist dabei ein Knoten in diesem Baum, der durch den Pfad von der Wurzel identifiziert wird. Ohne den Einsatz einer Hashfunktion, die die Daten möglichst zufällig anordnet, sind die Daten im Suchbaum in ihrer ursprünglichen Anordnung meist ungleichmäßig verteilt. D.h. es gibt Teilbäume mit viel mehr Daten als andere. Damit hier jedem Peer für eine gute Lastbalancierung gleichmäßig viele IDs aus dem Baum zugeordnet werden, werden die Daten entsprechend der Datenverteilung auf den Baum aufgeteilt. So sind dann mehr Peers für die Verwaltung von Teilbäumen mit mehr Daten zuständig. Dazu werden in 3nuts heterogene Hashtabellen (*Distributed Heterogenous Hash Tables*, DHHT) [SS05] benutzt. Zu den Eigenschaften des 3nuts Peer-to-Peer Netzwerks zählen:

- Die Daten werden in 3nuts in einem verteilten Suchbaum verwaltet. In diesem Baum wird die Ordnung der Daten beibehalten.
- Punktanfragen und Bereichsanfragen können in einem Netzwerk mit n Peers mit

hoher Wahrscheinlichkeit in $\mathcal{O}(\log n)$ Hops durchgeführt werden.

- Für einen Hop in der Lookup-Funktion stehen meist mehrere Peers zur Auswahl. Mit der *Netzwerklokalität* von 3nuts wird hier immer der Peer mit der geringsten Latenz gewählt, um die Laufzeit der Funktion zu minimieren.
- Peers können in einer *Interessenlokalität* freiwillig die Verwaltung von Pfaden im Baum übernehmen etwa für die Verwaltung eigener Daten in diesem Pfad oder einen schnelleren Zugriff auf den Pfad.

1.1. Verwandte Arbeiten

Pastry [RD01] ist das erste Peer-to-Peer Netzwerk, dass in der Routingtabelle Netzwerklokalität berücksichtigt. Hier stehen für die Einträge der Routingtabelle ebenso wie in 3nuts eine Menge Peers zur Auswahl. Dabei werden mit einer angenommenen Nachbarschafts-Metrik die nächsten bestimmt. Die Metrik beruht auf Werten wie den IP Routing Hops oder der geographischen Distanz. Zum Aufbau führt ein neuer Peer das Routing von einem im Netzwerk sehr nahen Peer zu seiner eigenen Position im ID-Raum durch und baut seine Routingtabelle durch Einträge von Peers auf diesem Pfad auf. Da im bestehenden Netzwerk das Routing bereits auf die Lokalität optimiert ist, sind so die Einträge für den neuen Peer ebenfalls nicht schlecht. In einer zweiten Phase kontaktiert er dann die Peers aus seiner initialen Routing-Tabelle und versucht mit deren Einträgen seine eigenen zu optimieren. Hierzu führt jeder Peer noch zusätzlich eine Nachbarschaftstabelle im Sinne der Metrik.

Es gibt einige semantische Overlay-Netzwerke wie *Prefix Hash Tree* [RRHS04] oder *Distributed Segment Tree* [ZSLS06], die in einem DHT einen Binärbaum verwalten. Die einzelnen Baumknoten werden verteilt auf dem DHT mit dem Pfad als Schlüssel abgelegt, womit eine gleichmäßige Lastbalancierung durch den DHT gegeben ist. Der Binärbaum erhält den Datenzusammenhang und ermöglicht so Bereichsanfragen, wobei das Routing im Baum durch Lookups im DHT darunter realisiert ist.

Dann gibt es natürlich auch zu 3nuts ähnliche Netzwerke wie P-Grid [ACMD⁺03], Baton [JOV05], Mercury [BAS04], Cerco [Rie] oder DPTree [LLS06], die aktive Lastbalancierung betreiben. DPTree etwa verwaltet einen binären Trie. Dabei verwaltet jeder Peer einen Teil im ID-Raum nebeneinander liegender Blätter mit Daten darauf und alle Baumknoten darüber bis zur Wurzel. Fürs Routing wird der ID-Raum des Baums in einem Skip-Graph basierten Overlay-Netzwerk umgesetzt. Zur Lastbalancierung wird mit einem *Last-Wavelet* weniger belastete Peers im Netzwerk ermittelt. Ein Last-Wavelet komprimiert dazu die Last eines gesamten Teilbaums eines Baumzweigs zu einer durchschnittlichen Last. So weiß jeder Peer in jeder Tiefe des Baums welche Last jeweils in den beiden Baumzweigen ist und ob er überlastet ist. Zur Lastbalancierung muss ein Teil der Last eines überlasteten Peers an einen weniger belasteten Peer abgegeben werden. Dazu gibt es zwei Techniken: zum einen kann der wenig belastete Peer ein Leave-Rejoin zur Position des überlasteten Peers durchführen und sich so die Last teilen. Oder aber der überlastete Peer gibt einen Teil seiner Daten an seinen Nachbarn

1. Einleitung

im ID-Raum ab. Kaskadierend gibt der Nachbar wiederum einen Teil seiner Daten zum nächsten Nachbarn ab solange bis ein Teil der Daten bis zum wenig belasteten Peer über diese Kette "durchgerieselt" ist.

P-Grid [ACMD⁺03] hat ein binäres Trie-strukturiertes Overlay-Netzwerk. Jeder Peer hat in diesem Binärbaum einen Binärstring als Pfad und verwaltet alle Daten, die den gleichen Präfix im Binärstring ihres Schüssels haben. Mehrere Peers können den gleichen Pfad haben um die Daten redundant zu verwalten. Für das Routing hat ein Peer für jedes Bit in seinem Binärstring einen Link auf einen Peer in dem jeweils komplementären Bit. Erhält ein Peer dann eine Anfrage, wobei im Suchpfad das i -te Bit mit dem Pfad des Peers nicht übereinstimmt, so kann er die Anfrage zu einem Peer in dem komplementären Bit weiterleiten, bei dem mindestens ein Bit mehr übereinstimmt. Dies wird dann soweit rekursiv fortgesetzt bis der gesamte Pfad übereinstimmt und die Daten gefunden wurden. Jeder Link in der Netzwerkstruktur muss für eine beweisbar logarithmische Routingzeit ähnlich wie bei 3nuts rein zufällig aus der Menge möglicher Peers gewählt sein. Zur Lastbalancierung (siehe [ADH05]) führen die Peers beim Erstellungsprozess des Netzwerks (*Bootstrapping*) zufällige Interaktionen aus, in denen sich die Peers in Absprache für einen Baumzweig entscheiden. Dabei werden die Peers entsprechend der inhomogenen Datenverteilung gleichmäßig auf die Teilbäume aufgeteilt (*Adaptive eager partitioning*, AEP). Im laufenden Betrieb des Netzwerks legt dann jeder Peer für jedes Level seines eigenen Pfads eine Statistik darüber an, wie viele Peers pro Daten (Replikationsfaktor) sich in beiden Baumzweigen 0 und 1 befinden. Um dabei Ungleichgewichte auszugleichen wechseln Peers mit geringer Auslastung ihren Pfad in überlastete Gebiete des Baums. Dabei wechseln die Peers ihren Pfad unabhängig voneinander mit einer gewissen Wahrscheinlichkeit, die mit wachsendem Ungleichgewicht steigt. Damit soll verhindert werden, dass zu viele Peers in einen überlasteten Pfad wandern, die darauf hin als unterforderte Peers wieder abwandern müssen und das Ganze beginnt zu schwingen.

1.2. Aufbau der Arbeit

Diese Masterarbeit gliedert sich in 3 Teile: Theorie, Implementierung und Messungen. Zuerst wird in Kapitel 2 das 3nuts Netzwerk theoretisch beschrieben und analysiert. Dies beinhaltet die Netzwerkstruktur sowie die Algorithmen für den Lookup und die Suche im Suchbaum. Für den Lookup wird dann auch eine Laufzeitanalyse angestellt. In Kapitel 3 folgt dann eine kurze Beschreibung der Implementierung des 3nuts Netzwerks. Im Vordergrund steht hier, wie das Netzwerkmodell für die praktische Anwendung verfeinert und konkretisiert werden musste, um beispielsweise mit dynamischen Netzwerken gut umgehen zu können. Zuletzt werden in Kapitel 4 Messergebnisse präsentiert, die im implementierten 3nuts Netzwerk durchgeführt wurden. Untersucht wurden hier u.a. die Performanz des Lookups, die Lastverteilung auf die Peers, sowie die Robustheit des Netzwerks beim Ausfall von Peers. Abschließend werden im Anhang A Screenshots gezeigt, die anhand der GUI die Möglichkeiten der Software zur Erstellung, Ansicht und Messung eines 3nuts Netzwerks veranschaulichen.

Beschreibung des Netzwerks

Dieses Kapitel behandelt theoretisch das semantische Peer-to-Peer Netzwerk 3nuts. Zunächst wird in Unterkapitel 2.1 die Struktur des Netzwerk beschrieben die speziell auf die Lookup- bzw. Suchalgorithmen zugeschnitten sind, die im folgenden Abschnitt 2.2 beschrieben werden.

2.1. Struktur

Die Daten oder Ressourcen können in 3nuts in einem Suchbaum strukturiert werden. Bei dem Baum ist der Baumgrad und die Benennung der Baumknoten grundsätzlich frei wählbar. Das Beispiel in Abbildung 2.1 illustriert wie die Daten z.B. in Bücher, Filme oder Musik kategorisiert werden können. Die Strukturierung muss dabei vom Benutzer selbst angegeben werden und kann dann später genutzt werden um Suchoperationen eingeschränkt auf diese Kategorien ausführen zu können. Die Daten in

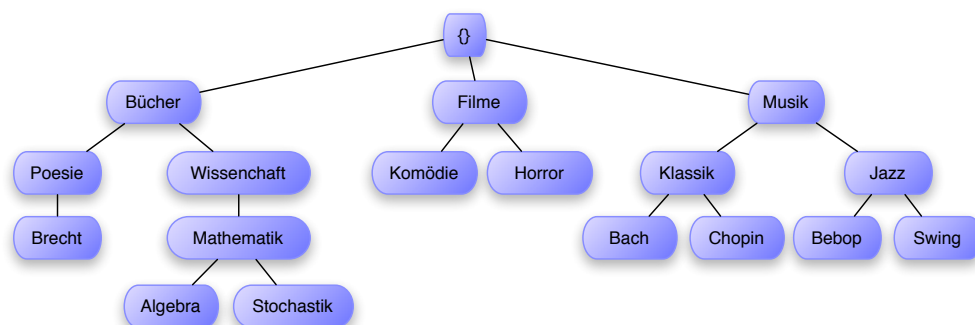


Abbildung 2.1.: *Beispiel für semantischen Datenbaum*

solch einem Baum sind höchst wahrscheinlich nicht gleichverteilt. So könnte im Beispiel die Rubrik 'Bücher' mehr Aufmerksamkeit erlangen und ein größeres Datenvolumen enthalten. Um die Datenlast nun gleichmäßig auf die Peers zu verteilen ist es daher nicht ausreichend die Peers einfach gleichmäßig auf den Schlüsselintervall zu verteilen.

2. Beschreibung des Netzwerks

Der Baum wird entsprechend der Datenmenge gewichtet. So erhält jeder Baumknoten (Blätter und innere Knoten) ein einzelnes Gewicht, dass die Datenmenge, Netzwerklast oder ähnliche Faktoren widerspiegeln kann. Diese Gewichte werden dann für einzelne Baumzweige aufaddiert, so dass sich ein gesamtheitliches Gewicht für die Zweige rekursiv ergibt (z.B. besteht das Gewicht des Zweigs Jazz aus Bebop und Swing). Die Peers teilen sich dann von der Wurzel an so auf den Baum auf, dass das Verhältnis der Gewichte in den einzelnen Zweigen dem Verhältnis der der Peers in den einzelnen Zweigen entspricht. Dazu wählt jeder Peer einen Pfad von der Wurzel an so tief im Datenbaum bis kein anderer Peer mehr den gleichen Pfad hat oder der Pfad in einem Blatt endet. Für die Wahl der Pfade wird *heterogenes verteiltes Hashing* benutzt (Abschnitt 2.1.1), womit Zweigen mit mehr Gewicht mehr Peers zugewiesen werden. Jeder Peer

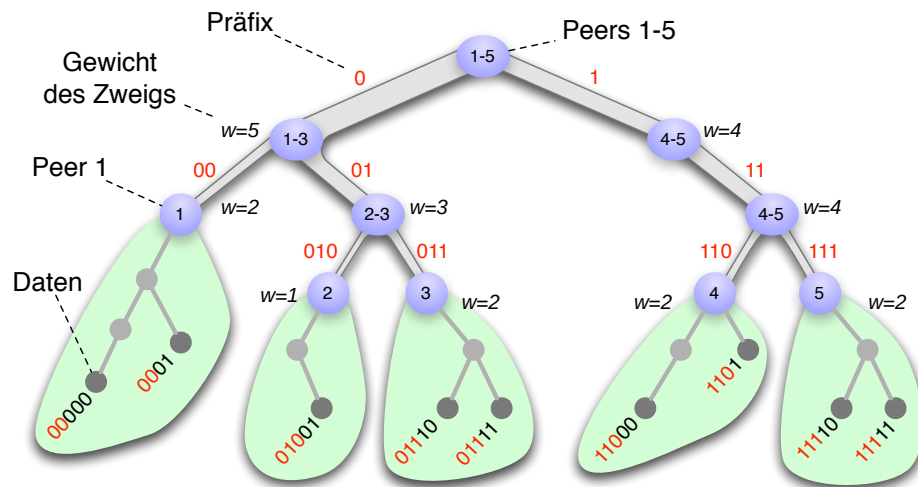


Abbildung 2.2.: Idee für Datenaufteilung in Binärbaum mit 5 Peers

verwaltet dann alle Daten, die den gleichen Präfix seines Pfads haben (z.B. in Abb. 2.2 Peer 1 mit Präfix '00'). Falls mal der Pfad mehrerer Peers so tief wie ein Datenpfad ist (z.B. Daten in Pfad '01' mit Peers 2 und 3), dann wird aus diesen Peers mit Hilfe von DHHT ein eindeutiger Peer zur Datenverwaltung bestimmt. Eine perfekte Aufteilung der Daten auf die Peers wird dabei allerdings nicht zwingend erreicht werden können. In Abbildung 2.2 befindet sich beispielsweise das Gewicht von 3 Daten im Präfix '01' welches hier unmöglich gleichmäßig auf die beiden Peers {2, 3} verteilt werden kann. So ergibt sich immer eine gewisse Fragmentierung, die umso größer ist je kleiner die Anzahl Peers ist. Des Weiteren können Baumzweige mit verhältnismäßig extrem kleinen Gewichten nicht auf eine in Relation stehende Anzahl von Peers abgebildet werden. Denn die minimale Anzahl an Peers in einem Baumzweig beträgt 1. Und sind z.B. n Peers auf 2 Baumzweige aufzuteilen, und das Gewicht des einen Zweigs liegt unter dem Anteil $1/n$, dann wird dieser Zweig mit minimal einem Peer überrepräsentiert. In den folgenden Abschnitten werden zunächst das gewichtete Hashing (2.1.1) sowie Zufallsnetzwerke (2.1.2) behandelt auf die dann der verteilte Suchbaum (2.1.3) aufbaut.

2.1.1. Gewichtete verteilte Hashtabellen

Gewichtete verteilte Hashtabellen (*Distributed Heterogeneous Hash Tables*, DHHT) veröffentlicht in [SS05] werden in 3nuts zur Lastbalancierung verwendet. Sie bieten hier eine sehr einfache Möglichkeit der Lastbalancierung, da jeder Peer alleine für sich selbst die Entscheidung treffen kann, welchen Teil des gewichteten Baums er übernimmt, in der Summe aber trotzdem die Peers gleichmäßig auf die Daten verteilt werden. Es müssen hier also keine Absprachen zwischen den Peers getroffen werden oder gar das eigene Verhalten von dem der anderen Peers abhängig gemacht werden um eine gleichmäßige Lastbalancierung zu erreichen. Wie in Abschnitt 1.1 dargestellt, ist dies in anderen Netzwerken wie z.B. P-Grid nicht der Fall.

Im folgenden wird zunächst das Prinzip der gewichteten verteilten Hashtabellen erläutert und darauf hin wie es in 3nuts angewendet wird. Beim gewichteten Hashing geht es darum eine Menge Datenelemente $X = \{x_1, \dots, x_n\}$ einer Menge Knoten $V = \{v_1, \dots, v_n\}$ eindeutig zuzuordnen. Dabei hat jeder Knoten v_i die Kapazität $w_i \in \mathbb{R}^+$ (Gewicht). Die Gesamtkapazität beträgt damit $W = \sum_{i \in V} w_i$. Das Ziel ist hierbei die Datenelemente mit Wahrscheinlichkeit w_i/W an Knoten v_i zuzuweisen, damit die Anzahl von Daten, die auf den Knoten v_i fallen, abhängig von seiner Kapazität ist. So sollen Knoten mit einer höheren Kapazität verhältnismäßig mehr Daten erhalten. Das Schema der Datenplatzierung auf die Knoten verläuft dabei wie folgt. Zunächst werden alle Datenelemente und Knoten auf den kontinuierlichen Bereich $M = [0, 1)$ mit Hilfe einer Hashfunktion abgebildet. Dabei wird Datenelement x_i abgebildet auf M mit $r_i = h(x_i)$ und Knoten v_i mit $s_i = h(v_i)$. Um nun zu ermitteln auf welchen Knoten ein Datenelement abgebildet werden soll, wird für ein Datenelement mit Hashwert r für jeden möglichen Knoten (*Host*) mit Hashwert s_i und Kapazität w_i eine Höhe berechnet:

$$L_w(r, s_i) := \frac{-\ln(1 - (r - s_i) \bmod 1)}{w_i}$$

Die Höhenwerte der einzelnen Knoten i für ein Datenelement werden dann verglichen und das Datenelement wird dem Knoten mit der geringsten Höhe zugewiesen:

$$H(r) := \min_i \left(\frac{-\ln(1 - (r - s_i) \bmod 1)}{w_i} \right)$$

Abbildung 2.3 zeigt ein Beispiel mit zwei Knoten $V = \{v_1, v_2\}$ auf denen zwei Datenelemente $X = \{x_1, x_2\}$ zu platzieren sind. In dem Graphen sind die Höhen der beiden Knoten für den gesamten Intervall M dargestellt. Das Gewicht w_1 von Knoten v_1 (blau) ist im Beispiel etwa halb so groß wie das Gewicht von Knoten v_2 (rot), so dass die Gewichtsfunktion entsprechend flacher verläuft und der Knoten so mehr Datenelemente zugeordnet bekommt. In dem Balken unter dem Graphen ist die Minimumfunktion farblich aufgetragen, in welchem Bereich des Intervalls welcher Knoten also ein Datenelement zugewiesen bekommt. So wird Datenelement x_1 mit Hashwert r_1 an Knoten v_1 zugewiesen, und Datenelement x_2 an Knoten v_2 .

Die Abbildung der Knoten auf den Intervall M erfolgt mit einer Hashfunktion zufällig gleichverteilt. Ist hierbei die Anzahl der Knoten klein, so kann die Aufteilung des

2. Beschreibung des Netzwerks

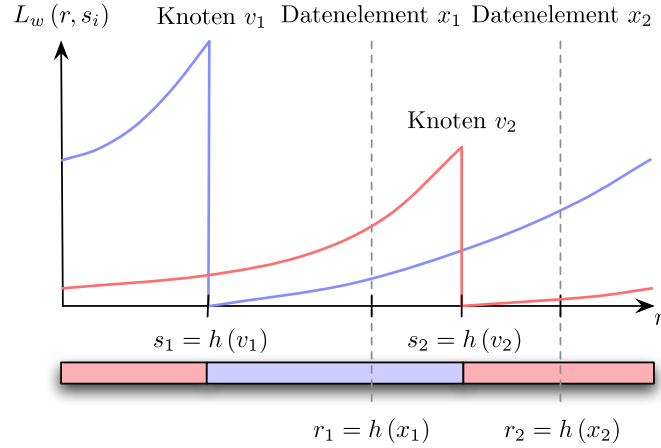


Abbildung 2.3.: Beispiel für gewichtete verteilte Hashtabelle

Intervalls auf die Knoten doch stark variieren. Man stelle sich vor, der Hashwert s_2 in Abbildung 2.3 wäre knapp größer als s_1 . Dann würde Knoten s_2 so gut wie im vollständigen Intervall dominieren. Um diesen Effekt zu vermeiden, wird *Double Hashing* benutzt. Hierzu wird der Hashwert r des Datenelements x mit dem Hashwert s_i jedes Knotens v_i konkateniert zu $pos_i := r \circ s_i$. Dieser Wert wird dann ein zweites mal mit einer zweiten Hashfunktion $h_2(pos_i)$ gehasht und das Ergebnis anstelle des ursprünglichen Hashwerts r in der Platzierungsfunktion benutzt. So lautet die Platzierungsfunktion dann

$$H_2(r) := \min_i \left(\frac{-\ln(1 - (h_2(r \circ s_i) - s_i) \bmod 1)}{w_i} \right)$$

Bildlich kann man sich das so vorstellen, als würde man die Funktionen der Knoten für jedes Datenelement auf dem Ring M auf eine andere Art verschieben, so dass sie für jedes Datenelement in einer anderen Anordnung positioniert sind. In jeder Anordnung dominieren die einzelnen Knoten dann verschieden stark. Über viele Datenelemente hinweg mittelt sich dies dann, und es gibt hier keinen statischen Fehler mehr als wenn die Knoten nur genau einmal gehasht würden.

Anwendung von DHHT in 3nuts

In der Definition von DHHT werden Datenelemente an gewichtete Knoten zugewiesen. Übertragen auf die Lastbalancierung von 3nuts werden Peers auf Pfade im gewichteten Baum abgebildet. Ein Peer beteiligt sich dann an der Verwaltung an dem ihm zugewiesenen *normalen Pfad*. Teilbäume mit höherem Teilgewicht werden so mehr Peers zugewiesen, so dass eine gleichmäßige Verwaltung des gewichteten Baums erzielt wird. Zur Wahl des normalen Pfads im Datenbaum beginnt jeder Peer in der Wurzel und wählt den nächsten Knoten auf dem Pfad aus den möglichen Kindknoten aus dem gewichteten Baum mit Hilfe von DHHT aus. Hierbei entspricht ein 3nuts Peer einem Datenele-

ment in DHHT und ein Baumknoten mit einem Gewicht entspricht einem Knoten mit einer Kapazität (Gewicht). Den Baumknoten mit einem höheren Gewicht werden also mehr Peers zugewiesen. Mit dem Gewicht ist hier immer das Gewicht des gesamten Subbaums gemeint, den der Baumknoten als Wurzel repräsentiert. Dies ist in Abbildung 2.4 dargestellt. Auf der linken Seite ist zu sehen, wie sich ein Peer auf seinem

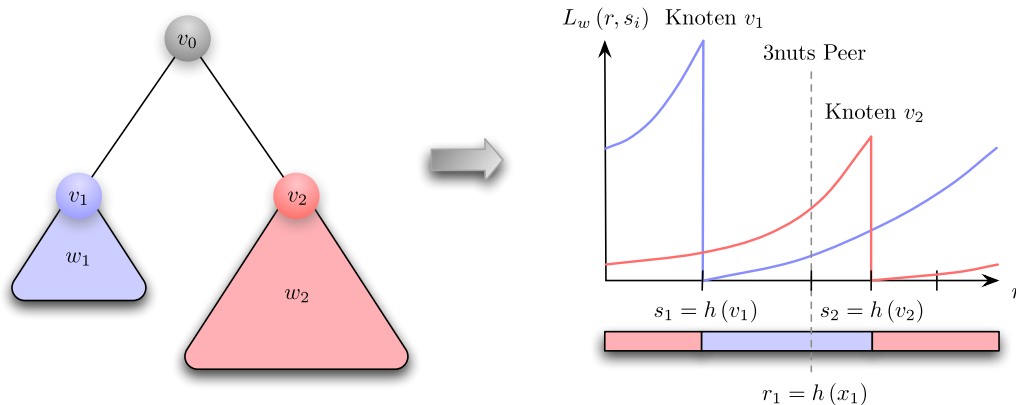


Abbildung 2.4.: Anwendung von DHHT in 3nuts

normalen Pfad endend in Knoten v_0 für einen der beiden Zweige v_1 oder v_2 entscheiden muss. Auf der rechten Seite ist die dazugehörige gewichtete verteilte Hashtabelle zu sehen. Dabei habe der der Subbaum von v_2 ein etwa doppelt so hohes Gewicht w_2 , so dass die Funktion um Faktor $\frac{1}{2}$ flacher verläuft. Der Peer entscheidet sich hier für Zweig v_1 .

Da der Grad des Baums klein sein kann z.B. bei einem Binärbaum 2, wird in 3nuts das Verfahren mit Double Hashing benutzt. Zusätzlich erhält ein Peer in verschiedenen Baumtiefen unterschiedliche Hashwerte. Hierzu wird einfach in jeder Tiefe auf die ID des Peers die Baumtiefe aufaddiert und das Resultat dann gehasht. Durch die in jeder Baumtiefe zufällig wechselnden Hashwerte der Peers wird gewährleistet, dass die Hashwerte immer über den gesamten Intervall der Hashtabelle verteilt sein können. In der Hashtabelle in Abbildung 2.4 würden beispielsweise alle Peers Knoten v_1 wählen (blau), die ihren Hashwert im den Grenzen $[s_1, s_2)$ haben (blauer Balken). Würden diese Peers dann in Knoten v_1 den nächsten Kindknoten mit den gleichen Peer-Hashwerten wählen, so hätten die Peers nur noch Hashwerte aus dem verkleinerten Bereich $[s_1, s_2)$ und nicht mehr aus dem gesamten Intervall $[0, 1)$. Aus den zufällig wechselnden Hashwerten in den Baumtiefen folgt außerdem, dass ein Peer sich z.B. in einem Binärbaum mit immer den gleichen IDs 0 und 1 in jeder Tiefe nicht immer für den gleichen Kindknoten bei gleichem Gewichtsverhältnis entscheidet. Denn seine Position in der Hashtabelle ist ja jedes mal anders.

Verteilte gewichtete Hashtabellen bieten hier also ein einfaches für die Peers unabhängiges Verfahren zur Lastbalancierung. Allerdings erkauft man sich diese Vorteile dadurch, dass die Lastbalancierung nicht zwingend optimal sein muss. Denn die For-

2. Beschreibung des Netzwerks

derung des gewichteten Hashing ist hier nur, dass entsprechend der Gewichte ein Peer mit einer Wahrscheinlichkeit w_i/W in Subbaum mit Wurzel v_i und Gewicht w_i und Gesamtgewicht aller Subbäume W gehasht wird. Dabei tritt natürlich eine gewisse Varianz auf. Außerdem wird man in einem Netzwerk, in dem sich Peers und Daten ständig dynamisch ändern, keine optimale Lastbalancierung erreichen können sondern nur nähern können.

Des Weiteren wird DHHT zusätzlich genutzt um für jeden Baumknoten aus allen darin teilnehmenden Peers einen eindeutigen verantwortlichen Peer zu bestimmen. Für den Fall, dass einmal mehrere Peers ihren normalen Pfad auf einem Datenpfad haben, verwaltet von diesen dann der Verantwortliche die Daten des Baumknotens. Hierzu können die Peers aufsteigend nach der Höhe der gewichteten Hashfunktion für den betreffenden Baumknoten geordnet werden, und der Peer mit der kleinsten Höhe am Anfang dieser Ordnung ist dann verantwortlich. Abbildung 2.5 zeigt ein Beispiel in dem zwei Peers in ihrer Verantwortung für einen Baumknoten v_1 geordnet werden, wobei Peer x_1 eine höhere Verantwortung mit einer kleineren Höhe in der Funktion besitzt als Peer x_2 . Da die Logarithmusfunktion $L_w(r, s)$ auf dem Ring $[0, 1)$ streng monoton

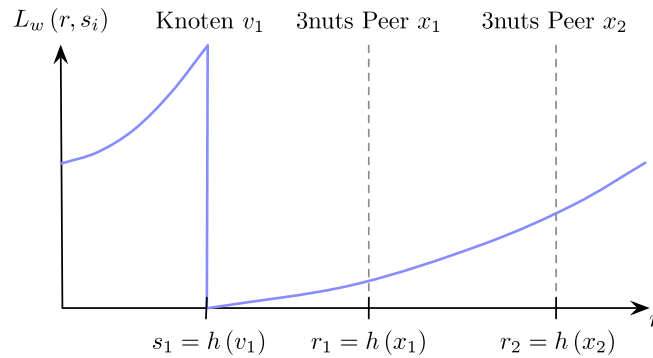


Abbildung 2.5.: Gewichtetes Hashing ordnet Verantwortung der Peers in Baumknoten

steigend ist bis auf die Sprungstelle (im Beispiel s_1), gilt hier für zwei unterschiedliche Hashwerte der Peers x_1 und x_2 , dass die Höhenwerte L_w hier auch unterschiedlich sein müssen und die Ordnung so eindeutig ist:

$$h(x_1) \neq h(x_2) \Rightarrow L_w(h(x_1), s) \neq L_w(h(x_2), s)$$

Die Monotonie der Funktion L_w kann auch für eine einfachere Berechnung der Ordnung zweier Peers genutzt werden: Hier reicht schon der Vergleich, welcher Hashwert der Peers sich rechtsseitig näher an der Sprungstelle auf dem Ring befindet.

2.1.2. Zufallsnetzwerke

Zufallsnetzwerke bilden das Grundgerüst eines 3nuts Netzwerks. Zu den Stärken von Zufallsnetzwerken zählen u.a. ein robuster Zusammenhang auch für kleine Grade, ein mit hoher Wahrscheinlichkeit logarithmischer Durchmesser sowie die Expansionseigenschaft. In einem 3nuts Netzwerk verbinden sich alle Peers, die sich in demselben Baumknoten befinden zu einem Zufallsnetzwerk um Informationen über diesen Baumknoten auszutauschen. Es gibt in 3nuts also mehrere Zufallsnetzwerke wobei als Basis des Netzwerks zunächst immer ein Zufallsnetzwerk in der Wurzel des Baums besteht. Die Implementierung der Zufallsnetzwerke erfolgte im Rahmen meiner Bachelorarbeit. Daher werden die Ergebnisse hier nur kurz zusammengefasst.

Die Zufallsnetzwerke basieren auf gerichteten Graphen mit einem regulären Ausgrad d . Der Grad wird hier recht klein gewählt z.B. $d = 3$. Bei 3nuts spielt die Forderung, dass die Struktur der Netzwerke echt zufällig ist, eine wichtige Rolle. Denn nur unter dieser Voraussetzung gilt, dass die Lookup-Operation von 3nuts eine mit hoher Wahrscheinlichkeit logarithmische Laufzeit $\mathcal{O}(\log n)$ in Abhängigkeit zur Peeranzahl n hat (Abschnitt 2.2.2). Zur Erzeugung und Aufrechterhaltung der zufälligen Struktur wird die *Pointer-Push&Pull*-Operation verwendet [MS06]. Dabei handelt es sich um eine Operation zur Transformation eines Multi-Digraph mit regulärem Ausgrad d . In solch einem gerichteten Graphen hat jeder Knoten genau d ausgehende Kanten zu anderen Knoten. Selbstschleifen von einem Knoten zu sich selbst sind dabei erlaubt. Auch Mehrfachkanten, also dass mehrere der d Kanten von einem Knoten auf den gleichen Zielknoten zeigen, können auftreten. Hierbei sind die einzelnen Kanten eindeutig nummeriert, so dass zwischen den Mehrfachkanten unterschieden werden kann. Der Graph muss nur schwach zusammenhängend sein. Es muss also kein gerichteter Pfad bei allen beliebigen Knotenpaaren bestehen.

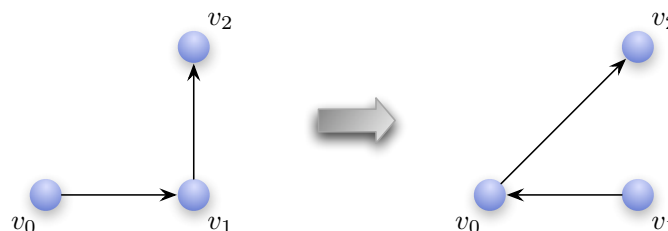


Abbildung 2.6.: *Pointer-Push&Pull Operation*

Die Transformation eines Graphen $G = (V, E)$ in einer Pointer-Push&Pull-Operation ist wie folgt definiert (vgl. Abb. 2.6):

1. Wähle einen zufälligen Knoten $v_0 \in V$
2. Durchlaufe einen Random-Walk beginnend bei Knoten v_0 und Länge 2 und Ergebnis (v_0, v_1, v_2) . Wähle dabei in jedem Schritt des Random-Walk eine zufällige Kante aus der Kantenmenge des besuchten Knoten.

2. Beschreibung des Netzwerks

3. Transformation des Graphen: Entferne Kante (v_0, v_1) und füge Kante (v_1, v_0) hinzu. Entferne Kante (v_0, v_1) und füge Kante (v_0, v_2) hinzu.

Die 3 Knoten v_0, v_1, v_2 sind hierbei vor und nach Ausführung der Transformationsoperation in einer Kette verbunden, so dass der Zusammenhang des Graphen durch die Anwendung nicht beeinflusst wird.

In einem Zufallsnetzwerk hat jeder Peer übertragen eine Routingtabelle mit d Einträgen, die auf zufällige Peers im Netzwerk – hier als Nachbarn bezeichnet – zeigen. Abbildung 2.6 zeigt in einem vereinfachten Sequenzdiagramm den Ablauf der Operation in einem Zufallsnetzwerk. Damit die Operation wie im ersten Schritt bei einem

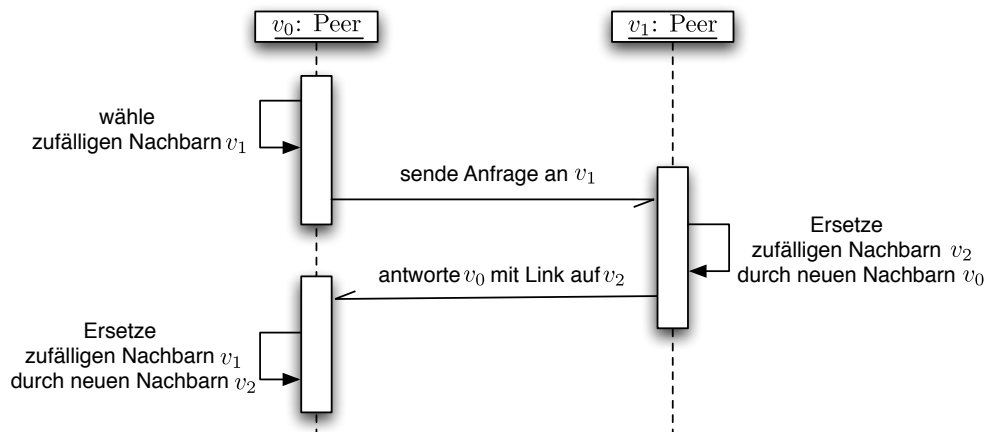


Abbildung 2.7.: Sequenzdiagramm für Pointer-Push&Pull in Zufallsnetzwerk

zufälligen Peer v_0 beginnt, entscheidet jeder Peer im Netzwerk für sich selbst verteilt in einem Markov-Prozess und Rundenintervall, wann er die Operation zufällig anstößt. Daraufhin wählt er den zweiten Peer v_1 in dem Random-Walk (v_0, v_1, v_2) aus Schritt 2 zufällig in seiner Routingtabelle und sendet diesem eine Anfrage. Peer v_1 fährt mit dem Random-Walk fort und wählt den letzten Peer v_2 zufällig in seiner Routingtabelle. Damit kann dann die Änderung der Netzwerkstruktur beginnen, indem Peer v_1 den Link auf v_2 gegen einen Link auf v_0 ersetzt. Der ersetzte Link v_2 wird in einer Antwortnachricht an Peer v_0 zurückgesendet. Peer v_0 kann die Operation dann beenden, indem er in der eigenen Routingtabelle den Link auf v_1 durch den neuen Link auf v_2 ersetzt.

Da es sich im Protokoll der Operation um eine Interaktion bestehend aus Anfrage und Antwort handelt, kann diese auch gleich zur Überprüfung der Nachbarschaftsliste genutzt werden, ohne dass zusätzlicher Netzwerkverkehr entsteht. Antwortet Peer v_1 nicht auf die Anfrage von Peer v_0 z.B. weil v_1 das Netzwerk verlassen hat, dann ist der Link (v_0, v_1) ungültig und muss repariert werden. Dazu wird in der Routingtabelle von Peer v_0 der Link auf v_1 durch einen anderen gültigen Link ersetzt. Da die Operation von Peer v_0 aus immer mit einem zufälligen Nachbarn v_1 ausgeführt wird, wird dabei mancher Nachbar in der Routingtabelle mit geringer Wahrscheinlichkeit über lange Zeit nicht überprüft. Um dies zu vermeiden verfügt ein Peer über eine *Ping*-

Operation, in der alle Nachbarn einmal eine Ping-Anfrage erhalten und zur Überprüfung mit einem Pong antworten müssen. Die Ping-Operation kann optional und dann in einem weitaus höheren Intervall als die Pointer-Push&Pull-Operation angestoßen werden, um so einen bestimmten Überprüfungsintervall zu garantieren. Des Weiteren verfügt jeder Peer noch um eine zusätzliche Routingtabelle mit begrenzter Länge, in der die aus den letzten Pointer-Push&Pull-Operationen ersetzten Links gespeichert werden. Die Links in dieser Routingtabelle werden nicht aktiv überprüft, verursachen also keine Netzwerkkosten und werden nur zur Erhöhung der Netzwerkzusammenhangs benutzt. Wird ein Link in der aktiven Routingtabelle durch Ping- oder Pointer-Push&Pull-Operation als ungültig identifiziert, so wird auf diese Links zurückgegriffen und der ungültige Link wird durch einen dieser Links ersetzt. Und die Qualität dieser Links ist nicht schlecht. Denn in der Pointer-Push&Pull-Operation wird immer genau der Link (v_0, v_1) in der aktiven Routingtabelle ersetzt und in die zusätzliche Routingtabelle verschoben, der gerade mit Anfrage- und Antwort-Nachricht überprüft wurde.

Zusätzlich bietet ein Peer in einem Zufallsnetzwerk die Möglichkeit an, Daten an zufällige Nachbarn im Netzwerk zu übertragen. Diese Daten werden hier einfach den beiden Nachrichten in der Pointer-Push&Pull-Operation angehängt. Praktisch hat dies den Vorteil, dass zur Datenübertragung keine zusätzlichen Nachrichten (Pakete) versendet werden müssen, sondern die einzelnen Nachrichten nur etwas größer werden. In 3nuts wird dies maßgeblich zum Austausch von Bauminformationen benutzt. Und propagiert hier jeder Peer immer die neuste Version der gleichen Informationen an zufällige Nachbarn, so wirkt diese Operation wie ein *Broadcast*.

Das Protokoll der Pointer-Push&Pull Operation kann auch genutzt werden um den *Ping* – oder auch bezeichnet als *Round Trip Time* (RTT) – zu messen. Hierzu muss der erste Peer v_0 in der Operation einfach die Zeitspanne zwischen dem Versenden der Anfrage und dem Erhalt der Antwort von Peer v_2 messen. Die hieraus gewonnen Ping-Information mit dem dazugehörigen Link (v_0, v_1) wird in 3nuts dann zum Aufbau der Latenz-optimierten Routingtabellen verwendet. Dabei entsteht in einem 3nuts Netzwerk also auch kein zusätzlicher Nachrichtenaufwand.

Weise teilen sich die Peers immer weiter in Teilmengen auf, bis jeder Peer alleine ist oder aber eine Menge Peers in demselben Blatt endet und sich nicht weiter aufteilen kann. In der Regel ist aber davon auszugehen, dass die Anzahl Daten sehr viel größer als die Anzahl Peers ist und daher meist jeder Peer alleine endet. Um die Netzwerkstruktur klein zu halten werden lange Pfadendungen ohne weitere Abzweige nicht im Netzwerkbaum repräsentiert. Denn ohne Abzweige kann die Menge Peers und somit die Last nicht weiter unterteilt werden. So endet der Netzwerkbaum im Beispiel schon im Pfad '1' mit 2 Peers in einem Blatt, da dieser Pfad nur durch den einzelnen Pfad '10' erweitert wird. Befindet sich ein Peer alleine in einem Subbaum (Beispiel Pfad '00'), dann muss dieser Peer diesen Subbaum ohnehin alleine verwalten. Den Subbaum hier durch weitere Zufallsnetzwerke zu repräsentieren wäre unnötig da der Peer ohnehin dort alleine wäre. So wird in diesem Fall nur die Wurzel als Zufallsnetzwerk erzeugt zu dem Zweck, dass weitere Peers beitreten können. Und wenn dies geschieht und der Peer nicht mehr alleine ist, wird der Netzwerkbaum erweitert (im Beispiel durch Zufallsnetzwerke der Pfade '000' bzw. '001').

Mit DHHT werden die Peers unabhängig voneinander mit einer Wahrscheinlichkeit abhängig von den Zweiggewichten auf die einzelnen Zweige gehasht. Hierbei kann der Fall eintreten, dass sich gar kein Peer für einen bestimmten Baumzweig entscheidet etwa dann, wenn das Gewicht dieses Teilbaums verhältnismäßig klein ist. Wenn sich mehrere Peers in einem Baumknoten befinden und sich auf die Zweige darunter aufteilen um die Datenverwaltung untereinander aufzuteilen, dann muss sich aber in jedem Zweig mindestens ein Peer befinden. Denn es muss ja einen Peer geben, dem die Verantwortung für den Pfad zugewiesen werden kann. Eine einfache Lösung ist hier

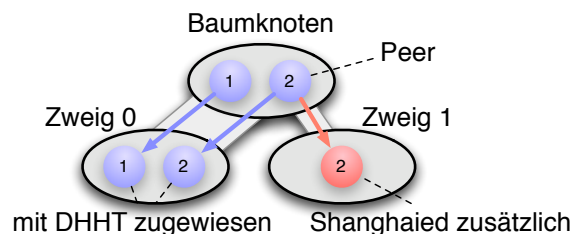


Abbildung 2.9.: 2 Peers wählen mit DHHT zur Aufteilung denselben Zweig 0. Peer 2 muss deshalb zusätzlich Shanghaied in Zweig 1 teilnehmen.

einen nicht zugewiesenen Zweig von einem Peer mitverwalten zu lassen. Dazu nimmt ein Peer zum einen im Zweig teil, den er mit DHHT bestimmt hat und noch zusätzlich in dem weiteren Zweig, in dem noch kein Peer ist. Solch ein Peer wird in 3nuts dann als *Shanghaied* bezeichnet. Dieser wird aber meist nicht sehr viel Mehraufwand haben, da das Gewicht im zusätzlichen Pfad wahrscheinlich klein ist. Und ein einzelner Peer in einem Pfad benötigt auch keine Instandhaltungskosten für ein 1 Peer Zufallsnetzwerk. Gewählt wird hier immer der Peer, der mit DHHT die höchste Verantwortlichkeit für die Wurzel dieses Zweigs hat und im Vater-Knoten darüber bereits teilnimmt. Dieser Fall tritt verstärkt bei der Aufteilung einer kleinen Peerzahl n weit unten im Baum auf,

2. Beschreibung des Netzwerks

da hier die Wahrscheinlichkeit $(1 - \frac{w_i}{W})^n$ für einen nicht zugewiesenen Pfad größer ist.

Ein Peer darf auch neben dem normalen Pfad mit DHHT auch noch zusätzlich in weiteren Pfaden freiwillig teilnehmen. Dies wird in 3nuts als *Interessenlokalität* bezeichnet. Nimmt ein Peer freiwillig in einem Baumknoten teil, so darf er bevorzugt gegenüber den anderen Peers die Daten dort verwalten. So könnte er z.B. mit einem freiwilligen Pfad eigene Daten selbst verwalten. Zudem hat der Peer dann den Vorteil, dass er mit der Teilnahme in dem Pfad direkt dorthin routen kann und so Anfragen betreffend des Pfads schnell für sich selbst und für andere beantworten kann. Da der Peer mit dem normalen Pfad schon mindestens in der Wurzel teilnimmt muss er dies für weitere freiwillige Pfade nicht doppelt machen. So muss er nur in den Zufallsnetzwerken der zusätzlichen freiwilligen Baumknoten teilnehmen wo er es nicht schon ohnehin tut.

Die interne Repräsentation des Baums bei einem Peer besteht also nicht nur aus dem normalen Pfad mit allen abgehenden Zweigen sondern aus einem Teilbaum des gesamten Baums. Der Peer hat sozusagen eine eingeschränkte Sicht auf den Baum. Selbst nimmt er dabei in den Zufallsnetzwerken des normalen Pfads, freiwilligen Pfaden sowie einzelnen Knoten als einzelner Shanghaied'er Peer teil. Alle Baumknoten, in denen sich ein 3nuts Peer im Zufallsnetzwerk verbindet, werden als *TrunkNode* bezeichnet. Für alle von diesem Teilbaum abgehenden Zweige hat er einen Link auf einen Peer, der dort teilnimmt. Dieser wird in 3nuts als *BranchLink* bezeichnet. Abbildung 2.10 zeigt zusammenfassend ein Beispiel für die interne Baumstruktur eines Peers.

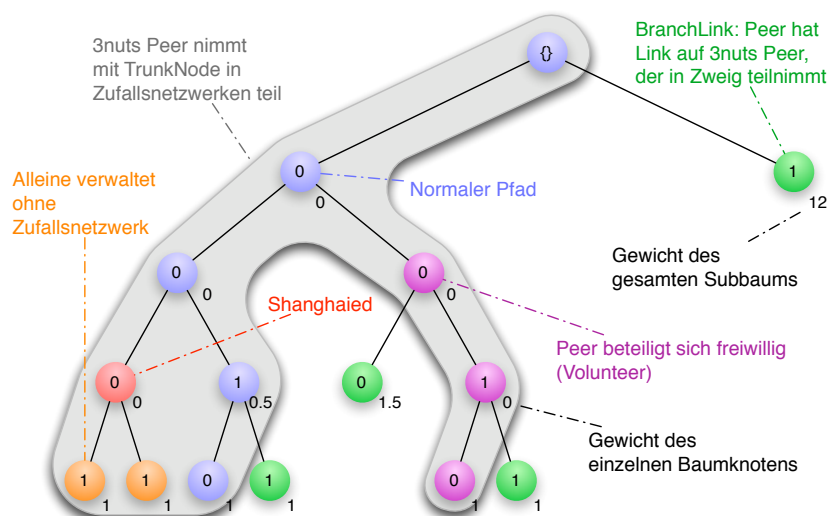


Abbildung 2.10.: Eingeschränkte Sicht eines 3nuts Peers auf den Netzwerkbaum

Verantwortlichkeit in Baumknoten

In 3nuts ist die eindeutige Abbildung der Daten auf die Peers nicht allein durch die Netzwerkstruktur gegeben. So gibt es hier keine Forderung, dass bei der Aufteilung der

Peers auf den Baum jeder Peer einen eindeutigen Pfad haben muss und die Datenpfade immer länger als die Pfade der Peers sind. Denn dann könnten jedem Peer einfach alle Daten zugeordnet werden, die den Präfix des Peer-Pfads beinhalten (siehe Abb. 2.2). In 3nuts dürfen die Datenpfade beliebig aufgebaut sein. Daten können hier also nicht nur auf Blättern sondern auch auf internen Knoten des Baums abgelegt werden. Dabei kann es nun dazu kommen, dass mehrere Peers in einem Baumknoten teilnehmen, auf dem Daten mit einem Gewicht liegen. Grundsätzlich gewährleisten erst einmal alle diese Peers das Routing zu diesem Pfad. Aus dieser Menge Peers muss ein eindeutiger Peer bestimmt werden, der für die Daten in dem Pfad eindeutig verantwortlich ist. Dieser speichert dann die Daten auf Applikationsebene und kann Anfragen dazu beantworten. Hierzu führen alle Peers im selben Zufallsnetzwerk des gleichen Baumknoten eine so genannte *Verantwortlichkeitsliste*. Diese Liste beinhaltet die Peers aus dem Zufallsnetzwerk absteigend nach ihrer Verantwortlichkeit für den Baumknoten. Da sich viele Peers gleichzeitig im Zufallsnetzwerk eines Baumknotens befinden können (z.B. in der Wurzel alle Peers), hat die Liste eine begrenzte Länge. Die Ordnung der Peers in der Liste hat 2 Kriterien:

1. Verantwortungsgrad: Volunteer, Normal, Volunteer_Down, Shanghaied
2. kleinere Höhe in DHHT des Baumknotens

Zunächst werden hierbei also diejenigen Peers mit dem Verantwortungsgrad Volunteer priorisiert, die in dem Baumknoten freiwillig teilnehmen und die Verantwortung dafür haben möchten um die Daten dort zu verwalten. Die zweite Priorität mit Verantwortungsgrad Normal erhalten dann diejenigen Peers, die auf ihrem normalen Pfad mit DHHT in dem Baumknoten teilnehmen. Bei der Ordnung von Volunteer und Normal stellt sich das Problem, dass ein Peer nur freiwillig in einem einzelnen Baumknoten am Ende eines Pfads teilnehmen möchte. Er möchte nicht bevorzugt gegenüber den "normal eingestuftten Peers" die Verantwortung für alle Baumknoten von der Wurzel an erhalten. Deshalb gibt es nochmals einen abgestuften Verantwortungsgrad Volunteer_Down mit einer niedrigeren Priorität als Normal. Dieser Grad wird dann auf einem freiwilligen Pfad bis zum eigentlichen letzten freiwilligen Baumknoten gesetzt, wo dann Volunteer benutzt wird. Der geringste Grad der Verantwortung ist zuletzt Shanghaied. Dies sind diejenigen Peers, die notgedrungen als *Shanghaied* die Verantwortung für einen Baumknoten übernommen haben, weil kein andere Peer auf dem normalen Pfad oder gar freiwillig dem Baumknoten beigetreten ist. Sobald ein Peer mit höherem Verantwortungsgrad dem Zufallsnetzwerk des Baumknotens beitrifft, wird der Shanghaied'e Peer nicht mehr benötigt und kann seine Rolle dort aufgeben.

Wenn 2 Peers den gleichen Grad der Verantwortung haben, dann ergibt sich die Ordnung durch DHHT. Hierbei erhält der Peer die höhere Verantwortlichkeit, der die kleinere Höhe in der Hashtabelle für den Baumknoten hat. Wie schon in Kapitel 2.1.1 beschrieben, ist die Ordnung zweier verschiedener Hashwerte der beiden Peers eindeutig, so dass die Liste ebenfalls eine eindeutige Ordnung erhält. Sollte der Fall auftreten, dass 2 Peers den gleichen Hashwert haben, dann kann hier noch mal die Netzwerk-

2. Beschreibung des Netzwerks

adresse als drittes Kriterium verglichen werden, die dann aber eine eindeutige Ordnung festlegt.

Bei der Erzeugung der Verantwortlichkeitsliste erstellt jeder Peer eine leere Liste und fügt sich selbst hinzu. Aus seiner eigenen Sicht ist er also erst einmal selbst verantwortlich. Über das Zufallsnetzwerk wird die Verantwortlichkeitsliste mit zufälligen anderen Peers ausgetauscht. Dabei wird dann die eigene Liste aktualisiert indem sie mit der Liste eines anderen zufälligen Peers entsprechend der Ordnung der Verantwortlichkeit zusammengefügt wird. Diese Liste wird dann auf eine konstante Länge (z.B. 3) abgeschnitten, so dass nur die "Top-Verantwortlichen" übrig bleiben. Der erste in dieser Liste ist dann der Verantwortliche für den Baumknoten. Alle weiteren Listeneinträge werden für eine bessere Robustheit benutzt: fällt der Verantwortliche aus, so wird direkt der zweite in der Liste als Verantwortlicher benutzt, der direkt jedem Peer bekannt ist. Tritt der Fall ein und ein Peer fällt aus, dann wird der Eintrag aus der Liste gelöscht und ist ein Peer selbst nicht in der Auswahl der Liste enthalten, fügt er sich mit dem freigewordenen Platz am Ende der Liste ein. Durch den ständigen Austausch der Liste zwischen zufälligen Peers im Zufallsnetzwerk werden die Listen aller Peers auf eine einheitliche Version abgeglichen. Hierbei werden die korrekten Verantwortlichen wie bei einer epidemischen Informationsausbreitung publiziert. Denn hat ein Peer einmal einen Eintrag aus der letztendlich korrekten Liste erhalten, dann wird dieser niemals wieder durch einen anderen Eintrag aus der Liste verdrängt und kontinuierlich mit Push&Pull an andere zufällige Peers weitergegeben.

Eine Aufgabe des Verantwortlichen eines Baumknotens besteht in der Verwaltung der Daten für diesen Pfad auf Applikationsebene. Diese Verantwortung wird aber nur durchs 3nuts Netzwerk zugewiesen und die dadurch resultierende Verwaltung muss in der Applikation umgesetzt werden, die den verantwortlichen 3nuts Peer benutzt. Die Granularität der Datenverwaltung in 3nuts beschränkt sich somit auf einzelne Pfade. Es könnten aber mehrere Applikationen/Peers gleichzeitig Daten unter demselben Pfad publizieren, so dass die Verwaltung dieser Menge Daten theoretisch noch weiter durch die unterschiedlichen Quellen unterteilt werden könnte. Um dies zu erreichen müssten die einzelnen Daten, die auf den gleichen Pfad abgebildet sind, eine eindeutige Pfaderweiterung erhalten. Dann liegen sie auf verschiedenen eindeutigen Baumknoten und können von verschiedenen Peers verwaltet werden. Mit den Pfaderweiterungen liegen die Peers dann immer noch lokal im selben Subbaum mit entsprechenden Zugriffsmöglichkeiten. Solch ein Verfahren würde an dieser Stelle auch durch die Applikation erfolgen.

Ist ein Peer in einem Pfad verantwortlich, dann muss die Applikation des Peers für diesen Pfad oder genauer gesagt für diesen Baumknoten das Gewicht bestimmen und im Peer setzen. Zur Bestimmung des Gewichts können die Daten selbst, die in der verantwortlichen Applikation liegen, oder auch andere Faktoren wie Zugriffshäufigkeit oder Ähnliches herangezogen werden. Der verantwortliche Peer sorgt dann dafür, dass das Gewicht im Netzwerk verbreitet wird und so der gewichtete Baum für die Lastbalancierung entsteht. Dies kann auch bedeuten, dass der Peer hierzu zunächst den Netzwerkbaum um den neuen Pfad erweitern muss.

Austausch und Instandhaltung der Baumstruktur

Die Peer- und Datenmenge kann sich zur Laufzeit des Netzwerks ständig ändern. Demzufolge müssen die Peers ständig ihre Informationen über das Netzwerk aktualisieren um so zum einen die Routingstruktur aufrecht zu erhalten und zum anderen die Lastbalancierung an die sich ändernden Daten anpassen zu können. Hierzu tauschen sie Informationen jeweils mit anderen Peers im selben Baumknoten über das darunter liegende Zufallsnetzwerk aus. Dies ist sinnvoll, da all diese Peer sowohl die gleiche Routingstruktur in die Kindknoten des Baumknotens aufrechterhalten wollen, als auch über die Daten in dem Baumknoten selbst auf dem Laufenden gehalten werden wollen, was die Wahl des Verantwortlichen und das Gewicht dieser Daten beinhaltet. Ein Peer, der in einem ganz anderen Teil des Baums teilnimmt, könnte mit diesen Daten nichts anfangen. Der Austausch der Daten erfolgt mit zufälligen Nachbarn im Zufallsnetzwerk wofür das Zufallsnetzwerk mit *Pointer-Push&Pull* eine Transportmöglichkeit anbietet (Kapitel 2.1.2).

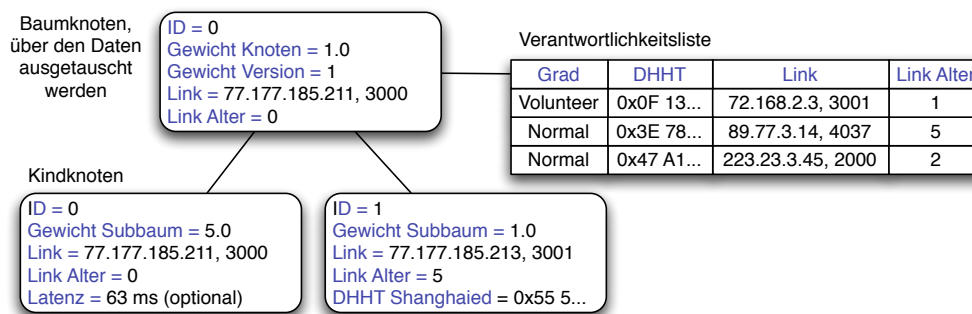


Abbildung 2.11.: Beispiel für ausgetauschte Daten über einen Baumknoten

Die Informationen für den Baumknoten selbst enthalten die Verantwortlichkeitsliste um diese zwischen den Peers abzugleichen und einen Verantwortlichen zu ermitteln. Der Verantwortliche bestimmt dann ein Gewicht für den Baumknoten und publiziert es an dieser Stelle. Da sich das Knotengewicht ändern kann, wird es versioniert. So überprüft ein Peer bei einem empfangenen Gewicht zunächst bevor er es übernimmt, ob der andere Peer denn den richtigen Verantwortlichen kannte und die Gewichtsversion neuer ist. Auf diese Weise erhält jeder teilnehmende Peer das aktuelle Gewicht und kann es dann weiter im Netzwerk verbreiten. Des Weiteren wird für jeden Abzweig des Baumknotens ein BranchLink ausgetauscht. Dieser enthält die ID des Zweigs (z.B. Binärbaum $\{0,1\}$), einen Link eines darin teilnehmenden Peers sowie ein Gewicht für den gesamten Subbaum. Dabei errechnet sich das Gewicht des Subbaums immer aus der Summe aller Baumknoten darin. Nimmt der Peer dabei selbst in einem Baumknoten teil, so erhält er dafür immer das aktuelle Gewicht vom Verantwortlichen wie oben angegeben und kann dies hier in die Summe einbringen. Oder aber er nimmt in einem Zweig selbst nicht teil und hat dann von einem anderen Peer einen BranchLink für den Zweig erhalten, der ein schon zusammengerechnetes Gewicht für den gesamten Zweig

2. Beschreibung des Netzwerks

enthält. So kann jeder Peer mit seiner eingeschränkten Baumsicht rekursiv das Gesamtgewicht des Baums errechnen und damit seinen normalen Pfad bestimmen und bei Änderungen gegebenenfalls anpassen. Falls ein Peer Shanghaied in einem Zweig ist, dann fügt er dem BranchLink für den Zweig seinen DHHT Hashwert hinzu (Abb. 2.11). So können die anderen Peers dann prüfen ob sie dort eine höhere Verantwortung haben und dann den aktuellen Peer als Shanghaied ablösen müssen.

Der Austausch dieser Daten erfolgt immer zwischen zwei zufälligen Nachbarn im Zufallsnetzwerk. Da jeder Peer jedes mal immer die gleichen Bauminformationen nur auf einem aktualisierten Stand an einen zufälligen Nachbarn in Push&Pull Manier (siehe [KSSV00]) schickt, kann diese Form des Austauschs auch als *Broadcast* verstanden werden. Nimmt beispielsweise nur ein Peer in einem Baumzweig teil, dann schickt er den BranchLink auf den Zweig an einen zufälligen Nachbarn im Baumknoten darüber, dieser Nachbar speichert den BranchLink auf den Zweig und verbreitet ihn selbst weiter an zufällige Nachbarn. Auf diese Weise verbreitet sich die Nachricht über den Zweig im Zufallsnetzwerk immer weiter bis jeder Peer informiert ist und in den Zweig routen kann. Die Verbreitung erfolgt über mehrere Runden der *Pointer-Push&Pull-Operation* und über mehrere Hops. Die Links die dabei dann weitergeschickt werden und in den Routingtabellen gesetzt werden stammen daher nicht zwangsweise direkt vom Sender. Aus diesem Grund werden die Links mit einem *Alter* versehen, dass kontinuierlich steigt. Nimmt ein Peer in einem Zweig selbst teil, dann verschickt er dafür immer seinen eigenen Link mit Alter 0 (siehe Abbildung 2.11 Kindknoten mit ID 0). Erhält ein Peer mehrere BranchLinks für den gleichen Zweig, dann behält er immer den Link mit dem kleinsten Alter. So ist keine aktive Überprüfung der Links erforderlich, denn alte Links werden durch neue nach und nach automatisch überschrieben. Dieses Vorgehen ist besonders wichtig, falls einer der Teilbäume ein sehr kleines Gewicht hat. In dem Fall nimmt in dem Zweig nur eine ganz kleine Teilmenge der Peers teil, und da jeder Peer einen Link auf den Zweig haben muss, würden diese Peers bei einer aktiven Überprüfungsstrategie dann übermäßig häufig angefragt und hätten dadurch eine höhere Netzwerklast. In 3nuts werden Links nur durch den Nachrichtenaustausch im Routing selbst aktiv überprüft, was keine zusätzliche Nachrichtenlast erfordert. Um hierbei das Routing sicherzustellen und eventuell häufiger auftretende inkorrekte Routingableneinträge ausmerzen zu können, werden alle Links in der Baumstruktur durch eine Liste mit konstanter Länge abgesichert. So hat ein Peer immer mehrere Möglichkeiten in einen Teil des Baums zu routen falls das Routing über einen Link mal fehlschlägt. Dies erfordert dann einen um einen konstanten Faktor größeren Speicherbedarf für den Peer, der heutzutage mit ein paar Kilobytes kein Problem darstellen sollte.

Jeder Peer verbreitet BranchLinks auf Zweige, in denen er teilnimmt, und speichert von anderen BranchLinks in Zweige, in denen er nicht ist, um dorthin routen zu können. Dabei baut sich ein Peer in jedem BranchLink 2 Listen mit redundanten Links in den Zweig auf. Die eine enthält Links auf zufällige Peers im Zweig, die durch den Zufallsprozess der Verbreitung von BranchLinks in der Pointer-Push&Pull Operation kontinuierlich zufällig gehalten werden. Durch die zufällige Wahl der Peers ist logarithmisches Routing in 3nuts (Kapitel 2.2.2) beweisbar. Zusätzlich wird hier auch eine

Routingtabelle aufgebaut, in der die Einträge anhand der physikalischen Netzwerklokalität optimiert sind. Denn möchte ein Peer in einen bestimmten Baumzweig routen, dann gibt es immer eine Teilmenge Peers, die im betreffenden Baumzweig teilnehmen und fürs Routing zur Auswahl stehen. Für ein schnelles Routing ist hier dann natürlich der Peer mit der geringsten Latenz zu wählen. Plakativ dargestellt ist z.B. ein Peer aus dem lokalen Netzwerk einem Peer von einem anderen Kontinent auf jeden Fall vorzuziehen. Die Pointer-Push&Pull Operation bietet hier eine Latenzmessung an (Kapitel 2.1.2). Dabei misst der anfragende Peer vom Senden der Anfrage bis zum Erhalt der Antwort die Latenz zum anderen Peer. Der antwortende Peer nimmt in irgendeinem Abzweig des Baumknotens teil und hat dafür dann einen BranchLink auf sich selbst geschickt welcher zusammen mit der gemessenen Latenz einen so genannten *lokalen Link* ergibt (siehe Abb. 2.11). Der anfragende Peer sortiert diesen dann aufsteigend nach der Latenz in seine Latenz-optimierte Routingtabelle für den Baumzweig ein.

Verteilte Erzeugung des Netzwerkbaums

Der Datenbereich des Baums im verteilten Index wird dynamisch durch die Daten vorgegeben und steht nicht von vornherein fest wie z.B. der Intervall $[0, 2^{128}]$ bei einem DHT. Somit muss die Struktur des Datenbaums während des laufenden Betriebs des Netzwerks ermittelt werden und in den entsprechenden Netzwerkbaum umgesetzt werden. Dabei gibt es keine zentrale Instanz, die den Netzwerkbaum durch weitere Baumknoten und Zufallsnetzwerke zentral erweitert. Die einzelnen Peers müssen verteilt und parallel die Baumstruktur errichten.

Allerdings gibt es in jedem Baumknoten mit dem Verantwortlichen einen ausgezeichneten Peer. So ist die Erzeugung von neuen Kindknoten nebst Zufallsnetzwerk nur dem Verantwortlichen im Vaterknoten darüber gestattet. Dazu erzeugt er das Zufallsnetzwerk und publiziert einen BranchLink auf sich selbst für den neuen Kindknoten, so dass andere Peers beitreten können. Falls der neue Knoten nicht auf dem normalen Pfad des Verantwortlichen liegt, tritt er auch nur als Shanghaied bei, so dass er ihn wieder verlassen kann sobald andere Peers beigetreten sind. Mit diesem Prozedere soll eingeschränkt werden, dass viele Peers gleichzeitig parallel den gleichen neuen Kindknoten erzeugen und so die gleichen eindeutigen Pfade mit mehreren Zufallsnetzwerken koexistieren. Dieser Fall kann aber nicht gänzlich ausgeschlossen werden. Denn auch der Verantwortliche muss zunächst in einem Prozess verteilt bestimmt werden. Im Entstehungsprozess ist dabei der wirkliche Verantwortliche noch nicht im gesamten Zufallsnetzwerk verbreitet. So werden zu Anfang von verschiedenen Peers verschiedene Verantwortliche angenommen. Dies kann dann dazu führen, dass zunächst Subbäume parallel von den uneindeutigen Verantwortlichen erzeugt werden. Diese parallelen Teilbäume enthalten dann alle nur einen Teil der Daten und sind dann wieder zu einem einheitlichen Baum zu synchronisieren, der dann alle Daten enthält.

Im 3nuts Netzwerk ist sichergestellt, dass das Zufallsnetzwerk in der Wurzel nur einmal erzeugt wurde. Denn die Erzeugung der Wurzel obliegt dem ersten 3nuts Peer im Netzwerk. Alle weiteren Peers treten diesem Zufallsnetzwerk dann bei. Um zwei

2. Beschreibung des Netzwerks

irrtümlich parallel erzeugte gleiche Baumknoten wieder zu verschmelzen, werden einfach die beiden dazugehörigen Zufallsnetzwerke miteinander verbunden. Dazu werden im Zufallsnetzwerk im Vaterknoten darüber (initial die Wurzel), welches schon verbunden sein muss, Links aus den beiden Teilnetzen ausgetauscht um damit eine Verknüpfung der Teilnetze herzustellen. Durch das Verbinden der Teilnetzwerke wird auch automatisch die Verantwortlichkeitsliste verschmolzen, da sie ja ständig ausgetauscht und abgeglichen wird. Dieser Synchronisierungsprozess kann sich dann rekursiv in den Kindknoten fortsetzen bis die gesamte Baumstruktur korrigiert ist. Der gleiche Prozess findet natürlich auch im Fehlerfall seine Anwendung, wenn viele Peers das Netzwerk verlassen und evtl. Teile der Baumstruktur neu aufgebaut werden müssen.

Beim Vereinen von 2 parallelen Zufallsnetzwerken für denselben Baumknoten muss praktisch eine Brückenkante von dem einen zum anderen Teilnetzwerk erzeugt werden. Durch die kontinuierliche Anpassung der Netzwerkstruktur mit der *Pointer-Push-&Pull*-Operation verwächst das Netzwerk dann weiter miteinander. Als Brückenkante kann aber nicht einfach eine zusätzlicher Link dem Netzwerk hinzugefügt werden, da ja jeder Peer den regulären Grad des Zufallsnetzwerks einhalten muss. Es muss also ein Link aus dem bestehenden Netzwerk entfernt werden und gegen die neue Brückenkante ausgetauscht werden. Das Löschen einer anderen Kante darf hier aber auch nicht den Zusammenhang eines der beiden Teilnetzwerke gefährden. Die einzigen Links, die hier bedenkenlos gelöscht werden können sind daher Selbstlinks. Davon gibt es im Erwartungswert in einem Zufallsnetzwerk mit n Peers und regulärem Grad d genau d viele. Die Peers im selben Baumknoten tauschen nun immer für alle Kindknoten einen BranchLink aus, der einen Link in dem Zufallsnetzwerk des Kindknotens enthält. Erhält nun ein Peer einen BranchLink für einen Kindknoten, in dem er selbst teilnimmt und er hat in diesem Zufallsnetzwerk selbst einen Selbstlink, dann ersetzt er diesen Selbstlink durch den anderen BranchLink. So verbindet er dann möglicherweise falls nicht schon geschehen zwei parallele Zufallsnetzwerke miteinander.

Durch koexistierende gleiche Teilbäume oder inkorrekte Verantwortlichkeitslisten kann es mehrere Verantwortliche für den gleichen Pfad geben. Publizieren dann mehrere Applikationen Daten für den gleichen Pfad, so können diese fälschlicherweise bei verschiedenen Peers landen. Wenn die Netzwerkstruktur korrigiert wird und der korrekte Verantwortliche bestimmt wird, müssen auch entsprechend die Daten auf Applikationsebene abgeglichen werden. Stellt ein Peer bei der Aktualisierung der Verantwortlichkeitsliste fest, dass er die Verantwortlichkeit an einen anderen Peer verloren hat, dann weist er die Applikation an die für diesen Pfad verwalteten Daten an die Applikation des neuen Verantwortlichen zu schicken. So erhält der richtige Verantwortliche nach und nach alle Daten für den Pfad. Die hierdurch gesteigerte Datenverfügbarkeit ist für Anfragen auf Applikationsebene wichtig. Es spielt aber auch für das 3nuts Netzwerk selbst eine große Rolle. Denn anhand der Daten wird vom Verantwortlichen das Gewicht eines Baumknotens bestimmt. Würden Baumgewichte, die eigentlich durch die Daten einen festen Wert haben, bloß wegen Netzwerkanpassungen zu stark schwanken, dann passt sich das Netzwerk wiederum an die Änderungen dieser Schwankungen an, was dann im Wechselspiel zu Schwingungen führen kann.

2.2. Algorithmen

In diesem Kapitel werden die Grundfunktionen des 3nuts Netzwerks vorgestellt, die von einer Applikation genutzt werden können. Dies umfasst die Erzeugung eines Netzwerks (2.2.1) sowie den Lookup für Punktanfragen (2.2.2) und die allgemeinere Suche für Bereichsanfragen (2.2.3).

2.2.1. Netzwerkerzeugung

Zunächst einmal gilt es ein Netzwerk zu erstellen. Das grundlegende Rückrad und initiale Struktur dabei bildet das Zufallsnetzwerk in der Wurzel des Baums. So hat der erste Peer im Netzwerk in der Operation `create()` diesen Zufallsnetzwerk zu erzeugen. Weitere Peers, die dem Netzwerk in der `join()`-Operation beitreten wollen, benötigen dann eine Adresse eines Peers in diesem Zufallsnetzwerk (der zweite Peer verwendet einen Link zum ersten Peer). Weitere Routingstrukturen werden dann mit dem Mechanismus zum Austausch von Bauminformationen aufgebaut: Ist ein Peer zur Wurzel beigetreten, so erhält er automatisch Informationen über die Kindknoten und kann dem Zufallsnetzwerk des Kindknotens auf seinem normalen Pfad beitreten. Hier erhält er wiederum Informationen über die Kindknoten und kann so rekursiv weiter die Baumstruktur aufbauen.

Mit der `leave()`-Operation kann ein Peer das Netzwerk wieder verlassen. Hier wird einfach die zuvor erzeugte Baumstruktur des Peers wieder abgebaut. Da es in einem 3nuts Netzwerk nur gerichtete Links gibt, ein Peer als in der Umkehr nicht weiß, welche anderen Peers einen Link auf ihn haben, gibt es auch keine Möglichkeit diese Links im Netzwerk direkt zu entfernen. Die betroffenen Peers mit einem Link auf den gelöschten Peer müssen den toten Link erkennen und die Routingtabelle eigenständig reparieren.

Der gewichtete Datenbaum wird von der Applikation vorgegeben. Die einzelnen Gewichte müssen Pfad für Pfad mit der Methode `setWeight(Pfad, Gewicht)` in 3nuts gesetzt werden, damit der Baum auf die Netzwerkstruktur abgebildet werden kann. Hierbei gibt es seitens des 3nuts Netzwerks die Einschränkung, dass das Gewicht eines Baumknotens immer nur exklusiv im für den Baumknoten verantwortlichen Peer gesetzt werden kann. Der verantwortliche Peer erzeugt dann die Netzwerkstruktur des Pfads und publiziert das Gewicht im Netzwerk. So wird sichergestellt, dass die Baumgewichte eindeutig sind und nicht von mehreren Peers verschiedene Werte publiziert werden. Der Mechanismus zum Setzen des Gewichts auf Applikationsebene wird hier kurz schraffiert. Zu jedem Peer gehört eine Applikation, die diesen Peer verwendet. Dabei kann es mehrere Applikationen geben, die für den gleichen Pfad Daten bereitstellen. Die Menge all dieser Daten führt in der Summe zu einem Gewicht, dass seitens der Applikation bestimmt werden muss. In das Gewicht können natürlich auch Eigenschaften wie die Zugriffshäufigkeit auf den Pfad oder Ähnliches einfließen. Da dieses Gewicht nur im verantwortlichen Peer für den Pfad gesetzt werden kann, sollte man also praktischerweise all diese Informationen entsprechend in der Applikation des Verantwortlichen Peers bündeln, daraus ein Gewicht berechnen und dies dann im Peer setzen. Dazu muss jede Applikation für die eigenen Daten die `lookup()`-Operation aufrufen, die für

2. Beschreibung des Netzwerks

einen Pfad die Adresse des verantwortlichen Peers ermittelt, und die eigenen Daten darauf hin den verschiedenen Verantwortlichen mitteilen. Diese von den Verantwortlichen verwalteten Indexdaten könnten dann auch zur Beantwortung von Anfragen auf Applikationsebene dienen, und dafür zusätzliche Metainformationen enthalten.

2.2.2. Lookup

Der Lookup ist neben der Suche die Hauptfunktion eines Peer-to-Peer Netzwerks und liefert für einen Suchschlüssel – in 3nuts ein Pfad im Baum – die Adresse des verantwortlichen Peers. In einem DHT werden auf einem vorher festgelegten Intervall z.B. 2^{128} Peers sowie Daten abgebildet, so dass für jeden beliebigen Suchschlüssel aus diesem Intervall ein Peer verantwortlich ist. Im Gegensatz dazu hat ein Suchbaum nicht fest definiert lange Pfade (z.B. Länge 128 in einem Binärbaum), sondern der Baum wird dynamisch erweitert und verändert. Dabei kann es sein, dass einem Suchpfad kein verantwortlicher Peer zugeordnet ist, falls dieser Pfad noch nicht erzeugt wurde. In diesem Fall gibt die Lookup-Operation nur die Adresse des Peers zurück, der diesen Pfad im Netzwerk erzeugen kann nicht aber zwangsläufig auch dafür später verantwortlich ist.

Algorithmus 1 beschreibt die Funktion `Lookup(key, peer)`, die für einen Suchpfad `key` beginnend bei Peer `peer` den verantwortlichen Peer zurückgibt. Im Algorithmus sind

Algorithm 1 Lookup(Pfad `key`) bei Peer `p`

```
1: if p hat BranchLink auf Peer p', der mit key längeren Präfix teilt then
2:   Leite Lookup(key) weiter an p';
3: else
4:    $v \leftarrow$  letzter Knoten von Pfad key in Baumsicht von p;
5:    $p' \leftarrow$  verantwortlicher Peer in Knoten v;
6:   if p = p' then
7:     return p;
8:   else
9:     Leite Lookup(key) weiter an p';
10:  end if
11: end if
```

Reparaturmechanismen vernachlässigt und er terminiert erfolgreich insofern, die Routingtabellen korrekt sind. Der Algorithmus ist rekursiv aufgebaut. Zunächst wird in Zeile 1 überprüft, ob der Peer auf dem Suchpfad `key` in seiner Baumsicht einen BranchLink auf einen Peer `p'` hat. Falls dem so ist wird die Operation rekursiv bei Peer `p'` fortgesetzt. Peer `p'` nimmt dann mindestens in einem Knoten des Suchpfads `key` mehr teil. So wird in jedem Rekursionsschritt der bekannte Pfad des ausführenden Peers sukzessive immer weiter an den Suchpfad `key` angepasst. Dies geschieht solange, bis die Funktion einen Peer `p` ohne BranchLink auf dem Suchpfad erreicht, der also den gesamten Präfix des Suchpfads kennt, der im Netzwerk publiziert wurde (Zeile 3). Im letzten publizierten Baumknoten des Suchpfads können mehrere Peers teilnehmen und durch die Verantwortlichkeitsliste wird einem Peer daraus die Verantwortlichkeit eindeutig

zugewiesen. Wird die Operation beim verantwortlichen Peer p' für den Knoten v ausgeführt, dann terminiert die Funktion in Zeile 7 und gibt diesen zurück. Andernfalls wird die Funktion in einem letzten Rekursionsschritt beim Verantwortlichen p' fortgesetzt, wo die Funktion dann terminiert. Im dynamischen Netzwerk ist die Verantwort-

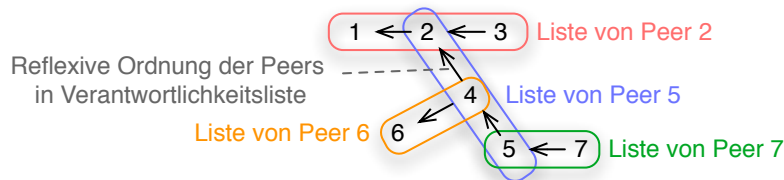


Abbildung 2.12.: Unvollständige Verantwortlichkeitslisten. Verlauf für letzten Rekursionsschritt in Lookup-Funktion begonnen bei Peer 7: ($7 \rightarrow 5 \rightarrow 2 \rightarrow 1$)

lichkeitsliste evtl. noch nicht in allen Peers abgeglichen. Dann hat jeder Peer nur eine partielle Ordnung von den Peers in der Liste, die er schon kennt. In dem Fall gibt es evtl. mehrere Verantwortliche in der Ordnung wo der Algorithmus terminiert und noch keinen einheitlichen Verantwortlichen (siehe Abb. 2.12). Da es sich bei der Verantwortlichkeitsliste aber um eine reflexive Ordnung handelt und die Menge Peers endlich ist, wird der Algorithmus hier ohne Kreise entsprechend in endlich vielen Schritten terminieren.

Abbildung 2.13 zeigt ein Beispiel für die Durchführung der Lookup-Operation. In den 3 Kästen sind dabei die internen Baumsichten von 3 verschiedenen Peers dargestellt. Ein blauer Baumknoten zeigt dabei, dass ein Peer darin mit einem TrunkNode teilnimmt, ein grüner Baumknoten weist auf einen BranchLink hin und graue halb ausgeblendete Knoten sind für den Peer nicht bekannt. In der Lookup-Operation wird in

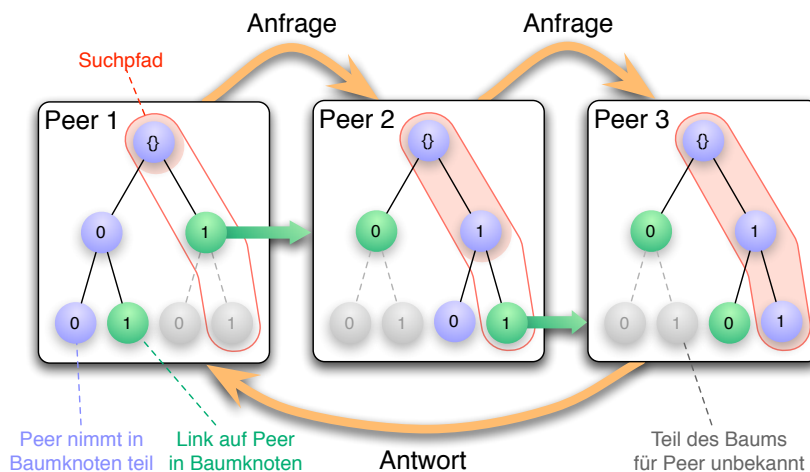


Abbildung 2.13.: Beispiel für Ausführung der Lookup-Operation

2. Beschreibung des Netzwerks

einem Binärbaum beginnend bei Peer 1 nach dem Pfad '11' gesucht. Peer 1 nimmt nur bis zur Wurzel im Suchpfad teil und kennt für den Präfix 1 den Peer 2. Die Operation wird bei Peer 2 fortgesetzt in Baumtiefe 1. Peer 2 ist der komplette Suchpfad bekannt, er nimmt aber nicht im letzten Baumknoten '11' in Tiefe 2 teil, so dass die Operation auf dem verlinkten Peer 3 rekursiv fortgesetzt wird. Peer 3 nimmt nun im kompletten Suchpfad '11' teil, und da er hier verantwortlich ist, kann er die Antwort generieren. Die allgemeine Laufzeit des Lookup-Algorithmus wird im folgenden Theorem 2.1 beschrieben.

Theorem 2.1 *Die Lookup-Operation in einem 3nuts Netzwerk mit n Peers benötigt höchstens $\mathcal{O}(\log n)$ Hops mit hoher Wahrscheinlichkeit unabhängig von der Struktur des Netzwerkbaums.*

Beweis: Im Netzwerkbaum nehmen in der Wurzel alle n Peers teil, die sich dann auf die einzelnen Baumzweige immer weiter aufteilen. Die Beweisidee ist die, dass sich nach jeweils einer konstanten Anzahl Hops die Anzahl Peers im erreichten Suchpfad jeweils mindestens halbiert (z.B. $n, \frac{n}{2}, \frac{n}{4}, \dots, 1$). Damit bleibt nach logarithmisch vielen Runden nur noch der 1 verantwortliche Peer übrig und die Operation terminiert erfolgreich. Sei P_{data} der Suchpfad in der Lookup-Operation von dem der Präfix $P_{\text{net}} = (v_1, \dots, v_i, v_j, \dots, v_k) \subseteq P_{\text{data}}$ im Netzwerkbaum erzeugt wurde. Dabei seien T_i bzw. T_j die Teilbäume mit Wurzel v_i bzw. v_j . T_i sei auf dem Suchpfad P_{net} der kleinste Teilbaum, in dem wenigstens die Hälfte aller Peers teilnehmen mit $|T_i| \geq \frac{n}{2}$. In der nächsten Tiefe auf dem Suchpfad enthält der Subbaum T_j dann nur noch weniger als die Hälfte aller Peers mit $|T_j| < \frac{n}{2}$.

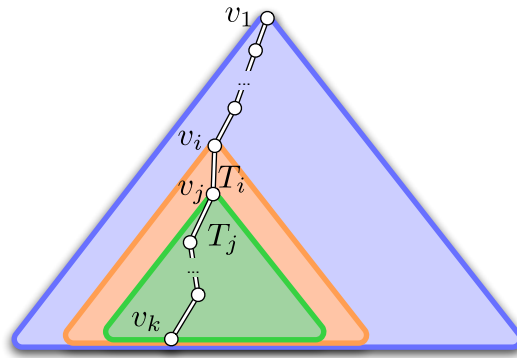


Abbildung 2.14.: Halbierung der Anzahl Peers auf Lookup-Pfad

Der Lookup beginnt bei einem beliebigen anfragenden Peer p in der Wurzel des Baums v_1 . Kennt Peer p durch seinen normalen Pfad den Suchpfad P_{net} nicht vollständig, dann leitet er die Anfrage über einen BranchLink auf dem Suchpfad weiter an einen zufälligen Peer p' . In diesem Zweig T_{hop} sind höchstens noch die $(n - 1)$ anderen Peers ohne p enthalten. In dieser Teilmenge T_{hop} aus allen Peers fehlen zunächst einmal die Peers, deren normaler Pfad am wenigsten mit dem Suchpfad übereinstimmt

2.2. Algorithmen

und dabei zuletzt die Peers aus T_i , so dass der Anteil Peers aus T_i hier über die Hälfte ausmacht mit $|T_i \cap T_{hop}| / |T_{hop}| \geq \frac{1}{2}$. Da der BranchLink auf einen echt zufälligen Peer p' in der Menge T_{hop} zeigt, liegt die Wahrscheinlichkeit für $p' \in T_i$ bei $\geq \frac{1}{2}$. Wurde beim ersten Hop kein Peer in T_i erreicht, so ist die Wahrscheinlichkeit bei einem zweiten Versuch mit einem zweiten Hop auch $\geq \frac{1}{2}$, da der Lookup in jedem Hop einen Fortschritt erzielt und die Menge involvierter Peers jedes mal kleiner wird. So ist die bedingte Wahrscheinlichkeit genau mit einem Hop T_i zu erreichen $\geq \frac{1}{2}$, mit 2 Hops $\geq \frac{1}{4}$, und mit k Hops $\geq 2^{-k}$. Daraus folgt der Erwartungswert

$$E[\text{\#Hops um Peer in } T_i \text{ zu erreichen}] \leq \sum_{k=1}^{\frac{n}{2}-1} k \cdot 2^{-k} \leq 2$$

Ist ein Peer in T_i erreicht, so kann mit nur einem weiteren Hop ein Peer in T_j erreicht werden. Denn T_i ist der kleinste Teilbaum mit $|T_i| \geq \frac{n}{2}$ und da beim nächsten Hop die Peer-Anzahl kleiner wird, wird unweigerlich ein Teilbaum mit $|T_j| \leq \frac{n}{2}$ erreicht. Somit werden im Erwartungswert höchstens 3 Hops benötigt um die Anzahl Peers mindestens zu halbieren.

In T_j ist nun die Anzahl Peers halbiert und man kann denselben Schritt auf dem noch verbleibenden Pfad $(v_j, \dots, v_k)$ im Teilbaum T_j noch mal ausführen um die Peer Anzahl rekursiv weiter zu halbieren. Dabei wird die Anzahl Peer in weiteren Rekursionsschritten solange mindestens halbiert bis die Anzahl Peers auf 1 reduziert wurde. Jeder der Rekursionsschritte benötigt im Erwartungswert höchstens 3 Hops.

Nun bleibt noch zu zeigen, dass der Lookup nach $\mathcal{O}(\log n)$ Hops mit hoher Wahrscheinlichkeit beendet ist. Dabei gilt:

$$P[\text{\#Peers ist in 3 Hops halbiert}] \geq \frac{1}{2} + \frac{1}{4} = \frac{3}{4}$$

Unterteilt man den Lookup nun in Sequenzen mit 3 Hops (Rekursionsschritte), dann kann die Analyse in eine Sequenz von unabhängigen Zufallsvariablen $X_1, X_2, \dots, X_{c \cdot \log n}$ aufgeteilt werden. Für jede Zufallsvariable gibt es zwei Ereignisse: ob die Anzahl Peers in dem Schritt halbiert wurde mit Wert 1 oder nicht halbiert wurde mit Wert 0 wobei $P[X_i = 1] = \frac{3}{4}$. Die Zahl erreichter Halbierungen ist dann gegeben mit $X = \sum_{i=1}^{c \cdot \log n} X_i$. Hier kann die Chernoff-Schranke benutzt werden um zu zeigen, dass für $c \geq 5$ mit hoher Wahrscheinlichkeit gilt $X > \log n$. Mit 3 Hops pro Schritt ist dann der Lookup in $3c \cdot \log n$ Hops mit hoher Wahrscheinlichkeit abgeschlossen. Die erwartete Anzahl erfolgreicher Schritte ist dabei

$$E[X] = \frac{3}{4}c \cdot \log n$$

2. Beschreibung des Netzwerks

So ergibt sich für $\delta = 1 - \frac{4}{3c}$ die Chernoff-Schranke zu

$$\begin{aligned} P[X \leq \log n] = P[X \leq (1 - \delta) \cdot E[X]] &\leq e^{-\frac{1}{2} \cdot \left(1 - \frac{4}{3c}\right)^2 \cdot E[X]} \\ &\leq e^{-\frac{1}{2} \cdot \left(\frac{11}{15}\right)^2 \cdot \frac{3}{4} \cdot c \cdot \log n} \\ &\leq n^{-\frac{121}{600}c} \\ &\leq n^{-c'} \end{aligned}$$

Damit ist die Wahrscheinlichkeit für $c \cdot \log n$ Schritte polynominell klein zur Peer Anzahl n . Und damit benötigt die Lookup-Operation mit hoher Wahrscheinlichkeit $3c \cdot \log n$ Hops mit $c \geq 5$. Da hier keine Annahmen über die Baumstruktur getroffen werden, gilt dieser Beweis ebenso für degenerierte Bäume mit linearer Tiefe. $\square$

2.2.3. Suche

Im Gegensatz zum Lookup, der für einen Suchschlüssel genau ein Ergebnis zurückliefert, wird bei der Suche nach einem allgemeineren Suchkriterium gesucht, dass auf eine Menge von Suchschlüsseln zutreffen kann. Hierbei liegt der Fokus auf Bereichsanfragen (alle Elemente von 'a' bis 'm') und der Präfixsuche (alle Elemente, die mit 'd' beginnen). Die große Stärke eines semantischen Netzwerks liegt hierbei nun in der Lokalität der Daten, da diese sortiert abgelegt sind. So muss nur der im Suchschema zutreffende Bereich durchsucht werden und etwa der Bereich 'n-z' kann in einer Bereichsanfrage für den Bereich 'a-m' völlig außer acht gelassen werden.

Hierbei sei erwähnt, dass die Daten entsprechend des Suchkriteriums eindimensional sortiert sein müssen. Sind die Daten beispielsweise alphabetisch sortiert und die Anfrage lautet "alle Schlüssel, die auf 'ung' enden", so müsste auch hier der gesamte Intervall durchsucht werden. Für verschiedene Suchkriterien müssen hier die Daten dann in verschiedenen Sortierungen abgelegt werden, was durch das mehrfache Ablegen der Daten in mehreren Bäumen oder Baumzweigen erreicht werden kann.

Um einen Bereich zu durchsuchen gibt es 2 mögliche Strategien:

Serielle Suche Bei der seriellen Suche wird zuerst ein Lookup nach dem ersten Element durchgeführt ($\mathcal{O}(\log n)$ Nachrichten). Daraufhin wird der Bereich Schritt für Schritt durchlaufen bis das Ende des Bereichs erreicht ist. Hierzu muss der Datenbereich allerdings durch einen Ring verbunden sein, der durchlaufen werden kann. Dann beträgt die Anzahl benötigter Nachrichten $\mathcal{O}(\log n) + |\text{Suchbereich}|$. Speziell bei der Präfix-Suche wobei ein gesamter Teilbaum mit Grad d und Tiefe h durchsucht wird, beträgt die Nachrichtenzahl $\mathcal{O}(\log(n) + d^h)$. Ein Nachteil ist die große Latenz, da all diese Nachrichten in einer Abfolge gesendet werden und die Anzahl Hops in Serie hier der Nachrichtenanzahl entspricht. Die Steuerung der Suche ist einfach, da sie immer von genau einem Peer durchgeführt wird.

Parallele Suche Bei der parallelen Suche wird zunächst die Suchanfrage bis zu einem internen Knoten weitergeleitet, der gerade noch den gesamten Bereich mit dem

Präfix abdeckt. Danach wird die Suchanfrage parallel an alle Kindknoten rekursiv weitergeleitet, die zum Suchbereich gehören. Bei der Präfixsuche in einem Teilbaum mit Grad d und Tiefe h beträgt die Anzahl produzierter Nachrichten hier nur $\mathcal{O}(d^h)$. Die Latenz ist hier allerdings erheblich kleiner mit nur $\mathcal{O}(\log(n))$. Eine Schwierigkeit der parallelen Suche ist die Steuerung, da mehrere Peers parallel die Suche durchführen (z.B. wann ist die Suche zu Ende).

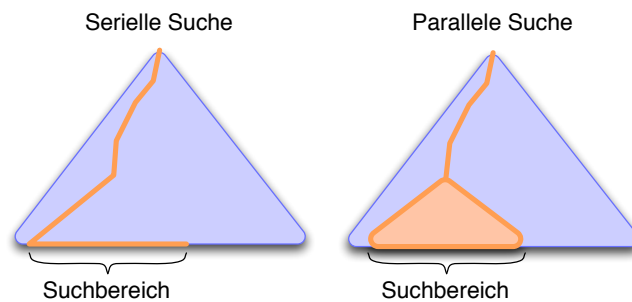


Abbildung 2.15.: Illustration von serieller und paralleler Suchstrategie

Beim Vergleich der Nachrichtenzahl von serieller und paralleler Suche liegt die Vermutung nahe, dass die parallele Suche beim Durchsuchen eines vollständigen Baums der Tiefe h und Baumgrad d und Daten nur auf den Blättern fast doppelt so viele Nachrichten benötigt, da alle inneren Knoten besucht werden müssen. Dies ist allerdings nicht der Fall, da bei der parallelen Suche im Präfixbaum der Vorteil des "Durchrutschens" genutzt werden kann. Abbildung 2.16 zeigt hierzu ein Beispiel. Dabei ist ein Netzwerk-

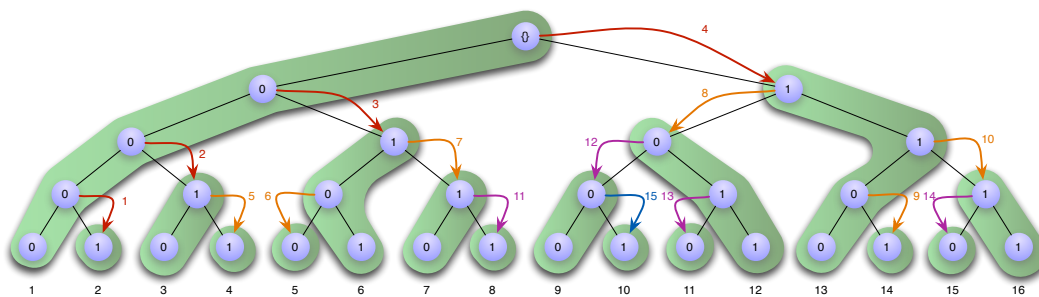


Abbildung 2.16.: Beispiel für parallele Suche im Präfixbaum mit $d = 2$ und $h = 4$

baum dargestellt, der vollständig durchsucht werden soll. Die Suche wird von mehreren Peers parallel durchgeführt und der grüne Hintergrund markiert den normalen Pfad des suchenden Peers ab dem Präfix für den er die Suche begonnen hat. So beginnt ein Peer initial die Suche in der Wurzel in Tiefe 0 und kennt schon ein Blatt auf seinem normalen Pfad '0000' in Tiefe 4. In allen anderen Tiefen 0-3 kann er die Suche parallel an andere Peers weiterleiten. Diese Eigenschaft haben natürlich alle anderen Peers allgemein auch: erhält ein Peer die Anfrage für den Präfix mit Länge i , so kennt er ein Blatt

2. Beschreibung des Netzwerks

in Tiefe h und kann die Suche in den Tiefen $[i, h - 1]$ parallel weiterleiten. Dies führt zu einer rekursiv definierten Nachrichtenzahl für die Suche in einem Subbaum der Tiefe h und Baumgrad $d \geq 2$ von

$$t(h) = \begin{cases} \sum_{i=0}^{h-1} (d-1) \cdot (1 + t(i)) & \text{für } h \geq 1, \\ 0 & \text{für } h = 0. \end{cases}$$

Die Summe im Rekursionsschritt kann man dabei mit den $(h - 1)$ verschiedenen Baumtiefen gleichsetzen, in denen ein Peer die Nachricht weiterleiten muss. Dabei gibt es jeweils $(d - 1)$ Kindknoten in denen er selbst nicht mitspielt und für jeden eine Nachricht an einen Peer dort weiterleiten muss. Diese Peers startend mit Präfix der Länge $(i + 1)$ produzieren dann jeweils rekursiv für einen noch übrig bleibenden Baum der Tiefe $(h - (i + 1))$ die zusätzlichen Nachrichten $t(h - (i + 1))$. Hierbei kann man umrechnen $\sum_{i=0}^{h-1} t(h - (i + 1)) = \sum_{i=0}^{h-1} t(i)$. Die Rekursion terminiert für einen noch ausstehenden Baum der Tiefe $h = 0$, da der Peer im gesamten Baum teilnimmt, der nur aus der Wurzel besteht. Daher kann der Peer hier die Anfrage komplett bearbeiten und es werden keine weiteren Nachrichten erzeugt. Die rekursiv definierte Nachrichtenzahl $t(h)$ hat eine obere Schranke beschrieben im folgenden Lemma 2.1.

Lemma 2.1 *In einem vollständigen Baum der Tiefe h und Baumgrad $d \geq 2$ benötigt die parallele Suche höchstens $d^h - (d - 1)$ Nachrichten.*

Beweis: Der Beweis erfolgt per Induktion. Die Induktionsbehauptung lautet:

$$t(h) \leq d^h - (d - 1)$$

Beim Induktionsanfang für einen Baum der Tiefe $h = 0$ muss keine Nachricht verschickt werden. Dies stimmt mit der Induktionsbehauptung überein:

$$t(0) \leq d^0 - (d - 1) = 0$$

Der Induktionsschritt für $h \rightarrow h + 1$ lautet:

$$\begin{aligned} t(h+1) &= \sum_{i=0}^h (d-1) \cdot (1 + t(i)) \\ &= (d-1) \cdot (1 + t(h)) + \underbrace{\sum_{i=0}^{h-1} (d-1) \cdot (1 + t(i))}_{t(h)} \\ &= (d-1) + d \cdot \underbrace{t(h)}_{\text{IB}} \\ &\leq (d-1) + d \cdot (d^h - (d-1)) \\ &= d^{h+1} - (d-1)^2 \\ &\leq d^{h+1} - (d-1) \end{aligned} \quad \square$$

Beim vollständigen Baum beträgt die Anzahl Blätter d^h . Um diesen bei der seriellen Suche zu durchlaufen sind mindestens $d^h - 1$ Schritte und damit auch Nachrichten nötig. Die parallele Suche hingegen benötigt nur h Schritte und ungefähr gleich viele Nachrichten $t(h) \leq d^h - (d - 1)$.

Bei der Bereichsanfrage hingegen kann die serielle gegenüber der parallelen bis maximal die Hälfte weniger Nachrichten benötigen. Den schlimmsten Fall stellt hier die Suche nach einem Bereich mit 2 Elementen dar, z.B. mit den Pfaden 01^{h-1} und 10^{h-1} . Die serielle Suche muss hier nur einen Lookup nach dem erst genannten Pfad durchführen und darauf hin einen Schritt zum zweiten Pfad unternehmen, so dass bis zu $(h + 1)$ Nachrichten produziert werden. Die parallele Suche hingegen teilt die Suche bereits in der Wurzel auf, da sich die ersten beiden IDs unterscheiden. Von da an kann die Suche nicht weiter aufgeteilt werden und es werden praktisch 2 Lookups der beiden Pfade durchgeführt, so dass hier bis zu $2h$ Nachrichten erzeugt werden. Dies sind fast doppelt so viele. Für größere Suchbereiche wird dieser Faktor allerdings kleiner und fällt bis auf 1 beim vollständigen Baum.

In 3nuts wird ausschließlich die parallele Suche benutzt. Da sich die Nachrichtenzahl bei beiden Strategien nur um einen Faktor unterscheiden gibt bei der Abwägung die Laufzeit den entscheidenden Ausschlag. Hier ist die logarithmische Laufzeit um Größenordnungen schneller als die lineare Laufzeit der seriellen Suche. Des Weiteren durchschreitet die parallele Suche die gleichen Pfade wie der Lookup und kann daher ebenfalls von den lokal-optimierten Routingstrukturen profitieren und so nochmals die Laufzeit optimieren. Bei der seriellen Suche hingegen gibt es im Ring um den Datenbereich herum für jeden Zeiger nur genau einen nächsten möglichen Peer, so dass hier die Routingstruktur festgelegt ist und nicht weiter optimiert werden kann. Daher u.a. ist ein Ring in 3nuts nicht notwendig.

Die in 3nuts verwendete Suchfunktion wird in Algorithmus 2 dargestellt. Darin wird eine Präfixsuche umgesetzt, die den gesamten Subbaum eines Präfix-Pfads *key* beginnend bei Peer *p* durchsucht und alle Subpfade als Ergebnis zurückliefert. Zunächst muss hierbei ein Peer erreicht werden, der in dem Präfix *key* teilnimmt. Hierzu wird genau wie bei der Lookup-Operation die Such-Operation rekursiv ausgeführt und die Suche über die BranchLink's hinweg weitergeleitet (Zeile 1-2). Nimmt Peer *p* dann im Suchpfad *key* teil, so kann *key* als eins von möglicherweise mehreren Ergebnissen zurückgegeben werden (Zeile 11). Ist der aktuelle Pfad *key* abgearbeitet, dann kann die Suche auf die direkten Kindknoten ausgeweitet werden. Dazu ruft der Peer bei sich selbst für jeden Pfad der Kindknoten die Suchfunktion rekursiv auf (Zeile 13). Ist dabei ein Kindknoten in seiner Baumsicht weiter enthalten, so kann er dort die Anfrage rekursiv selbst weiter beantworten. Andernfalls hat er für den Kindknoten (*key* $\circ$ *id*) einen BranchLink und leitet die Suche darüber weiter (Zeile 2). Hierdurch kommt dann die Parallelität zu Stande, da die Schleife in Zeile 12 zur Weiterleitung an mehrere Peers führen kann, die dann gleichzeitig verschiedene Teilbäume durchsuchen können. Der Algorithmus wird noch durch den Fall verkompliziert, dass von langen Pfaden die Endungen ohne weitere Abzweige nicht im Netzwerkbaum publiziert werden. Die Endung wird dann vom verantwortlichen Peer im letzten publizierten Baumknoten auf

2. Beschreibung des Netzwerks

Algorithm 2 Suche(Pfad key) bei Peer p

```
1: if  $p$  hat BranchLink auf Peer  $p'$ , der mit  $key$  längeren Präfix teilt then
2:   Leite Suche( $key$ ) weiter an  $p'$ ;
3: else
4:    $v \leftarrow$  letzter Knoten von Pfad  $key$  in Baumsicht von  $p$ ;
5:    $p' \leftarrow$  verantwortlicher Peer in Knoten  $v$ ;
6:   if  $p$  hat Pfad  $key$  in Baumsicht then
7:      $V_{ID} \leftarrow$  IDs aller direkten Kindknoten von  $v$ ;
8:     if  $V_{ID} = \emptyset \wedge p \neq p'$  then
9:       Leite Suche( $key$ ) weiter an  $p'$ ;
10:    else
11:      return  $key$ ;
12:      for each  $id \in V_{ID}$  do
13:        Suche( $key \circ id$ );
14:      end for
15:    end if
16:  else
17:    if  $p \neq p'$  then
18:      Leite Suche( $key$ ) weiter an  $p'$ ;
19:    end if
20:  end if
21: end if
```

dem Pfad lokal verwaltet, wo die Suche dann auch fortgesetzt werden muss. Dafür gibt es 2 Fälle: Peer p nimmt in key teil, kennt darunter aber keine Kindknoten (Zeile 9) oder Peer p nimmt nicht im Suchpfad key teil und kennt auch keinen weiteren BranchLink auf dem Pfad (Zeile 18).

Den Algorithmus der Präfixsuche kann man auch zu einer Bereichsanfrage für einen Bereich $[key_1, key_2]$ erweitern. Dazu wird der Algorithmus für den längsten gemeinsamen Präfix von key_1 und key_2 aufgerufen. Damit wird nur noch der kleinste Teilbaum durchsucht der den Suchbereich $[key_1, key_2]$ vollständig abdeckt. Um dies weiter auf den Suchbereich reduzieren, darf die Rekursion in Zeile 13 nur fortgesetzt werden falls $(key \circ id) \in [key_1, key_2]$.

Implementierung

Die Implementierung von 3nuts erfolgte in der Programmiersprache *Java* und ist kompatibel zur Version 1.5. Das Projekt erfordert neben dem Standard-*JDK* keine zusätzlichen Pakete. Lediglich ein Paket zum PDF-Export von Grafiken beim Testen/Messen wird hier benutzt, dass allerdings zum Compilieren nicht erforderlich ist. Das Paket wird dynamisch geladen und ist es nicht im Verzeichnis `lib/` enthalten, so kann es in der Testumgebung mit entsprechenden Anweisungen ausgewählt werden.

3nuts ist nur ein Werkzeug und keine eigenständige Applikation. Dementsprechend kann ein 3nuts Peer auch nicht selbstständig als Anwendung gestartet werden. Eine konkrete Anwendung wie z.B. zum "Filesharing" wurde hier nicht implementiert. Stattdessen gibt es eine Testumgebung, die ein ganzes 3nuts Netzwerk erzeugen kann um zum Testen Beispieldaten in diesem Netzwerk zu verwalten. Diese kann mit dem Skript `run_3nuts` gestartet werden und dient ebenso als Plattform für alle angestellten Messungen in Kapitel 4. Auf die Implementierung wird hier nicht eingegangen.

3.1. Grundgerüst

Dieser Abschnitt führt alle allgemein implementierten Werkzeuge ein, die in 3nuts benutzt werden. Dies umfasst die Netzwerkkommunikation 3.1.1, Zeitsteuerung mit Threads 3.1.2, Zufallsnetzwerk-Peers 3.1.3 sowie gewichtete Hashtabellen 3.1.4.

3.1.1. Communication

In 3nuts wird die eigentliche Netzwerkschicht durch ein abstraktes Kommunikationsmodell mit asynchronem Nachrichtenaustausch von der Overlay-Netzwerkschicht abgekapselt. Jeder Teilnehmer (z.B. Peer) mit Schnittstelle `Participant` ist dabei mit einer `Communication` verbunden. Will er eine Nachricht versenden, so erzeugt er diese auf Objektebene vom Typ `Message` und versendet diese durch Aufruf der Methode `Communication.send(Message)`. Der Rest der Übermittlung bis ins Netzwerk erfolgt intern in der `Communication`. Wird umgekehrt eine Nachricht aus dem Netzwerk für einen

3. Implementierung

Teilnehmer empfangen, so ruft die **Communication** beim Empfänger die Methode **Participant.handleMessage(Message)** auf, so dass dieser die eingehende Nachricht im selben Thread asynchron verarbeiten kann.

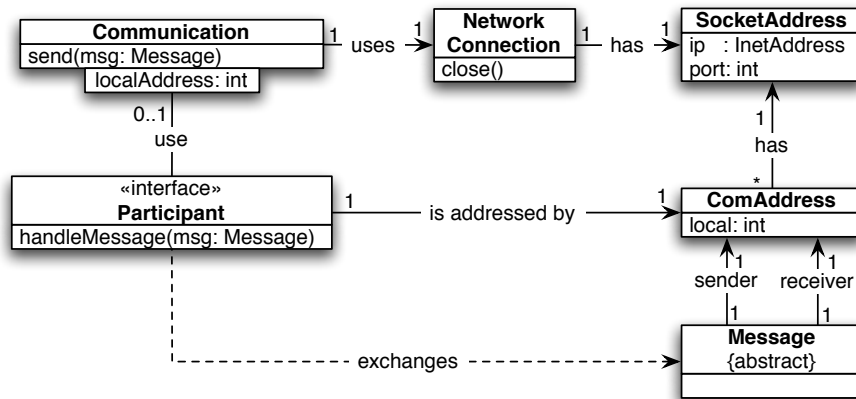


Abbildung 3.1.: Klassendiagramm der *Communication* zum asynchronen Nachrichtenaustausch im Netzwerk

Eine **Communication** enthält intern genau eine Netzwerkverbindung erreichbar durch einen Port auf dem Rechner. Diese Netzwerkverbindung kann von mehreren Teilnehmer zusammen genutzt werden. Dazu erhält jeder der Teilnehmer eine interne lokale Adresse, über die er eindeutig identifiziert wird und hat so eine so genannte **ComAddress** bestehend aus IP, Port und lokaler Adresse. Solch eine Zieladresse muss dann in jeder Nachricht angegeben werden. Wie wir später sehen werden, benutzt ein 3nuts Peer gleich mehrere unabhängig implementierte Zufallsnetzwerk-Peers um in den Baumknoten teilzunehmen. All diese Peers benutzen zusammen eine **Communication**. So wird für alle Peers immer nur ein Port benötigt, der dann z.B. in einer Firewall freigegeben werden muss. Neben den Peers kann es natürlich noch weitere Services geben, die die **Communication** mitbenutzen.

Intern besteht eine **Communication** aus einem eingehenden und ausgehenden Kanal mit einer Pipeline-Architektur (Abbildung 3.2). Beide Pipelines sind durch Elemente erweiterbar um den Nachrichtenstrom zu verarbeiten und zu manipulieren (Z.B. Empfangsbestätigung für Nachrichten, Nachrichtenzähler). In beiden Pipelines erfolgt ein Schnitt zwischen der objektorientierten Overlay-Netzwerkebene und der Byte-Daten Netzwerkschicht. Dazwischen müssen die Daten umgewandelt werden. Die Serialisierung ist speziell und optimiert für jeden Nachrichtentyp implementiert um so die Netzwerklast zu minimieren. Die Nachrichten der Peers haben in der Regel einen recht einfache Struktur und erfordern daher keine allgemeinen Methoden wie die Umwandlung in XML oder die Java-Serialisierung, welche das Datenvolumen um ein Vielfaches unnötig vergrößern. Um die Byte-Daten einer eingehenden Nachricht wieder in ein Nachrichten-Objekt umzuwandeln, wird der Nachrichtentyp hier durch ein erstes Byte als ID im Header definiert. Jeder Teilnehmer-Typ wie z.B. 3nuts Peer kann eine be-

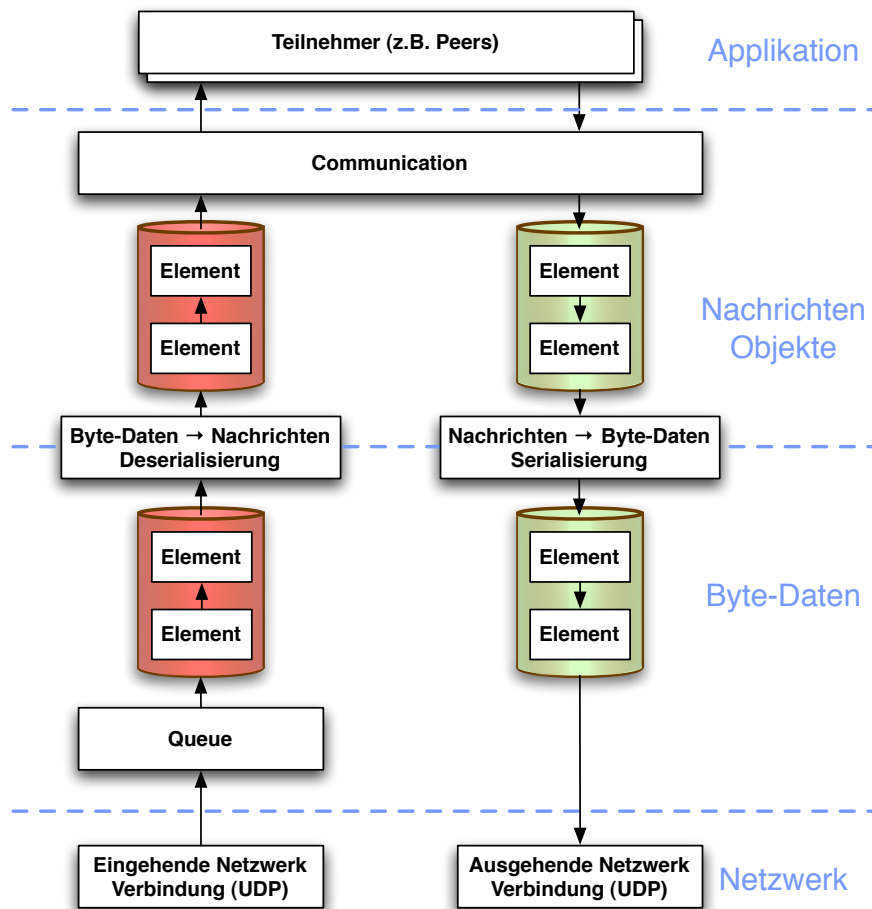


Abbildung 3.2.: Pipeline-Architektur der Communication zum Nachrichtenaustausch

stimmte Menge an verschiedenen Nachrichtentypen verarbeiten und erstellt dafür eine Abbildung (Nachrichten-ID $\mapsto$ Klassentyp). Anhand dieser Abbildung kann die Communication dann ein Objekt der Nachrichtenklasse instanziiieren. Die Nachrichtenklasse enthält dann die weitere Beschreibung der Daten und wie diese aus den Byte-Daten in die Objekt-Daten übertragen werden können.

Die eingehende Pipeline einer Communication enthält zu Anfang eine Queue, in die alle empfangenen Daten aus dem Netzwerk gelegt werden. Ein Pool mit einer festen Anzahl Threads entfernt diese dann wieder aus der Queue um sie parallel zu verarbeiten. Die Nachrichten durchlaufen dann parallel die eingehende Pipeline um zum Schluss ebenfalls parallel beim Empfänger im selben Thread verarbeitet werden zu können.

3. Implementierung

3.1.2. TaskScheduler

Ein Peer muss zeitlich geplante Aufgaben ausführen. Dazu gehört etwa das rundenbasierte Anstoßen der Pointer-Push&Pull-Operation. Oder erwartet ein Peer eine Antwort auf eine Nachricht, dann muss der Peer nach einem Timeout eine Fehlerroutine starten. All diese zeitlichen Aufgaben werden in 3nuts in einem allgemeinen Konstrukt den so genannten **TaskScheduler** verwaltet. Die Grundidee ist folgende: Jede Aufgabe, die ein Peer in der Zukunft ausführen möchte, fügt er zusammen mit der geplanten Ausführungszeit einem **TaskScheduler** hinzu. Ist der Zeitpunkt erreicht, so wird die Aufgabe – hier als **Task** bezeichnet – in einem Thread des **TaskScheduler** ausgeführt.

In 3nuts sind alle Prozesse eines Peers vollständig parallel und asynchron definiert. Immer dann wenn synchron eine **Sleep**-Anweisung benutzt werden müsste, wird die Aufgabe in einen **TaskScheduler** gelegt um sie zeitversetzt fortzuführen. Auf diese Weise sind die eigentlichen Threads abgekapselt von der Peer-Implementierung. So wird zunächst im Peer alles vollständig parallel implementiert und in den **TaskScheduler** kann dann unabhängig davon spezifiziert werden mit wie vielen Threads diese Tasks des Peers dann ausgeführt werden. Dies ist besonders ein Vorteil bei den Messungen vorgestellt in Kapitel 4. Denn ein **TaskScheduler** ist ein allgemeines Konstrukt, welches von mehreren Peers – 3nuts sowie Zufallsnetzwerk Peers – gemeinsam genutzt werden kann. Und so können ohne weiteres viele tausend autonome Instanzen parallel und unabhängig Aufgaben verzahnt ausführen, ohne dass dafür ebenso viele Threads erzeugt werden müssten was technisch nicht möglich wäre. Für 3nuts gibt es eine Singleton-Klasse `p2p.freenuts.TaskSchedulerProvider`, die für alle Belange in 3nuts die **TaskScheduler** zentral bereitstellt. Dort können die **TaskScheduler** individuell den Anforderungen der Anwendung angepasst werden (z.B. Anzahl Threads).

Das Klassendiagramm in Abbildung 3.3 zeigt den Aufbau eines **TaskScheduler**. Er

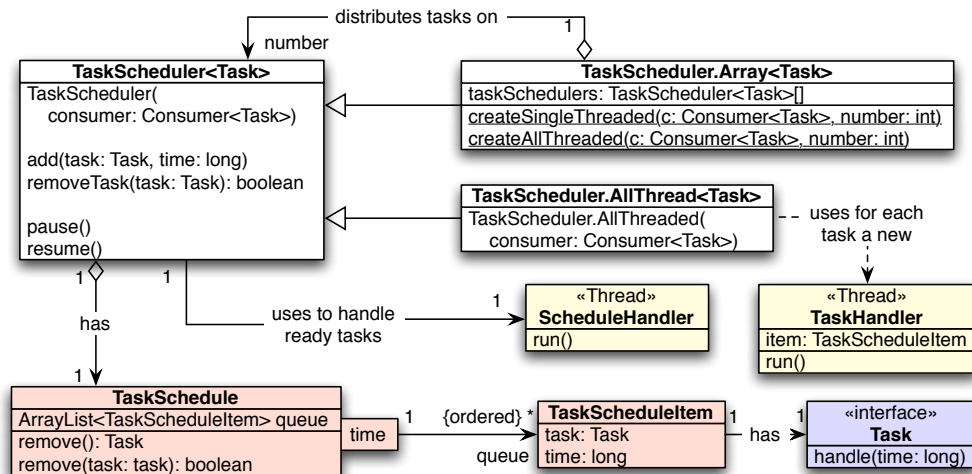


Abbildung 3.3.: Klassendiagramm von **TaskScheduler**

enthält einen Zeitplan `TaskSchedule` über alle hinzugefügten Aufgaben `TaskScheduler`letern, die zu einer bestimmten Zeit ausgeführt werden sollen. Um hier alle Aufgaben zeitlich geordnet anstoßen zu können, müssen alle Aufgaben aufsteigend nach der Ausführungszeit sortiert werden. Ein Thread namens `ScheduleHandler` arbeitet diesen Zeitplan dann ab und wartet jeweils auf die Ausführungszeit des ersten Tasks im Zeitplan um ihn auszuführen. Zur Ausführung eines Tasks wird dann die Methode `handle()` im Task aufgerufen. Hier wird also praktisch die zeitlich verzahnte Ausführung konkurrierender Threads durch eine lineare Ausführung einer sortierten Liste ersetzt. Und jeder `TaskScheduler` hat dabei genau einen Thread zur nebenläufigen Ausführung. Um mehrere Threads für einen `TaskScheduler` zu nutzen, kann einfach ein so genannter `TaskScheduler.Array` erzeugt werden, der nach außen wie ein `TaskScheduler` aussieht, intern aber aus einem Array von mehreren `TaskScheduler` besteht, von denen jeder einen Thread hat. Die Tasks werden dann gleichmäßig auf das Array verteilt. Die Unterteilung eines Zeitplan in mehrere kleinere Zeitpläne im Array sorgt zusätzlich für eine bessere Performanz, da jeder neue Task zeitlich sortiert eingefügt werden muss.

Ein `TaskScheduler` bietet speziell für die Messung eine Pausefunktion, die mit den Methoden `pause()` und `resume()` gesteuert werden kann. Hier kann der Zeitplan eingefroren werden um ihn dann zeitversetzt nach der Pause fortzusetzen.

3.1.3. Zufallsnetzwerk-Peers

Alle 3nuts Peers im gleichen Baumknoten unterhalten zusammen ein Zufallsnetzwerk (vgl. Kapitel 2.1.2). Um in einem Zufallsnetzwerk teilzunehmen, benutzt ein 3nuts Peer einen unabhängig implementierten Zufallsnetzwerk-Peer mit Namen `RandomNetworkPeer`. Da ein Peer z.B. in seinem normalen Pfad gleich in mehreren Zufallsnetzwerken aller Baumknoten teilnehmen muss, unterhält er für jeden dieser Baumknoten jeweils einen Zufallsnetzwerk-Peer. All diese Zufallsnetzwerk-Peers benutzen die `Communication` des 3nuts Peers mit, so dass alle die gleiche IP und Port haben.

Ein Zufallsnetzwerk-Peer benutzt die `Pointer-Push&Pull-Operation` um die Netzwerkstruktur zufällig zu halten und dabei gleichzeitig die Routingtabelle zu warten. Zur Ausführung der Operation werden `TaskScheduler` benutzt: zum einen um die Operation rundenbasiert anzustoßen und zum anderen um eine Fehlerroutine einzuleiten falls auf eine Anfrage in der Operation keine Antwort folgt und daraus resultierend ein fehlerhafter Eintrag in der Routingtabelle korrigiert werden muss. Neben den Nachbarn mit regulärem Ausgrad im Feld `neighbors` unterhält jeder Peer noch eine zweite Routingtabelle mit alten Nachbarn `oldNeighbors` zur Erhöhung der Netzwerkrobustheit. Die Einträge sind vom Typ `RandomNetworkAddress` und enthalten neben der Adresse des Zufallsnetzwerk-Peers mit IP, Port und lokaler Adresse zusätzlich die lokale Adresse des 3nuts Peers, welcher die Anwendung des Zufallsnetzwerk-Peers darstellt. So kann man damit sowohl den Zufallsnetzwerk-Peer also auch den 3nuts Peer kontaktieren.

Jede `Pointer-Push&Pull-Operation` besteht aus eine Anfrage und einer Antwort-Nachricht. Als so genannter `BearerListener` kann die Anwendung des Peers (hier ein 3nuts Peer)

3. Implementierung

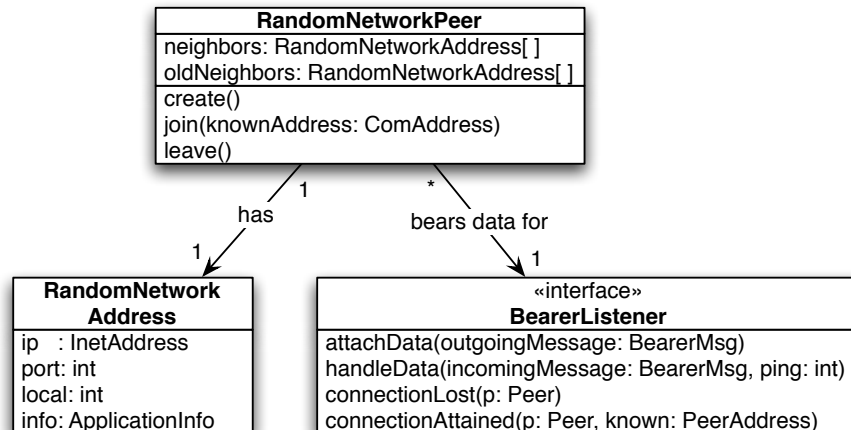


Abbildung 3.4.: Klassendiagramm eines Zufallsnetzwerk-Peers

den Nachrichten der Pointer-Push&Pull-Operation zusätzliche Informationen anheften (hier Bauminformationen) um diese an zufällige Nachbarn im Zufallsnetzwerk zu schicken. So können Informationen ohne zusätzliche Netzwerkpakete und damit geringer Netzwerklast im Netzwerk verbreitet werden. Kann dabei der Ping zum Sender ermittelt werden, so wird dieser entsprechend mit den Bauminformationen übergeben. Über das Interface **BearerListener** wird die Anwendung (hier 3nuts Peer) zusätzlich informiert, ob der Peer alleine im Zufallsnetzwerk ist oder nicht (Abhängig davon ob ein 3nuts Peer alleine in einem Baumknoten ist, erweitert er seinen normalen Pfad).

Als weitere Funktionsmerkmale bietet ein Zufallsnetzwerk-Peer noch eine Lookup-Operation zur Suche in einem Random-Walk mit begrenzter Länge sowie einen Broadcast-Service im Zufallsnetzwerk mit *Randomized Rumor Spreading* [KSSV00]. Beides wird in 3nuts allerdings momentan nicht benutzt.

3.1.4. Gewichtete verteilte Hashtabellen

Heterogenes Hashing (DHHT) vorgestellt in Kapitel 2.1.1 ist im Wesentlichen eine Funktion zur Abbildung von Datenelementen auf Knoten. Diese Funktion ist in der Klasse **WeightedDistributedHashing** in der Methode `getAssignedNode()` implementiert. Sie gibt für die Eingabeparameter eines Datenelements vom Typ **DataElement** und einer Menge gewichteter Knoten vom Typ **Node** den zugewiesenen Knoten zurück. Hier gibt es eine Version mit und ohne Double-Hashing von Datenelement und Knoten.

Es gibt unterschiedliche Implementierungen für Schlüssel/Hashwerte der Knoten bzw. Datenelemente mit 1-8 Byte bzw. 16 Byte Länge. Über die statische Funktion `get(byteNumber: byte)` erhält man die speziellen Versionen. Der Unterschied liegt hier in der Rechnung, wobei die Logarithmus-Funktionen der 1-8 Byte Werte auf den einfachen Java-Datentyp **double** berechnet werden können. Die 16 Byte = 128 Bit Version kann dies nicht und erhält hier eine spezielle Berechnung wobei unter anderem die

Logarithmus-Berechnung der gewichteten Hashfunktion durch eine Reihenentwicklung erfolgt, die auf der Potenzreihenentwicklung des *Areatangens Hyperbolicus* beruht [Wik09b]. Zusätzlich gibt es hier noch eine Methode `getHashValue(key:byte[]):byte[]`

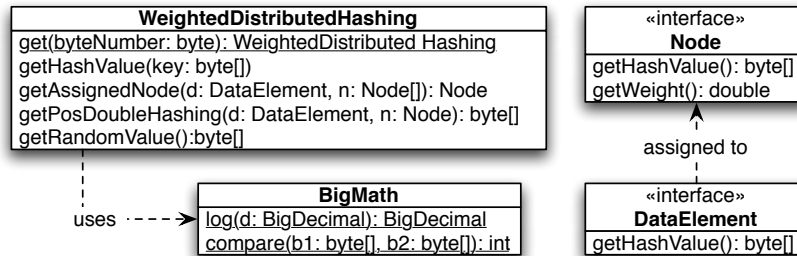


Abbildung 3.5.: Klassendiagramm für Funktionsbibliothek DHHT

um für die Schlüssel von Knoten und Datenelementen einen Hashwert mit der kryptographischen Hashfunktion MD5 (*Message-Digest algorithm 5*) zu erzeugen.

3.2. Struktur eines 3nuts Peer

Abbildung 3.6 zeigt in einer Übersicht der Struktur eines 3nuts Peers, wobei **FreeNutsPeer** die Hauptklasse zur Initialisierung darstellt. Darin können die Hauptoperationen zur Verbindung zu einem Netzwerk sowie die Suchoperationen aufgerufen werden. Ebenso können hier die Gewichte der Pfade aus dem gewichteten Datenbaum gesetzt werden, für den dann der Netzwerkbaum erstellt wird. Zur Konfiguration eines 3nuts Peers wird diesem ein Konfigurationsobjekt mit dem Typ **FreeNutsProperties** übergeben, dass alle Eigenschaften enthält. Diese Konfiguration ist global in einem gesamten 3nuts Netzwerk gleich. Sie wird vom Peer, der das Netzwerk mit der Methode `create()` erzeugt, vorgegeben und beim Beitritt anderer Peers mit der `join()`-Methode übergeben. Jeder Peer hat einen 128-bit Schlüssel gesetzt in der **FreeNutsID**, durch den das gewichtete Hashing bestimmt wird. Entsprechend muss jeder Peer hier einen eindeutigen Schlüssel haben.

Zur Kommunikation zwischen den 3nuts Peers wird eine **Communication** benutzt. Jeder Zufallsnetzwerk-Peer (**RandomNetworkPeer**), der von einem 3nuts Peer initialisiert wird um in einem Baumknoten teilzunehmen, verwendet diese **Communication** mit.

3.2.1. Baumstruktur

Jeder Peer hat eine eingeschränkte Sicht auf den Netzwerkbaum. Diese beginnt in jedem Fall bei der Wurzel und der **FreeNutsPeer** hat zunächst eine Referenz auf den Wurzel-Baumknoten im Feld `treeRoot`, sobald er mit einem Netzwerk verbunden ist. In jedem Baumknoten, in dem ein 3nuts Peer teilnimmt so auch in der Wurzel, unterhält er einen **TrunkNode**, der ihm eine "Verbindung" in diesen Baumknoten gewährt.

3. Implementierung

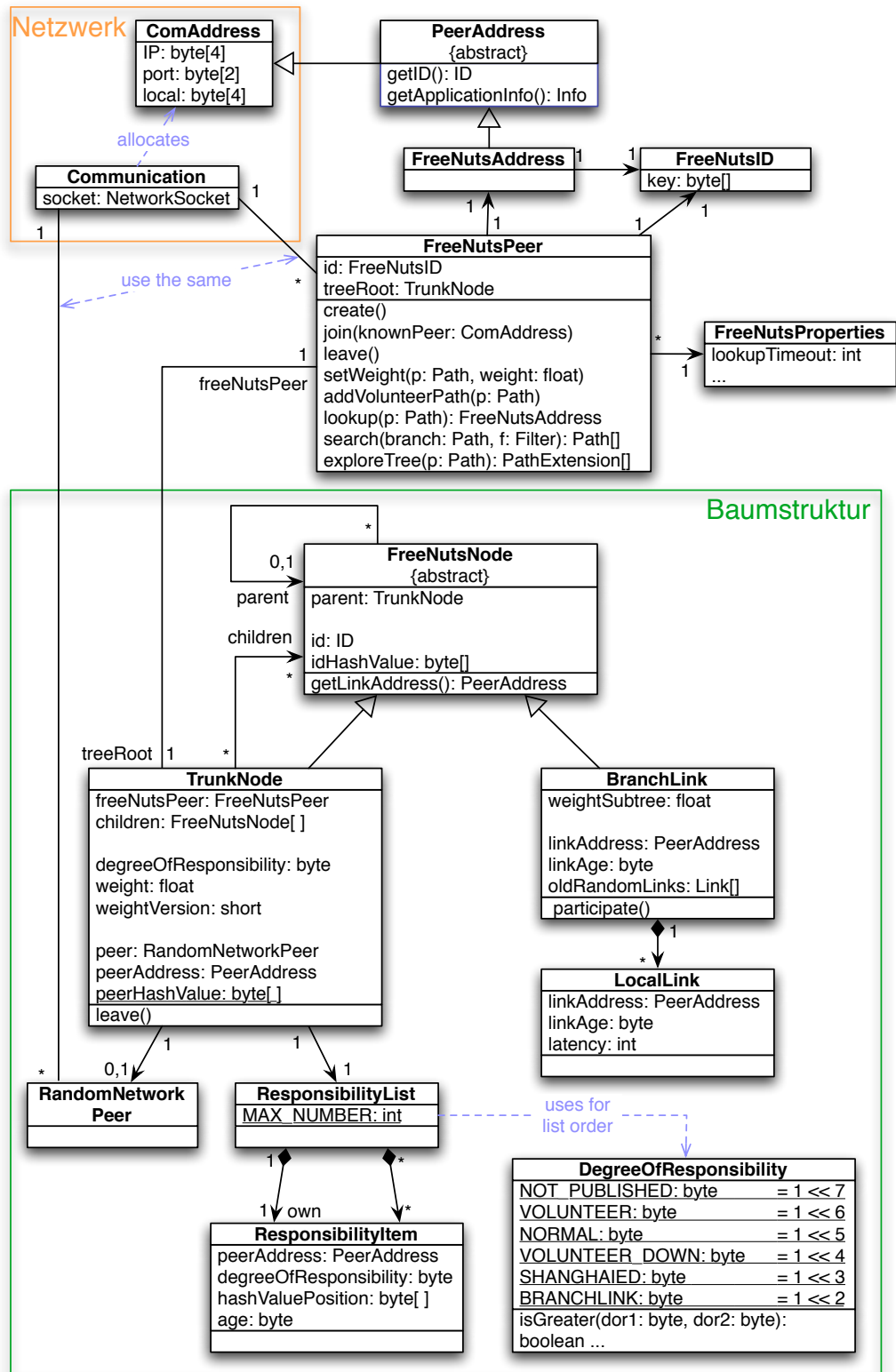


Abbildung 3.6.: Klassendiagramm eines 3nuts Peers

Ein `TrunkNode` enthält dann eine Liste mit allen direkten Kind-Baumknoten im Feld `children`, die den allgemeinen Typ `FreeNutsNode` haben. Nimmt der 3nuts Peer wieder in dem Kindknoten teil (z.B. normaler Pfad), dann ist dieser `FreeNutsNode` wieder ein spezieller `TrunkNode`. Ansonsten ist es ein `BranchLink`, der gleich den gesamten Baumzweig repräsentiert, in dem der Peer zwar nicht teilnimmt aber er dorthin mit dem `BranchLink` routen kann. Jeder `FreeNutsNode` wird durch eine eindeutige ID identifiziert z.B 0 oder 1 im Binärbaum.

Der gewichtete Baum ist wie folgt dargestellt: ein `TrunkNode` repräsentiert nur einen einzelnen Baumknoten und enthält im Feld `weight` daher nur das Gewicht des einzelnen Baumknotens. Dagegen repräsentiert ein `BranchLink` einen gesamten Baumzweig und enthält deshalb das Gewicht des gesamten Subbaums in `weightSubtree`, welches sich durch die Summe der Einzelgewichte errechnet. So kann ein 3nuts Peer immer alle Gewichte gleich ob `TrunkNode` oder `BranchLink` rekursiv aufaddieren und so das Gesamtgewicht eines Teilbaums bestimmen.

Alle 3nuts Peers im gleichen Baumknoten tauschen sich über ein Zufallsnetzwerk aus, wozu jeder 3nuts Peer einen `RandomNetworkPeer` in jedem `TrunkNode` unterhält, der im Netzwerk publiziert ist. Jeder Peer hat in einem `TrunkNode` einen Verantwortungsgrad `degreeOfResponsibility` (z.B. normaler Pfad). Er führt in jedem `TrunkNode` eine Verantwortlichkeitsliste vom Typ `ResponsibilityList`, worüber er den Verantwortlichen des Baumknotens ermitteln kann. In dieser Liste sind dann die Top-Verantwortlichen sortiert mit einem `ResponsibilityItem` enthalten, welche den Verantwortungsgrad, den Hashwert im DHHT und die Adresse des Peers enthalten. Die Applikation des Verantwortlichen darf dann das Gewicht im `TrunkNode` setzen und da sich dieses über die Zeit ändern kann, wird es versioniert im Feld `weightVersion`.

Die Aufgabe eines `BranchLink` ist maßgeblich einen Link auf einen zufälligen Peer in diesem Baumzweig bereitzustellen. Hierzu gibt es im `BranchLink` selbst zunächst einmal einen zufälligen Link. Dieser Link wird durch eine Liste `oldRandomLinks` für die Robustheit erweitert. Daneben stehen dem Peer noch eine Liste von Latenz-optimierten Links mit Typ `LocalLink` zur Verfügung. Die Liste ist nach der Latenz sortiert und hat eine begrenzte Länge. Der Link mit der geringsten Latenz wird fürs Routing in den Baumzweig genutzt. Da die Links in 3nuts nicht zwingend aktiv überprüft werden, sind diese mit einem Alter versehen (Feld `age`). Diese werden rundenbasiert von 0 bis 128 hochgezählt wobei ein Peer bei seinen eigenen Links immer fest Alter 0 hat und ein fremder Link bei 128 gelöscht wird. Der Intervall ist dabei an den Pointer-Push&Pull Intervall gekoppelt, in dem die Peers ihre eigenen Links mit Alter 0 publizieren.

Ein 3nuts Peer muss im gewichteten Datenbaum einen normalen Pfad wählen. Dies geschieht inkrementell beginnend in der Wurzel. Durch die Teilnahme in einem Baumknoten (initial der Wurzel) wird dieser über das Zufallsnetzwerk mit einem `BranchLink` auf jeden Kindknoten versorgt. Mit DHHT wählt der Peer dann den nächsten Baumknoten auf seinem normalen Pfad und nimmt in dem Zweig teil, indem er im `BranchLink` die Methode `participate()` aufruft und der `BranchLink` in einen `TrunkNode` umgewandelt wird (Umgekehrt kann er einen `TrunkNode` mit der Methode `leave()` wieder verlassen, der dann wieder zu einem `BranchLink` wird). Durch die Teilnahme mit ei-

3. Implementierung

nem TrunkNode wird der Peer dann direkt wieder mit den Kindknoten dort versorgt. Die Informationen werden dazu direkt in der Join-Methode im Zufallsnetzwerk des Baumknotens übertragen. Er kann den normalen Pfad weiter rekursiv aufbauen, solange bis er am Ende des Pfads alleine ist oder der Netzwerkbaum nicht tiefer geht.

3.2.2. Änderungen im Baum

Der gewichtete Datenbaum ändert sich ständig etwa durch neu hinzugefügte Daten. Der Netzwerkbaum muss sich diesen Änderungen anpassen. Genau dazu küren die Peers in einem Baumknoten einen Verantwortlichen, der dann auf Applikationsebene das Gewicht des Baumknotens bestimmt etwa durch die Daten, die auf dem Pfad liegen. Der Verantwortliche 3nuts Peer publiziert dann über das Zufallsnetzwerk des Baumknotens das Gewicht, so dass es alle teilnehmenden Peers kennen. Peers, die selbst nicht in einem Baumknoten teilnehmen, haben irgendwo einen BranchLink auf dem Pfad zum Baumknoten, in den das Gewicht des Baumknotens in das Gesamtgewicht des Zweigs eingerechnet ist. So gibt es ständig neue Pfade im Baum und die Gewichte in den einzelnen Baumknoten können sich auch ändern. Abbildung 3.7 zeigt in einer Übersicht, wie die Änderungen zum einen von der Applikation und zum anderen von anderen Peers aus dem Netzwerk kommen. Die Verbindung zu anderen Peers im Zufallsnetzwerk eines Baumknoten entscheidet dabei etwa darüber, wie weit der normale Pfad erweitert werden muss, damit jeder Peer am Ende alleine ist.



Abbildung 3.7.: Ereignisse die zur Änderung des Baums im 3nuts Peer führen

Die Peers aktualisieren den Netzwerkbaum, indem sie Baumdaten an die Nachrichten der Pointer-Push&Pull-Operation in den Zufallsnetzwerken anhängen und sich so zufällig austauschen. Die über ein Zufallsnetzwerk ausgetauschten Informationen sind in der Klasse `FreeNutsNodeInfo` zusammengefasst (Abbildung 3.8). Diese beziehen sich auf den Baumknoten des Zufallsnetzwerks und enthalten daher die Verantwortlichkeitsliste, das versionierte Gewicht des Baumknotens und einen `BranchLink` für jeden direkten Kindknoten übertragen in der Form eines `BranchLinkInfo`-Objekts. So wird für jeden Abzweig immer nur das aufsummierte Gewicht des Subbaums übertragen. Die Zuordnung der einzelnen Abzweige erfolgt über die eindeutige ID (z.B. 0,1 im Binärbaum). Nimmt ein Peer in einem Kindknoten selbst mit einem `TrunkNode` teil, so enthält die `BranchLinkInfo` einen Link auf sich selbst und sonst den zufälligen Link den er selbst als `BranchLink` kennt.

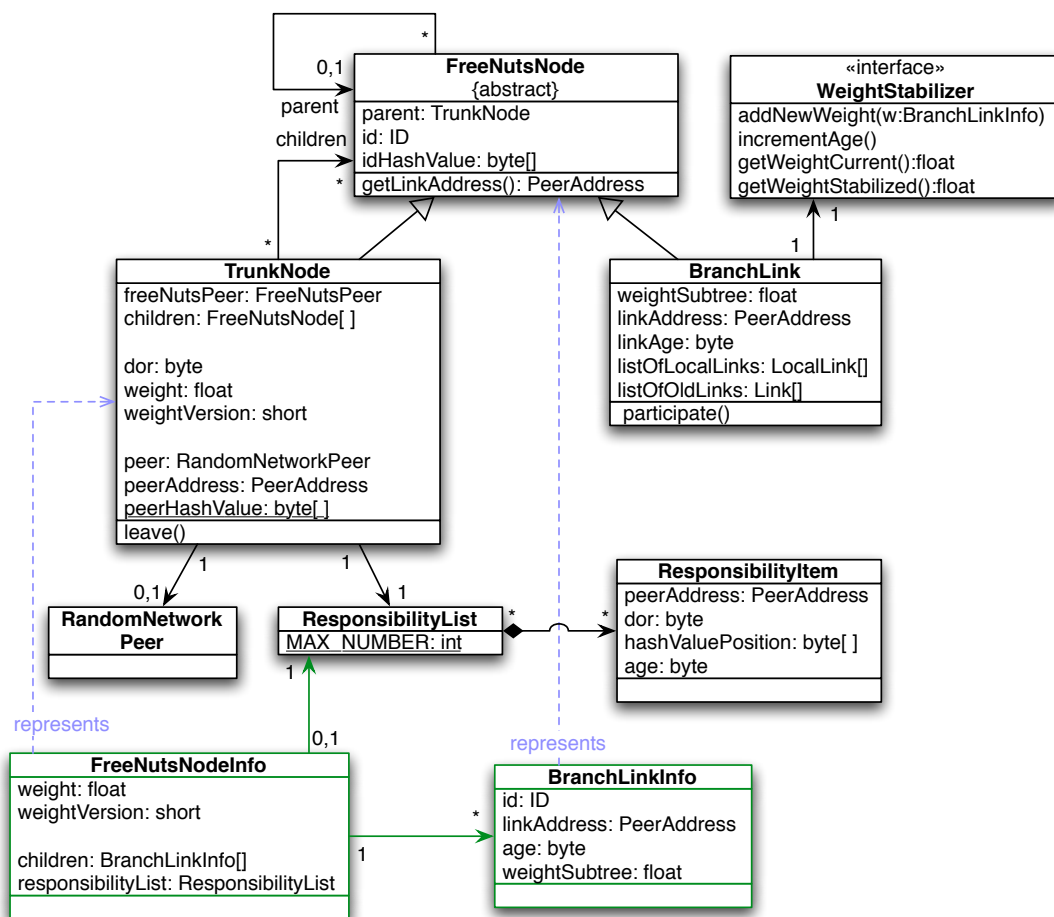


Abbildung 3.8.: Klassendiagramm über Änderungen der Baumstruktur eines 3nuts Peer

Erhält ein Peer solch eine `FreeNutsNodeInfo`, so verschmelzt er zunächst die beiden Verantwortlichkeitslisten miteinander. So wird die Liste um die Einträge erweitert, von

3. Implementierung

denen der Peer noch nicht gehört hat. Wird dabei festgestellt, dass der Sender den richtigen Verantwortlichen hatte, der am Anfang der Liste steht und damit das Gewicht des Baumknotens im Netzwerk setzen darf, dann wird auch das Gewicht des Baumknotens entsprechend der Versionsnummer aktualisiert. Durch verschiedene Ausbreitungsgeschwindigkeiten mit der Pointer-Push&Pull-Operation ist der korrekte Verantwortliche nicht überall gleich schnell gegeben.

Nimmt ein Peer in einem Kindknoten selbst teil, für den er eine `BranchLinkInfo` erhält, so kennt er dort das Gewicht selbst viel genauer und kann das aufaddierte Gewicht für den gesamten Zweig nicht verwerten. Der Link der `BranchLinkInfo` wird hier allerdings genutzt um den Zusammenhang des Zufallsnetzwerks herzustellen, falls dies durch Inkonsistenten im Netzwerk parallel für den gleich Baumknoten erzeugt wurde oder durch eine starke Peer-Fluktuation auseinander gefallen ist.

Hat ein Peer selbst einen `BranchLink` für eine erhaltene `BranchLinkInfo`, dann übernimmt er die enthaltenen Informationen mit dem zufälligen Link, falls das Alter des Links geringer ist. So überschreiben neue Links von aktiven Peers im Baumzweig alte Links von Peers die nicht mehr drin sind und ihre Links nicht mehr neu publizieren. Konnte in der Pointer-Push&Pull-Operation der Ping zum Sender ermittelt werden und die `BranchLinkInfo` enthält einen Link auf den Sender, so wird dieser Link als Latenz-optimierter Link einsortiert. Falls der zufällige Link übernommen wird, wird auch das übermittelte Gewicht für den Baumzweig gesetzt. Allerdings kann dieses Gewicht entsprechend der Netzwerkdynamik stark schwanken. Denn alle Peers, die solch ein aufsummiertes Gewicht für einen Subbaum erzeugen, nehmen alle in verschiedenen normalen Pfaden dieses Subbaums teil. So haben sie für die Pfade wo sie teilnehmen sehr aktuelle Gewichtsinformationen. Daneben haben sie aber im Subbaum genauso selbst `BranchLinks` die weniger aktuelle Gewichte enthalten. So kann sich zum gleichen Zeitpunkt das von verschiedenen Peers errechnete Gewicht für den gleichen Baumzweig stark unterscheiden, weil der eine Peer von dem einen Teil der Daten gehört hat und von anderen dafür noch nicht. Aus diesem Grund wird in jedem `BranchLink` versucht das Gewicht mit einem sogenannten `WeightStabilizer` zu stabilisieren. Hier wird aus der Serie der letzten erhaltenen Baumgewichte ein stabilisiertes Gewicht erzeugt. Es wird praktisch eine "zweite Meinung" eingeholt. Die momentan genutzte Implementierung speichert eine einstellbare Menge Gewichte (z.B. 4) aufsteigend sortiert nach dem Alter des Links im `BranchLink` und bildet daraus den Mittelwert. Einzelne abweichende Gewichte haben hier keinen so starken Einfluss auf den Mittelwert. An andere Peers wird das zufällige nicht stabilisierte Gewicht des `BranchLink` weitergegeben, damit bei der Stabilisierung jeder Peer den gleichen Prozess durchmacht. Dadurch verzögert die Mittelwertbildung eine wirkliche Gewichtsänderung auch nicht stark.

Das stabilisierte Gewicht wird bei der Ermittlung des normalen Pfads eingesetzt. Denn würde hier das Gewicht des Baums stark schwanken, dann würden die Peers ständig ihren normalen Pfad an die schwankende Gewichtsänderung anpassen. Dabei käme es dann dazu, dass für Daten verantwortliche Peers ihren normalen Pfad ändern, die Verantwortung der Daten dadurch verlieren, und das Gewicht neu bestimmt werden muss. Durch die neue Ermittlung schwankt dann das Gewicht größer was im

Kreislauf die Peers stärker schwanken lässt. Wird dabei die Dynamik im Netzwerk zu groß, kann das Ganze soweit aufschwingen, dass es in die Zweige keine gültigen Links mehr gibt und kein Routing mehr möglich ist. Dieses dynamische Problem wird unter anderem durch stabilisierte Gewichte unterbunden.

Ein weiteres effektives Mittel ist dabei, die Anpassung des normalen Pfads auf einen zeitlichen Mindestintervall zu begrenzen anstatt bei jeder Gewichtsänderung im Baum. Dies folgt auch daraus, dass ein Peer bei einer Pointer-Push&Pull-Operation nicht den aktuellen Stand über den gesamten Baum erhält, sondern er sich aus vielen Operationen eine Baumsicht aufbauen muss, wobei es jeweils nur partielle Änderungen gibt. Die zeitliche Begrenzung wird mit einem TaskScheduler realisiert. Stellt ein 3nuts Peer eine Baumänderung fest, so legt er einen Task im TaskScheduler an, in dem dann zeitlich verzögert der normale Pfad überprüft und gegebenenfalls angepasst wird. Und bevor ein bestehender Task eines Peers nicht ausgeführt ist, kann kein neuer erzeugt werden. Damit nicht alle Peers im Netzwerk gleichzeitig ihren normalen Pfad ändern wird die Verzögerungszeit aus den FreeNutsProperties mit einem zufälligen Faktor $k \in [1, 2]$ multipliziert um die Operationen über den Intervall zu verteilen.

Wegen der hohen Bedeutung eines stabilen Netzwerkbaums ist eine enge Verbindung von einem 3nuts Peer mit der Anwendung erforderlich, die den Peer zur Verwaltung von Daten benutzt. Ein sich ändernder Datenbaum führt zur Anpassung der normalen Pfade der Peers und damit zu einer Verschiebung von Verantwortlichkeiten. Bei solch einer Änderung sollte der neue Verantwortliche für einen Pfad möglichst sofort alle Daten des Pfads auf Applikationsebene zur Verfügung haben. Denn die Daten benötigt die Applikation zum Beantworten von Anfragen. Außerdem muss die verantwortliche Applikation das Gewicht anhand dieser Daten bestimmen und im 3nuts Peer für den Pfad setzen. Dazu muss die Anwendung die Schnittstelle FreeNutsAppli-

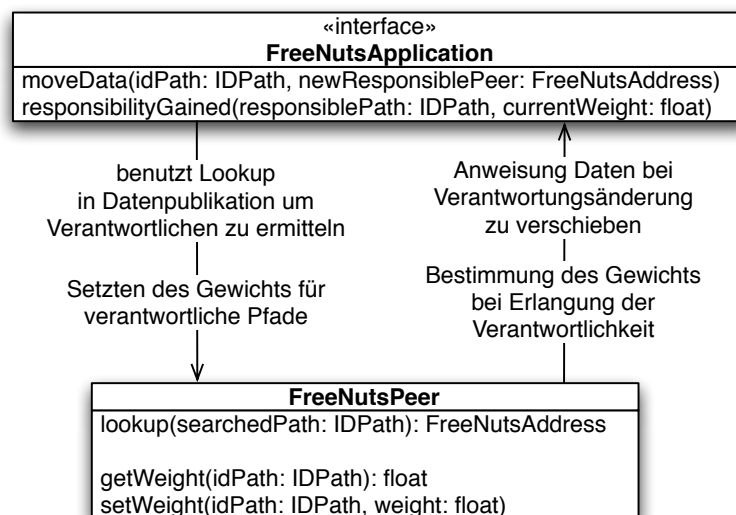


Abbildung 3.9.: Verbindung von 3nuts Peer mit Applikation, die den Peer benutzt

3. Implementierung

cation implementieren und ist darüber mit dem Peer assoziiert wie in Abbildung 3.9 dargestellt. Das Vorgehen bei der Datenverwaltung ist dann folgendes: Eine Applikation ruft im 3nuts Peer für die eigenen Daten die `lookup()`-Operation auf, erhält die Verantwortlichen für die Pfade und transferiert die Daten dorthin. Eine Verantwortliche Applikation ermittelt dann anhand der Daten (können mehrere für den gleichen Pfad sein) ein Gewicht und setzt dieses im 3nuts Peer mit der Operation `setWeight()`. Würde dies in einem statischen Netzwerk jede Applikation einmal machen, so wären alle Daten an ihrem richtigen Platz und in der Suchoperation des Netzwerks verfügbar. In einem dynamischen Netzwerk hingegen ändern sich die Verantwortlichkeiten, weil z.B. neue Peers in den Pfaden beitreten und Verantwortlichkeiten übernehmen. Um darauf zu reagieren, weist ein 3nuts Peer jedes mal die Applikation mit der Operation `responsibilityGained()` an, dass diese in einem Pfad die Verantwortung erlangt hat und dafür ein Gewicht bestimmen muss. Verliert eine Applikation umgekehrt die Verantwortung, so weist der 3nuts Peer mit der Operation `moveData()` die Applikation an, die Daten direkt zum neuen Verantwortlichen zu verschieben. Da es eine Zeit dauert, die Daten vom alten zum neuen Verantwortlichen eines Pfads zu transferieren, muss hier eine bestimmte Verzögerung bei der Bestimmung des Gewichts eingebaut werden, damit kein falsches Gewicht wegen fehlender Daten bestimmt wird.

3.3. Netzwerkoperationen

Zu den Netzwerkoperationen zählen zum einen die Erstellung eines Netzwerks mit `create()` zum Erzeugen eines Netzwerks, `join()` zum Beitreten zu einem bestehendem Netzwerk und `leave()` zum Verlassen des Netzwerks. Dabei erzeugt die `create()`-Methode einfach die Wurzel für einen neuen Netzwerkbaum im Peer und erstellt darin ein neues Zufallsnetzwerk ebenfalls mit der `create()`-Operation. Die `leave()`-Operation baut die Baumstruktur des Peers dann wieder vollständig ab. War der Peer hier für Daten verantwortlich, so werden diese mit der Methode `FreeNutsApplication.moveData()` zu den "Vize-Verantwortlichen" weitergegeben, so dass die Daten beim Verlassen des Peers aus dem Netzwerk nicht verloren gehen.

Zum Beitritt in ein Netzwerk mit der `join()`-Operation muss der 3nuts Peer zum einen die Parameter des Netzwerks mit Typ `FreeNutsProperties` in Erfahrung bringen, damit er sich konform zum Netzwerk verhält. Zum anderen muss er sich zum Netzwerkbaum verbinden. Die Operation ist in der Subklasse `FreeNutsPeer.Join` implementiert. Als Parameter wird die Adresse eines Peers aus dem Netzwerk übergeben und der Peer sendet eine Join-Anfrage an diesen Peer. Der andere Peer sendet dann in einer Antwort die `FreeNutsProperties` sowie die Adresse seines Zufallsnetzwerk-Peers in der Wurzel des Netzwerkbaums zurück. So kann sich der Peer mit der Wurzel des Netzwerkbaums verbinden und kann den Rest des Netzwerkbaums dann rekursiv weiter aufbauen wie im letzten Abschnitt 3.2.1 beschrieben. Der Peer nutzt hier einen `TaskScheduler` um nach einem Timeout ohne erhaltene Antwort die Anfrage zu wiederholen.

3.3.1. Lookup

Die Lookup-Operation liefert für einen Pfad die Adresse des verantwortlichen Peers zurück. In dieser verteilten Operation wird eine Anfrage-Nachricht über mehrere Hops bis zum Verantwortlichen geroutet, der dann eine Antwort an den fragenden Peer zurücksendet. Mit der erhaltenen Antwort kann die aufgerufene `lookup()`-Operation im 3nuts Peer synchron beendet werden und liefert die Adresse des Verantwortlichen aus der Antwort-Nachricht zurück. Im Fehlerfall nach einem Timeout terminiert die Operation in einer `Exception`, so dass dieser Fall entsprechend behandelt werden kann.

Die Steuerung der verteilten Operation ist in der Subklasse `FreeNutsPeer.Lookup` organisiert. Bei Initialisierung der Operation erzeugt der anfragende Peer eine Anfrage-Nachricht für den gesuchten Pfad, die er zunächst versucht selbst zu beantworten. Dazu wird in der Baumsicht des Peers der letzte bekannte Baumknoten auf dem Suchpfad ermittelt. Ist dies ein `BranchLink`, so kann der Peer die Anfrage nicht beantworten und leitet die Anfrage über den `BranchLink` weiter zu einem Peer, der in diesem Baumzweig auf dem Suchpfad teilnimmt. Ist der letzte bekannte Baumknoten beim Peer hingegen ein `TrunkNode`, dann nimmt der Peer im gesamten Präfix des Suchpfads teil, der im Netzwerkbaum publiziert ist. Hier gibt es 2 Möglichkeiten: ist der Peer anhand der Verantwortlichkeitsliste für den Baumknoten verantwortlich, dann kann er die Anfrage direkt mit einer Antwort an den anfragenden Peer beantworten. Sonst leitet er die Anfrage an den Verantwortlichen Peer weiter. Der Verantwortliche eines Baumknotens erzeugt immer die Kindknoten im Netzwerk, die nicht jedem Peer im Baumknoten direkt bekannt sind. Der letzte Hop zum Verantwortlichen kann diesen Fehler kompensieren, da der Verantwortliche auf jeden Fall in die Kindknoten falls vorhanden routen kann.

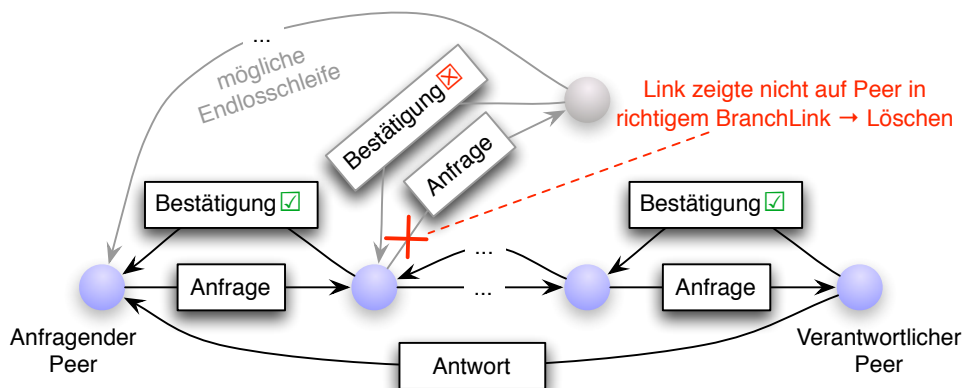


Abbildung 3.10.: Nachrichten im Multi-Hop Routing der Lookup-Operation

Für die Korrektheit der verteilten Operation muss sichergestellt sein, dass in jedem Rekursionsschritt ein Fortschritt erzielt wird und damit die Operation letztendlich terminiert und nicht in einer Endlosschleife landet. In einem dynamischen 3nuts Netzwerk ist dies nicht zwangsläufig gegeben. Denn dadurch, dass die Peers ihren norma-

3. Implementierung

len Pfad anpassen und die Links in den alten Pfaden dann ungültig werden, können die BranchLinks auf falsche Baumzweige zeigen. Die Forderung, dass ein Peer dann mindestens in der Tiefe des BranchLink-Pfads im Suchpfad teilnimmt, gilt dann nicht mehr. Aus diesem Grund wird die Anfrage-Nachricht um die Baumtiefe ergänzt, in der der Hop über einen BranchLink auf dem Suchpfad erfolgte. Nimmt der Empfänger dieser Nachricht nun nicht mindestens in der Tiefe auf dem Suchpfad teil, so darf er die Anfrage nicht bearbeiten. Denn sonst könnte er theoretisch aus einer geringeren Tiefe die Nachricht wieder an den letzten Peer weiterleiten, der dann in einer Endlosschleife wieder den falschen Link benutzt. Aus diesem Grund muss jeder Peer, der eine Anfrage-Nachricht entgegen nimmt, mit einer Bestätigungsnachricht quittieren, ob er den Lookup weiterführen kann (siehe Abb. 3.10). Fällt die Bestätigung negativ aus, dann weiß der Peer auf dem letzten Hop, dass der benutzte Link ungültig ist und eine Fehlerroutine wird angestoßen. Gleiches geschieht, wenn nach einem Timeout keine Bestätigung zurückkommt und der Empfänger der Anfrage das Netzwerk verlassen hat. Um die Fehlerroutine nach dem Timeout anzustoßen, wird ein Task in einem TaskScheduler erzeugt, der gleich wieder abgebrochen wird, falls eine positive Bestätigung erfolgt. In der Fehlerroutine wird zunächst der falsche Link gelöscht, welcher entweder ein BranchLink oder auch der Link auf den Verantwortlichen in der Verantwortlichkeitsliste sein kann. Dann wird die Anfrage einfach noch mal bearbeitet, denn alle Links sind durch eine Liste alter Links zur besseren Robustheit erweitert und einer dieser Links könnte ja funktionieren. Kommt es dabei dann dazu, dass ein Peer keinen Link mehr zum Weiterleiten der Anfrage auf dem Suchpfad hat, dann ist dies gleichbedeutend damit, dass der Peer in der Verantwortungshierarchie für den Suchpfad selbst verantwortlich ist. In dieser Weise wird immer eine Antwort generiert.

Aufbauend auf die Lookup-Operation gibt es noch eine zweite Operation `exploreTree()`, die für einen Suchpfad anstatt dem verantwortlichen Peer eine Liste aller Pfaderweiterungen zurückgibt. Dies sind dann BranchLinks auf jeden Baumzweig unterhalb des Suchpfads mit ID des Zweigs und einem Link auf einen Peer darin. In dieser Operation kann wahlweise auch gleich die Adresse eines Peers in dem Suchpfad übergeben werden, so dass nicht das Multi-Hop Routing des Lookups für den Pfad jedes mal ausgeführt werden muss, sondern direkt in einem Hop dieser Peer im Pfad erreicht wird. Auf diese Weise ist es möglich den Baum sukzessive zu erforschen.

3.3.2. Suche

Die Such-Operation durchsucht den gesamten Subbaum eines Pfads. Ergebnis der Operation sind die Pfade aller Knoten in diesem Subbaum inklusive einer Adresse eines teilnehmenden Peers. Die Such-Operation ist eine Erweiterung der Lookup-Operation mit dem Unterschied, dass sich das Routing im Lookup nur auf einen einzelnen Pfad beschränkt und in der Suche auf alle Pfade eines Subbaums ausweitet. In der Implementierung ist die Suche ebenfalls als Erweiterung des Lookups angelegt. Dies bietet den Vorteil, dass die Mechanismen zum Routing und der Fehlerbehandlung aus dem Lookup in der Suche mitbenutzt werden können.

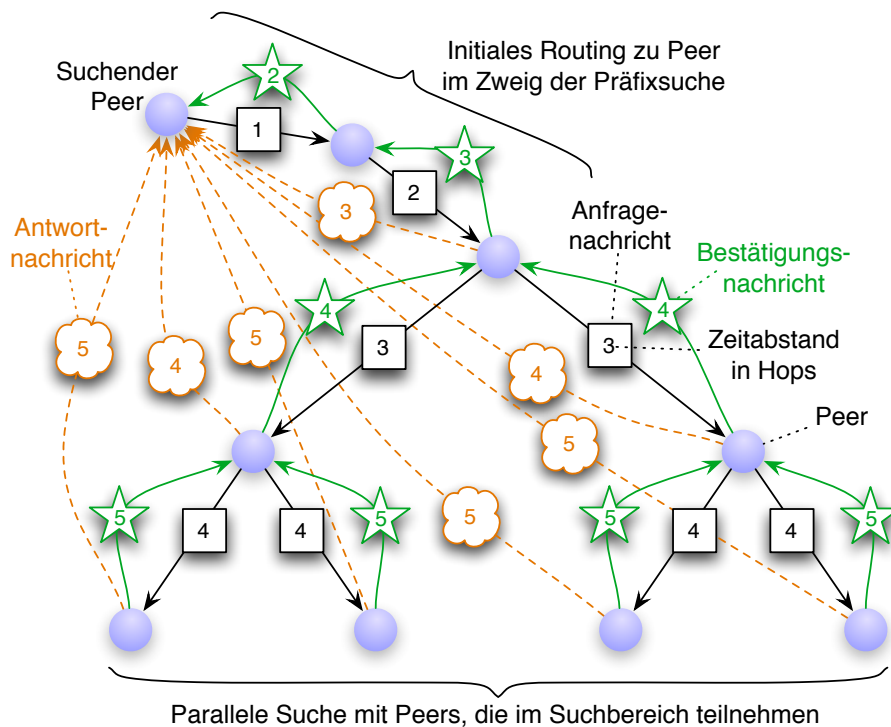


Abbildung 3.11.: Nachrichtenaustausch beim Routing der parallelen Suche

Dies wird so realisiert, dass die Anfrage- bzw. Antwort-Nachrichten der Suche von den Nachrichten im Lookup erben und als Basis die gleichen Informationen wie z.B. den Suchpfad enthalten. So können die Nachrichten zunächst einmal gleich behandelt werden. Bloß die Logik unterscheidet sich hier, in welche Zweige die Anfrage weiter geroutet werden soll und was die Ergebnisse sind. So wird grundsätzlich das gleiche Routing benutzt um wie bei einer Punktanfrage zur Wurzel des Teilbaums zu gelangen, der durchsucht werden soll. Wird die Suche dann in dem Baumknoten auf alle Kindknoten parallel ausgeweitet, so wird die initiale Anfrage-Nachricht kopiert und darin einfach der ursprüngliche Pfad durch den erweiterten Pfad des Kindknotens ersetzt. Diese Folgeanfrage wird dann wieder mit dem Routing einer Punktanfrage verarbeitet. Dabei führt der Peer, der den Kindknoten der Folgeanfrage ja kennt, entweder einen BranchLink-Hop durch oder kann diese Anfrage weiter rekursiv selbst beantworten.

Die Ergebnismenge der Präfixsuche kann durch einen benutzerspezifischen Suchfilter noch weiter eingeschränkt werden. Z.B. möchte man in einem Wörterbuch nicht nach alle Wörtern suchen die mit 'ka' beginnen, sondern nur den eingeschränkten Bereich 'kar'-'kat'. Das Suchfilter-Objekt wird dann an die Suchnachricht angehängt und im Beispiel würde dann die Suche beim Präfix 'ka' beginnen mit dem Suchfilter, der nur alle Pfaderweiterungen 'r-t' zulässt. In der Suchlogik des 3nuts Peer wird der Suchfilter dann jedes mal gefragt, welche Pfade ein Ergebnis sind und in welchen Pfaden weitergesucht werden soll. Auf diese Weise sind hier variabel verschiedene Suchmuster

3. Implementierung

möglich. Auf der Overlay-Netzwerkebene bestehen die Ergebnisse nur aus den Pfaden im Suchraum mit einer Adresse eines Peers dort. Die variable Programmierung des Suchfilters erlaubt hier zusätzlich, dass der Suchfilter auch direkt auf die Applikation des ausführenden Peers zugreift und dort veranlasst z.B. Ergebnisdaten direkt zum suchenden Peer zu schicken. So kann eine zusätzliche Anfrage auf Applikationsebene gespart werden.

Bei einer seriellen Bereichsanfrage ist die Terminierung der Operation einfach: Der Bereich wird von vorne bis hinten durchsucht. Dabei führen die Peers im Bereich nacheinander in einer Kette die Operation aus. Der letzte am Ende des Bereichs kann dann an den anfragenden Peer eine Notiz schicken, dass der gesamte Bereich durchsucht wurde. Der Such-Methodenaufruf im anfragenden Peer kann dann synchron mit der vollständigen Ergebnismenge terminieren. Bei der parallelen Suche steuern viele Peers parallel die Suche. Erreicht dabei eine Anfrage ein Blatt im Netzwerkbaum, ist erst einmal ungewiss, ob die parallelen Anfragen in anderen Teilbäumen ebenfalls schon die Blätter erreicht haben. Man könnte natürlich entgegen der Rekursion von den Blättern bis zur Wurzel des Suchbereichs Bestätigungen zurückschicken, so dass der Peer in der Wurzel des Suchbereichs dann weiß, wann alle Subbäume rekursiv abgearbeitet sind. Dies würde allerdings die Laufzeit verdoppeln. Stattdessen schickt hier jeder Peer, der die Anfrage parallel weiterleitet und auf den Subbaum ausweitet, jeweils eine Antwortnachricht an den anfragenden Peer zurück (siehe Abb. 3.11). Diese Nachricht enthält:

- den Pfad, für den der Peer die Anfrage erhalten hat und der abgearbeitet ist
- alle Ergebnispfade, die bei dem Peer "gefunden" wurden
- alle Pfade in Bearbeitung (Anfrage wurde über BranchLink weitergeleitet)

Die Nachricht wird meist mindestens ein Ergebnis enthalten, da ja jeder Peer auf dem normalen Pfad alleine endet und dort Daten verwaltet. Die Ergebnispfade werden dann beim suchenden Peer für ein Gesamtergebnis gesammelt. Für die Terminierung legt der anfragende Peer eine Liste mit allen Pfaden an, die noch parallel in Bearbeitung sind. Jeder Pfad von dem er eine Antwort erhält, wird dann aus der Liste gestrichen. Und ist die Liste leer, so ist die parallele Suche beendet und der Peer kann direkt den Such-Methodenaufruf synchron beenden und das Gesamtergebnis mit allen gefunden Pfaden zurückgeben. Beim `search()`-Methodenaufruf kann aber auch optional ein `Observer`-Objekt übergeben werden, dass über alle erhaltenen Teilergebnisse der Suche auf dem laufenden gehalten wird. So können die Teilergebnisse schon mal verarbeitet werden und beispielsweise eine Ergebnistabelle in einem Fenster der Applikation mit Zwischenergebnissen aktualisiert werden.

Messungen

In diesem Kapitel werden Messungen im 3nuts Netzwerk vorgestellt, die zur Evaluierung der Such-Performanz, Lastbalancierung und Robustheit des Netzwerks dienen. Der Einsatzbereich von Peer-to-Peer Netzwerken erstreckt sich von einigen wenigen bis Millionen von Peers wobei die Performanz in diesem Bereich skalierbar sein sollte. Daher sollten die Messungen hier auch in Netzwerken "so groß wie möglich" durchgeführt werden. Die Messungen werden hier in 3nuts Netzwerken mit realen Peers durchgeführt mit bis zu $2^{14} = 16.384$ Peers. Da es sich bei einem Peer-to-Peer Netzwerk um eine logische Netzwerkstruktur handelt, ist es hier möglich all diese Peers logisch auf einem physikalischen Computer zu erzeugen und es sind nicht so viele real existierende Peers nötig. Da in einem 3nuts Netzwerk die logische Netzwerkstruktur zur physikalischen Netzwerkstruktur angepasst wird, wird das physikalische Netzwerk – das Internet – wenn nötig simuliert. Die Peers werden alle in einer eigens für 3nuts programmierten Testumgebung erzeugt. Mit allen Peers in einem Prozess und Speicherbereich kann so auf alle Informationen des Netzwerks während der Messungen effizient zugegriffen werden. So können komplexe Messungen vorgenommen werden, da das Netzwerk vollständig erfasst und auch bei Bedarf manipuliert werden kann.

Mit einem vollständigen Netzwerk in nur einer Anwendung müssen sich alle darin befindlichen Peers die Ressourcen dieses Rechners teilen. Dies umfasst den Speicher, CPU-Rechenzeit und eine mögliche parallellläufige Ausführung. Der Testrechner hat eine *Intel Core 2 CPU* mit 4 Kernen je 2.4 GHz Taktung und in der *Java Virtual Machine* können 2,2 Gbyte erweiterter Speicher für den Heap allokiert werden. Der Speicher bestimmt hier maßgeblich die maximale Größe eines testfähigen Netzwerks. 3nuts ist ein zeitabhängiges System. Operationen wie die Pointer-Push&Pull-Operation werden rundenbasiert angestoßen. Nachrichten zwischen Peers erfordern Antworten in festgelegten Timeouts. Die Zeitbasis kann hier einfach herunter geregelt werden, so dass allen Peers im Netzwerk genügend Rechenzeit zur Verfügung steht. Damit rechenintensive Messungen das Verhalten des Netzwerks nicht durch eine zu hohe CPU-Auslastung beeinflussen, wird zur Laufzeit die CPU-Auslastung ermittelt und die Geschwindigkeit der Messungen dynamisch angepasst, so dass dem Netzwerk immer genügend Rechenzeit zur Verfügung steht. Bei 2^{14} Peers ist es weder möglich diese echt parallel

4. Messungen

laufen zu lassen noch können dafür so viele nebenläufige Threads angelegt werden. Wie in Kapitel 3.1.2 beschrieben sind alle internen Prozesse eines 3nuts Peers vollständig parallel und asynchron konzipiert. Die *Communication* stellt dabei eine feste Anzahl Threads zur Verfügung um eingehende Nachrichten zu verarbeiten. Für planendes Verhalten wie etwa das rundenbasierte Anstoßen der Pointer-Push&Pull-Operation stehen *Task Scheduler* mit einer festen Thread-Anzahl zur Verfügung, die die zeitlichen Aufgaben verwalten und zu gegebenen Zeitpunkten die Aufgaben in einem Thread ausführen. Die *Communication* wie auch die *Task Scheduler* können von allen 3nuts Peers gemeinsam verwendet werden, so dass alle Peers skalierbar eine feste Anzahl Threads zusammen benutzen. So kann hier eine Nebenläufigkeit erzeugt werden die durch die fest einstellbare Anzahl Threads beschränkt ist.

Da in einem 3nuts Netzwerk der Datenbaum auf die Netzwerkstruktur abgebildet wird, sind hier Testdaten erforderlich wofür das Netzwerk einen Baum erzeugen kann. Dazu wird jeder Peer zusammen mit einer Applikation erzeugt, die eigene Daten hat und den Peer dazu nutzt um die Daten im Netzwerk zu publizieren. Jede Applikation hat 5 Daten, so dass ein Netzwerk mit $2^{14} = 16.384$ Peers dann 81.920 Daten enthält. Dabei erhalten alle Daten einen eindeutigen Suchpfad im Baum. Anhand unterschiedlicher Datenverteilungen im Baum soll hier untersucht werden, welche Auswirkungen dies auf die Lastbalancierung und Performanz des Netzwerks hat. Des Weiteren wird auch untersucht, welchen Einfluss der Baumgrad – der Grad der einzelnen Knoten im Baum – auf das Netzwerk hat. Konkret werden hier 3 Verteilungen verglichen. Darunter sind 2 Binärbäume, wobei bei einem die Daten zufällig gleichverteilt auf die Teilbäume aufgeteilt sind und im anderen sind die Daten Zipf-verteilt auf die Pfade abgebildet. Die Pfade der Daten erfüllen dabei 2 Eigenschaften: zum einen liegen die Datenelemente nur in den Blättern des Baums. Zum anderen sind die Pfade der Datenelemente immer so lang gewählt, dass der abgebildete Netzwerkbaum niemals die Tiefe der Datenpfade erreicht. Damit kann jeder Peer seinen normalen Pfad so lang wählen, dass er alleine endet und der Netzwerkbaum kann seine maximale Tiefe entfalten. Bei der Gleichverteilung hat jeder Datenpfad die Länge 40 und die Pfade sind zufällig verteilt mit einer entsprechenden Varianz, so dass in den Subbäumen 0 und 1 eines jeden Knotens nicht zwingend exakt die gleiche Menge an Daten enthalten sein muss. Daten in der Praxis sind wahrscheinlich nicht so optimal gleichverteilt. Um dem Rechnung zu tragen wird ein Binärbaum künstlich mit einer Zipf-Verteilung generiert. Dabei gibt es $2^{20} \approx 1\text{Mio.}$ eindeutige binäre Pfade mit Tiefe 20, die anhand einer Zipf-Verteilung unterschiedliche Wahrscheinlichkeiten zugewiesen bekommen. Das Zipfsche Gesetz lautet [Wik09d]: Wenn die Elemente einer Menge nach ihrer Häufigkeit geordnet werden, ist die Wahrscheinlichkeit p ihres Auftretens umgekehrt proportional zur Reihenfolge n :

$$P(X = n) = \frac{1}{n \cdot S_N} \text{ wobei } S_N = \sum_{n=1}^N \frac{1}{n}$$

Hierbei handelt sich um eine diskrete Verteilung $n \in \mathbb{N}$ und somit ist

$$P(X \leq n) = \frac{1}{S_N} \cdot \sum_{i=1}^n \frac{1}{i}$$

Mit der Inversmethode [Wik09a] kann aus gleichverteilten Zufallszahlen eine Zipf-Verteilung erzeugt werden. Hierbei wird eine Gleichverteilung im Intervall $[0, 1]$ benutzt. Der Intervall entspricht hier den möglichen Funktionswerten der Zipf-Verteilung $P(X \leq n) : [1, n] \mapsto [0, 1]$. Mit Hilfe der Umkehrfunktion $P(X \leq n)^{-1} : [0, 1] \mapsto [1, n]$ wird jeder Wert der Gleichverteilung $[0, 1]$ auf eine Zahl im Bereich $[1, n]$ der Zipf-Verteilung abgebildet. Jeder der $n = 2^{20}$ Zahlen wird dann ein zufälliger binärer Pfad der Länge 20 zugewiesen. Auf diese Weise werden dann manche der Pfade häufiger gewählt und andere wiederum sehr selten. Durch die zufällige Zuweisung der Pfade auf die Verteilung ist die Anordnung der häufig vorkommenden Pfade entsprechend auch zufällig. So ist es höchst wahrscheinlich nicht so, dass alle häufig vorkommenden Pfade nur im Präfix 0 zu finden sind, sondern sie werden sich auf den Baum verteilen. Manche Pfade der Zipf-Verteilung werden dann in sehr vielen Datenelementen gewählt werden. Der häufigste Pfad wird im Erwartungswert etwa 5.700 mal gewählt. Das ist jeder 14-te Pfad. Damit alle diese Daten einen eindeutigen Pfad erhalten, wird jeder Präfix der Länge 20 aus der Zipf-Verteilung nochmals durch einen zufälligen Pfad der Länge 32 erweitert. So ist auch sichergestellt, dass der Datenbaum tiefer als der Netzwerkbaum ist. Durch das hohe Datenaufkommen in manchen Präfixen werden dort zur Lastbalancierung mehr Peers sein weshalb der Netzwerkbaum dort auch tiefer sein wird.

Als drittes Beispiel für einen Datenbaum mit hohem Grad werden Wörter aus Wörterbüchern benutzt. Hierbei repräsentiert jeder Baumknoten ein Zeichen in einem Wort und ein Pfad zusammengesetzt dann ein Wort oder einen Präfix davon. Als Alphabet werden hier nur kleine Buchstaben plus Zahlen und Sonderzeichen verwendet, so dass der maximale Baumgrad hier bei 32 liegt. Des Weiteren werden folgende Abkürzungen benutzt: gleichverteilter Binärbaum (kurz *Gleichverteilung*), Zipf-verteilter Binärbaum (kurz *Zipf-Verteilung*), aus Wörterbuch erzeugter Baum mit Zeichen als Baumknoten (kurz *Wörterbuch*).

Bei den folgenden Messungen wurden immer mehrere Messreihen durchgeführt, die dann gemittelt wurden. Dabei gab es keine nennenswerten Fehlerabweichungen.

4.1. Grad

Die erste Messung in Abbildung 4.1 zeigt eine Verteilung darüber, wie viele Zufallsnetzwerkpeers die einzelnen 3nuts Peers erzeugen um in den Baumknoten teilzunehmen. Dies gliedert sich zum einen in den normalen Pfad (Abb. 4.1(a)) wobei die Anzahl gleichbedeutend mit der Länge des normalen Pfads ist. Zum anderen nimmt ein Peer noch zusätzlich Shanghai in weiteren einzelnen Baumknoten teil (Abb. 4.1(b)) um das Routing dorthin zu gewährleisten. Wegen dem hohen Baumgrad beim Wörterbuch

4. Messungen

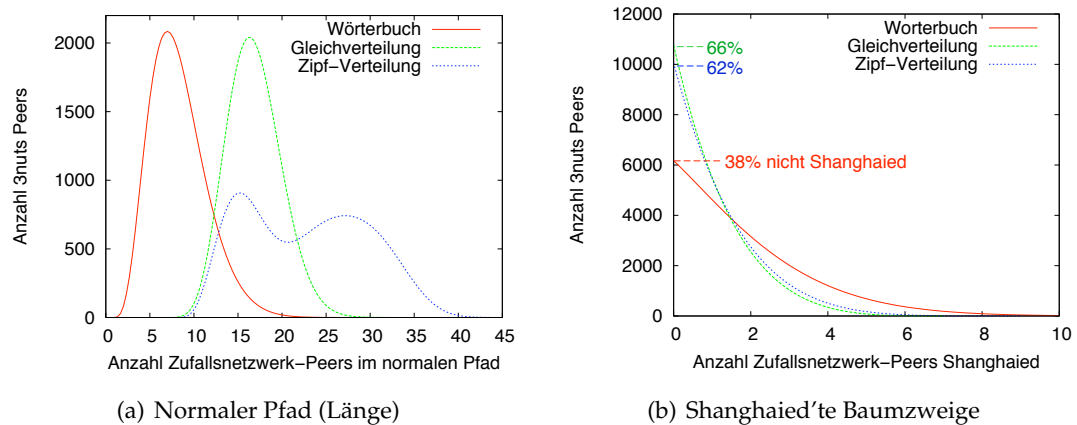


Abbildung 4.1.: Anzahl Zufallsnetzwerk-Peers eines 3nuts Peers bei Netzwerkgröße 2^{14}

hat der resultierende Baum eine hohe Expansion und entsprechend kurze Pfade. Damit haben die Peers hier auch die kürzesten normalen Pfade mit durchschnittlicher Länge 7,7 wie in Abbildung 4.1(a) zu sehen. Bei den Binärbäumen hat die Gleichverteilung die geringere Tiefe mit durchschnittlich 16,7 gegenüber 22,6 in der Zipf-Verteilung. Im gleichverteilten Datenbaum gibt es durch die zufällige Wahl der Datenpfade eine Varianz. Ebenso birgt das gewichtete Hashing bei der Abbildung der Peers auf den Netzwerkbaum eine Varianz, weshalb bei der Gleichverteilung nicht alle Peers einen exakt gleich langen normalen Pfad haben sondern die Länge leicht variiert. Beim Zipf-verteilten Binärbaum ist ein Großteil der Präfixe aus der Zipf-Verteilung wenig bis gar nicht mit Daten besetzt, weshalb dort nur wenige Peers enthalten sind, die sich auch nicht in den Pfaden darunter weiter aufteilen müssen. All diese Peers haben einen normalen Pfad ähnlich der Länge bei der Gleichverteilung, wodurch sich bei Tiefe 15 ein Maximum ergibt (Abb. 4.1(a)). Dann gibt es in der Zipf-Verteilung noch einige wenige Präfixe, die viel mehr Daten beinhalten. Und desto mehr Daten in solch einem Präfix sind, desto mehr Peers kommen wegen der Lastbalancierung in diesen Pfad. Desto mehr Peers in einem Präfix, desto tiefer ist der gleichverteilte Baum darunter, auf den sich die Peers aufteilen. Deshalb steigt die Anzahl Peers in der Pfadlänge nochmals zu einem zweiten Maximum bei 29 an, wo sich dann der Präfix der Zipf-Verteilung mit den meisten Datenelementen befindet. Die rechte Abbildung 4.1(b) zeigt die Anzahl an Shanghaied'ten Baumzweigen pro 3nuts Peer. Die 3nuts Peers gewährleisten dann alleine das Routing in diese Zweige und verwalten die Daten dort. Bei den Binärbäumen sind über 60% der Peers gar nicht Shanghaied. Die Anzahl an Shanghaied'ten Baumzweigen fällt dann schnell ab, so dass hier ein einzelner Peer maximal 6 Zweige gleichzeitig zusätzlich verwalten musste. Beim Wörterbuch fällt die Anzahl Shanghaied'er Pfade wegen dem hohen Baumgrad um Faktor 2,7 größer aus, weshalb die Funktion viel flacher verläuft. Man stelle sich einen Baumgrad von 20 in einem Baumknoten mit 10 Peers vor. Dabei können faktisch nur maximal 10 der Abzweige auf normalen Pfaden

liegen, womit mindestens 10 Zweige Shanghai verwaltet werden müssen. Hierbei ist zu beachten, dass sich die Peers im Baum auf einzelne Subbäume aufteilen, so dass sich weit unten im Baum immer kleinere Mengen von Peers ergeben wie im Beispiel. Bei Binärbäumen mit Grad 2 ist diese Fragmentierung entsprechend kleiner.

Abbildung 4.2 zeigt die Messergebnisse für den Grad der einzelnen Peers. Dabei sind Links von einem Peer zu sich selbst nicht einberechnet, da diese keinen Aufwand zur Aufrechterhaltung erfordern. Nur in Abbildung 4.2(a) werden Mehrfachlinks – mehrere Links zum gleichen Peer – mehrfach gezählt. Mehrfachlinks treten tief im Baum häufiger auf, da die zufälligen Links dort nur auf eine kleine mögliche Menge Peers zeigen kann. Ist der Peer hinter dem Link nicht mehr erreichbar, so kann die Reparatur zu allen Links auf diesen Peer gleichsam angestoßen werden, so dass auch die Darstellung ohne Mehrfachlinks in 4.2(b)-(d) ebenfalls sinnvoll ist. Bei der Messung gibt es 3 Arten von Links für jeden Baumknoten, in dem ein Peer teilnimmt, zu berücksichtigen:

- 4.2(a)-(b): Verantwortlichkeitsliste des Baumknotens
- 4.2(a)-(c): Links im darunterliegenden Zufallsnetzwerk
- 4.2(a)-(d): BranchLinks in Zweige darunter, in denen er nicht teilnimmt

Die Links im Zufallsnetzwerk werden mit der Pointer-Push&Pull-Operation aktiv überprüft. Die Verantwortlichkeitslisten der Baumknoten sowie BranchLinks auf die Baumzweige darunter werden zwischen den Peers in der Pointer-Push&Pull-Operation ausgetauscht und tragen so ebenfalls zur Last bei. Der Grad eines Knotens Peers ist dann ein Vielfaches der Anzahl von Baumknoten, in denen er teilnimmt (Vergleiche Abb. 4.1).

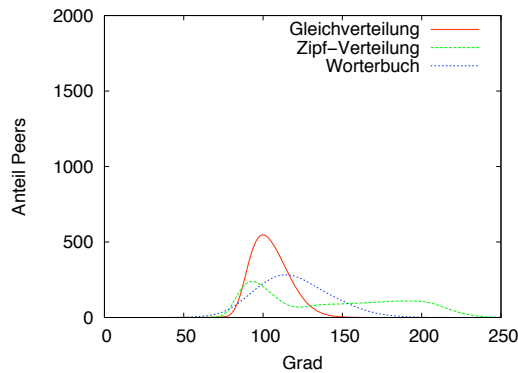
Die Baumstruktur des Wörterbuchs schneidet hierbei trotz des hohen Baumgrads vergleichsweise gut ab, wenn man alle Links (4.2(b)) zählt. Nimmt man eine ungefähr logarithmische Tiefe des Netzwerkbaums an, so ist für die Anzahl Peers n und den Baumgrad b die Länge eines Pfads $\geq \log_b(n)$. In jedem Baumknoten auf dem normalen Pfad benötigt der Peer dann eine Verantwortlichkeitsliste mit Grad r , d Links auf zufällige Nachbarn im darunter liegenden Zufallsnetzwerk, sowie $(b - 1)$ Links auf Abzweige, in denen er selbst nicht ist. Der letzte Knoten auf dem Pfad ist ein Blatt ohne Abzweige. Zusammen ergibt das dann einen Peer-Grad von

$$\text{Peer-Grad}(r, d, b) \geq \frac{r + d + (b - 1)}{\ln(b)} \cdot \ln(n)$$

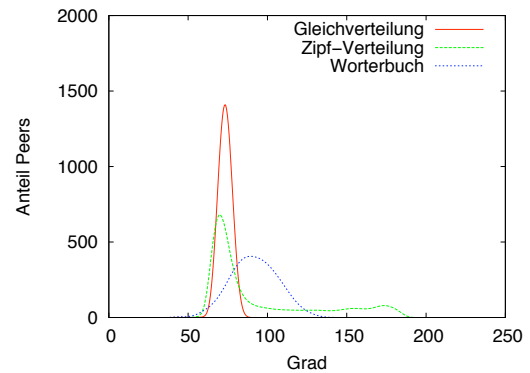
Diese Funktion hat ein Minimum, welches für die gewählten Parameter $r = 3, d = 3$ einen optimalen Baumgrad ≈ 6.1 für einen minimalen Peer-Grad ergibt. Bei zu kleinem Baumgrad ist der normale Pfad zu lang, so dass viele Baumknoten verwaltet werden müssen. Bei zu großem Baumgrad hingegen hat jeder Peer zu viele BranchLinks in den einzelnen Baumknoten.

Ein weiterer Grund für den doch recht geringen Peer-Grad beim Wörterbuch Datenbaum ist der, dass der Datenraum sehr dünn besiedelt ist und damit der maximal mögliche Baumknotengrad praktisch kaum vorkommt. In Abbildung 4.3(a) ist der Baumknotengrad in den einzelnen Baumtiefen im Netzwerkbaum fürs Wörterbuch darge-

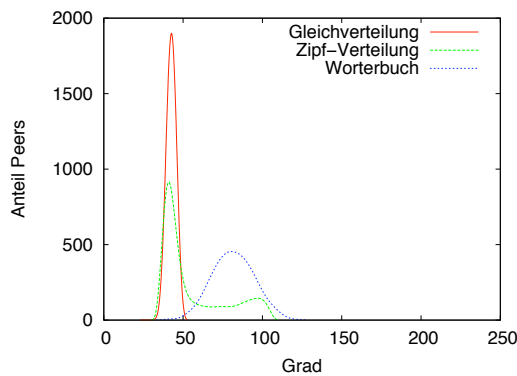
4. Messungen



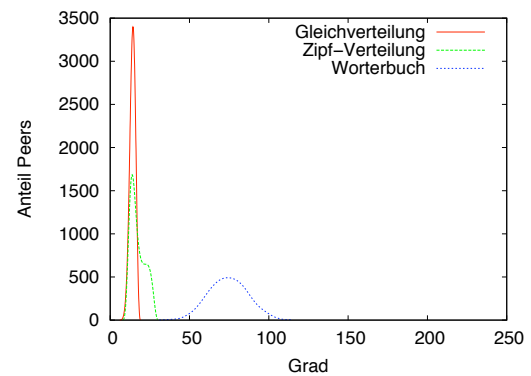
(a) BranchLinks, Zufallsnetzwerknachbarn, Verantwortlichkeitslisten (mit Mehrfachlinks)



(b) BranchLinks, Zufallsnetzwerknachbarn, Verantwortlichkeitslisten (ohne Mehrfachlinks)



(c) BranchLinks, Zufallsnetzwerk-Nachbarn (ohne Mehrfachlinks)



(d) BranchLinks (ohne Mehrfachlinks)

Abbildung 4.2.: Verteilung des Ausgrads der einzelnen Peers bei Netzwerkgröße 2^{14}

stellt. Dabei wird Minimum, Maximum und Mittelung aus allen Baumknoten der einzelnen Peers berechnet, die in dieser Tiefe teilnehmen. Dabei ist es dann so, dass alle Wörter in Tiefe 1 mit jedem möglichen Buchstaben beginnen können und so der Baumknotengrad dort 32 ist. Jedoch fällt der Baumknotengrad mit zunehmender Tiefe recht schnell ab und so gibt es bereits in Tiefe 7 für jeden Präfix im Schnitt nur noch 2 mögliche Zeichen, die diesen erweitern. Der durchschnittliche Baumknotengrad in allen Tiefen beträgt dabei $\approx 4,2$.

Abbildung 4.3(b) zeigt den Vergleich des durchschnittlichen Baumgrads für die einzelnen Baumtypen. Hier zeigt sich, dass bei den Binärbäumen der Baumgrad bereits deutlich ab Tiefe 10 unter 2 fällt, was bedeutet, dass es viele Pfad-Zwischenstücke gibt, die keine weiteren Abzweige haben und es daher keine Expansion gibt. Zum Vergleich mit realen Daten wurde hier noch mal das Wörterbuch binär dargestellt. Dies lieferte noch viel schlechtere Ergebnisse und es wurden wegen der dünnen Besiedlung Baum-

4.2. Entstehungsprozess des Netzwerks

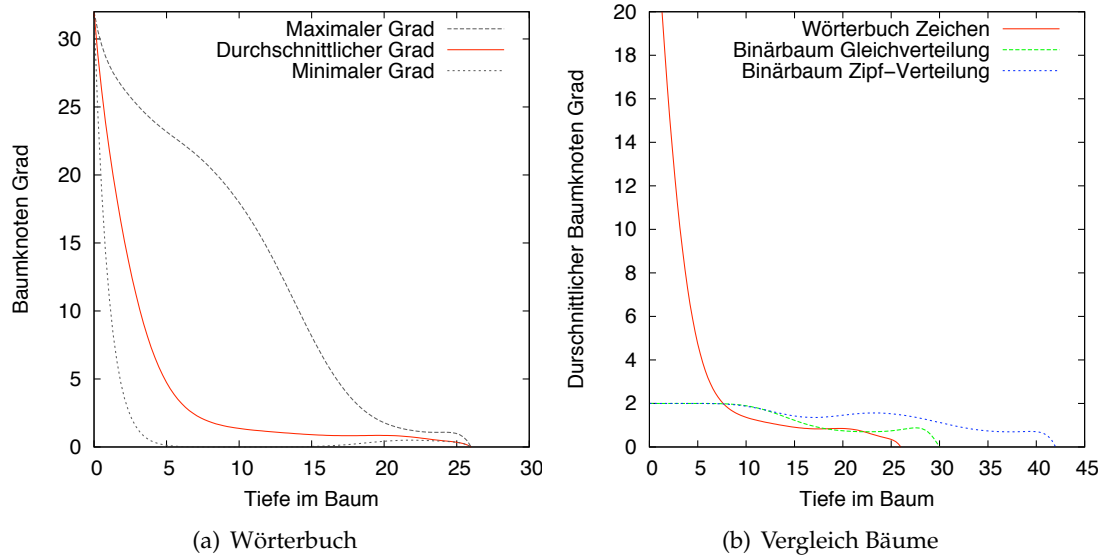


Abbildung 4.3.: Grad der Baumknoten abhängig von der Tiefe im Netzwerkbaum

tiefen bis 120 erreicht mit entsprechend geringer Expansion und hohem Peer-Grad. Für möglichst gute Resultate sollte der Datenraum hier also gut ausgenutzt werden um einen gleichmäßigen Baumgrad zu erzielen.

4.2. Entstehungsprozess des Netzwerks

Die Peers in einem 3nuts Netzwerk wählen ihre normalen Pfade zur Lastbalancierung abhängig von einer eingeschränkten Sicht auf den globalen gewichteten Datenbaum, der im Netzwerk verwaltet wird. Die Gewichte dieses globalen Baums ergeben sich aus der Summe der Einzelgewichte aller Daten und jeder Peer bringt einen Teil dieser Gewichte ins Netzwerk ein. Bei einem neu erzeugten Netzwerk kennt vom Gesamtbaum jeder Peer erst einmal nur das Gewicht seiner eigenen Daten und durch die Publikation im Netzwerk wird dieses jedem anderen Peer mitgeteilt. Die Verantwortlichen publizieren dann das Gewicht der Daten in den Baumknoten. Die Gewichte der einzelnen Baumknoten eines Teilbaums werden dann zusammenaddiert und als Gesamtgewicht des Baumzweigs mit einem BranchLink publiziert. Alle Informationen über die Baumgewichte werden mit der Pointer-Push&Pull-Operation zwischen den Peers ausgetauscht. So ergibt sich hier ein Prozess, in dem jeder Peer mehr und mehr an Informationen über den Gesamtbaum erhält. Da die Pointer-Push&Pull-Operation die Baumdaten in einem Zufallsprozess im Netzwerk verbreitet, haben nicht alle Peers zum selben Zeitpunkt die gleichen Informationen über den Baum. Hier kann jeder Peer von jedem möglichen Anteil der Gesamtdaten gehört haben. So unterscheidet sich in

4. Messungen

den Peers einerseits das Gesamtgewicht des Baums (=Anzahl Daten) und andererseits die Gewichtung in den einzelnen Teilbäumen. So kann es Peers geben, die schon mehr aus der '0' unter Wurzel eines Binärbaums gehört haben und andere aus der '1'.

Abbildung 4.4 zeigt als Beispiel den Entstehungsprozess eines 10.000er Netzwerks mit 5 Daten pro Peer also 50.000 Daten und Pfaden aus dem Wörterbuch. Im Graphen ist eine zeitliche Abfolge von gleich mehreren Verteilungen über das Gesamtbaumgewicht enthalten. Die einzelnen Momentaufnahmen haben dabei im Wechsel verschiedene Farben und Linientypen und wurden in zeitlich festen Abständen aufgenommen. Dabei wurde das Netzwerk so schnell wie möglich erzeugt mit einer initialen Verteilung beginnend bei Baumgewicht 0. Die Daten werden dann im Netzwerk publiziert und die davon abgeleiteten Baumgewichte im Netzwerk verbreitet, so dass die Verteilung zeitlich im Graphen nach rechts zu einem höheren Gesamtgewicht "wandert". Durch die Pointer-Push&Pull-Operation hat jede Verteilung eine gewisse Varianz, wobei manche Peers mit höherem Gewicht schon mehr über den Baum wissen. Andere wiederum weichen mit einem kleineren Baumgewicht nach unten ab, weil sie evtl. die Pointer-Push&Pull Operation weniger oft aufgerufen haben. Die Verbreitung der Baumgewichte erfolgt analog zur epidemischen Informationsausbreitung mit Push und Pull. Erreicht die Ausbreitungsgeschwindigkeit hierbei ihr Maximum bei den beiden Verteilungen zwischen Gewicht 8.000-20.000, so zieht sich die Verteilung entsprechend weiter auseinander, da viele neue Bauminformationen von den einzelnen Peers in Erfahrung gebracht werden und sich die Peers dabei stärker unterscheiden.

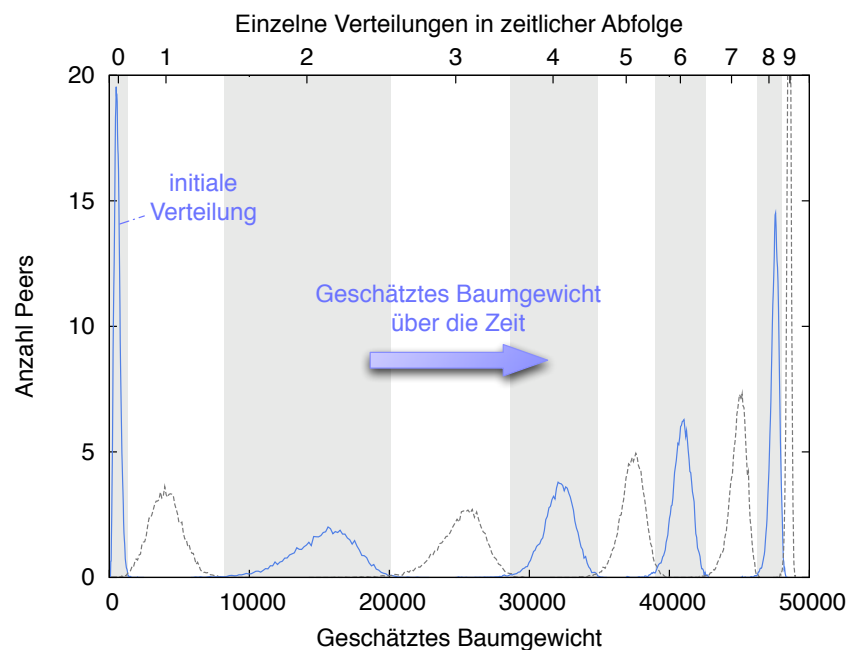


Abbildung 4.4.: Verteilung über geschätztes Baumgewicht beim Netzwerkaufbau

Abbildung 4.5 unterstreicht nochmals den Charakter der epidemischen Informati-

4.2. Entstehungsprozess des Netzwerks

onsausbreitung durch Push&Pull im Netzwerk. Hier wird über die Zeit das durchschnittliche, minimale und maximale Baumgewicht in allen Peers über die Zeit dargestellt, also eine andere Darstellung der Verteilungen aus Abbildung 4.4. In einer Start-

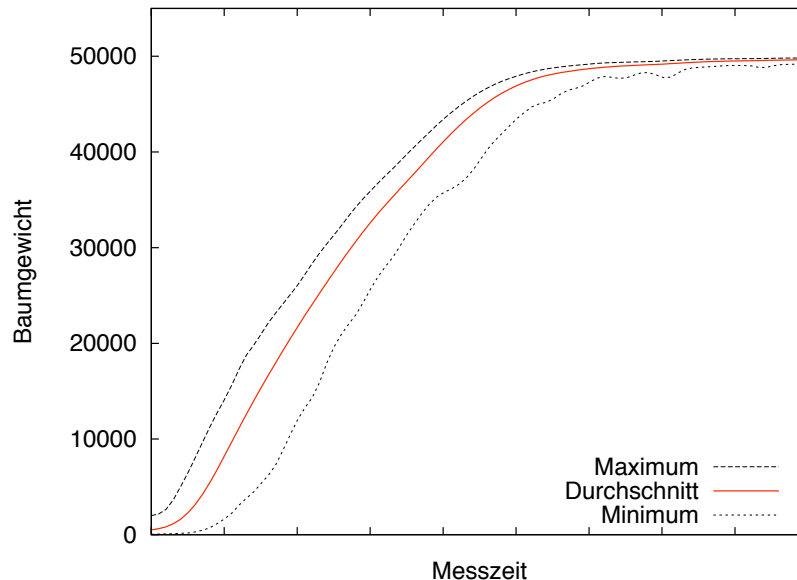


Abbildung 4.5.: Zeitlicher Verlauf des geschätzten Baumgewichts beim Netzwerkaufbau

phase kennen die Peers zunächst nur die Gewichte der selbst verwalteten Daten. Und erzeugt ein Peer einen BranchLink für einen Zweig, wo er Daten verwaltet, enthält dieser zunächst nur dieses Gewicht und noch nicht die Gewichte von anderen Peers in diesem Zweig. Durch die Verbreitung dieser BranchLinks bringen die Peers dann mehr und mehr Baumgewichte in Erfahrung, die von anderen Peers verwaltet werden. Diese geben sie dann wiederum selbst aufsummiert in den BranchLinks weiter. So kommt es zu einem starken Anstieg der Ausbreitungsgeschwindigkeit. Darauf folgt eine Sättigungsphase, wo der Großteil der Peers über den Großteil der Daten informiert ist. Dabei gibt es aber noch einen kleinen Teil an verantwortlichen Peers, die das verwaltete Gewicht nicht häufig genug verbreitet haben. Und ein anderer kleiner Teil an Peers wurde noch nicht über alle Daten informiert. Des Weiteren ergeben sich Verzögerungen in der Ausbreitung durch die Reorganisation des Netzwerks, das sich für die Lastbalancierung ständig an den gewichteten Datenbaum anpasst. Und der gewichtete Datenbaum macht bei der Netzwerkerzeugung von Gewicht 0 auf Gewicht 50.000 die stärkste Änderung durch. Zur Anpassung gehört zum einen die Ermittlung der Verantwortlichen in den einzelnen Baumknoten, wobei gegebenenfalls die Daten des Baumknotens verschoben werden müssen, falls ein neuer Verantwortlicher ermittelt wird. Des Weiteren haben die Peers aufgrund der noch unvollständigen Bauminformationen den falschen normalen Pfad gewählt und müssen diesen entsprechend des heterogenen Hashings anpassen, sobald sie mehr über den Datenbaum erfahren.

4.3. Lookup

In diesem Abschnitt wird die Leistung der Lookup-Operation für Punktanfragen analysiert. Da die allgemeinere Suche für Bereichsanfragen die gleichen Pfade im Netzwerk wie der Lookup benutzt um zu den einzelnen Punkten des Bereichs zu gelangen, stehen die Ergebnisse ebenso für die Suche. Die Geschwindigkeit einer Lookup-Operation ergibt sich durch den Pfad im Overlay-Netzwerk, über den die Lookup-Anfrage weitergeleitet wird. Dabei hat jeder Hop auf dem Pfad eine bestimmte Latenz. Hier beschränken sich die Messungen zunächst nur auf die Hop-Anzahl (=Pfadlänge) in der Lookup-Operation und das Ergebnis wird im folgenden Abschnitt 4.4 durch Latenzen auf den Hops präzisiert. Da jeder Peer eine andere Position im Overlay-Netzwerk durch die den normalen Pfade hat, ist auch der Routingpfad von den einzelnen Peers zum gleichen Verantwortlichen eines Baumzweigs verschieden. Und die Pfade können sich über die Zeit ändern, da die Links auf die Baumzweige mit Pointer-Push&Pull zufällig gehalten werden. So ergibt sich für das Kreuzprodukt (Peers $\times$ Daten) eine Verteilung von verschiedenen Hopanzahlen erzielt auf den einzelnen Lookup-Pfaden. Im Fall der getesteten Netzwerke sind dies $2^{28} \cdot 5$ Operationen aus denen eine Million zufällige Operationen getestet wurden wonach sich das prozentuale Verhältnis von Hops in den Operationen kaum noch änderte (Abbildung 4.8). Für eine einzelne Messung wurde im

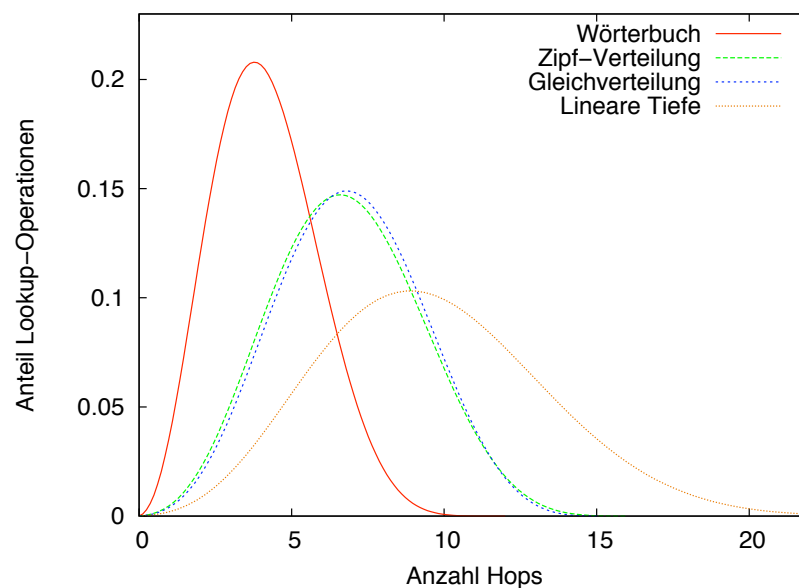


Abbildung 4.6.: Verteilung der Hopanzahl in Lookup-Operation für 2^{14} Peers

3nuts Peer eine erweiterte Lookup-Operation implementiert. Dabei wird die Anfrage-Nachricht mit der aktuellen Hop-Anzahl erweitert, die bei jedem Hop zum nächsten Peer inkrementiert wird. Die beim Verantwortlichen erzeugte Antwort enthält dann auch diese Hop-Anzahl, die dann als Ergebnis zusätzlich zurückgegeben wird. Jede

Einzelmessung besteht hier also aus einem einfachen Lookup-Funktionsaufruf im startenden Peer und der Hop-Anzahl als Messergebnis. So ist es hier auch möglich diese Messung verteilt durchführen zu können. Dazu können mehrere Testumgebungen auf mehreren Rechnern parallel im Verbund gestartet werden, die dann ein zusammenhängendes verteilt laufendes 3nuts Netzwerk erzeugen. Für diese Messung wird dann ebenfalls ein verteilter Service gestartet, wobei es einen Hauptservice gibt, der die Einzelmessungen anstößt und die Ergebnisse zentral erfasst. Die einzelnen Services können praktischerweise die Communication der 3nuts Peers mitbenutzen um sich bei der Ausführung der Messung auszutauschen. In der verteilten wie lokalen Ausführung der Messung wurden hier die gleichen Ergebnisse ohne Lookup-Fehler erzielt, was auf eine Lauffähigkeit von 3nuts in einem realen physikalischen Netzwerk schließen lässt.

Eine einzelne Hop-Verteilung (Abb. 4.6) ist wie folgt zu erklären: bei n möglichen Peers, die ein Datenelement verwalten können, gibt es die geringe Wahrscheinlichkeit $1/n$, dass ein Peer das gesuchte Datenelement selbst verwaltet, also im gesamten Suchpfad teilnimmt. Er kann die Lookup-Operation dann selbst ohne einen Hop beantworten und so beginnt jede Verteilung bei 0 Hops. Im schlechtesten Fall hingegen macht ein Lookup bei jedem Hop nur einen Fortschritt von +1 Tiefe im Baum, so dass hier die Anzahl Hops der maximalen Tiefe des Netzwerkbaums entsprechen kann. Dieser Fall ist auch recht unwahrscheinlich, so dass jede Verteilung nach rechts gegen 0 ausläuft. In den meisten Fällen ist es so, dass der ausführende Peer im Lookup immer gleich einen Teil des Suchpfads kennt und so gleich mehrere Tiefen im Pfad überspringen kann, wo es in der Verteilung ein entsprechendes Maximum gibt. Im Baum für das Wörterbuch sind wegen des hohen Baumgrads die Suchpfade kürzer und die Expansion größer und entsprechend ist auch die Zahl Lookup-Hops kleiner. Zwischen den Binärbäumen mit Zipf- bzw. Gleichverteilung gibt es keinen nennenswerten Unterschied und die Datenverteilung hat in diesem Bereich keinen Einfluss auf die Performanz.

Es ist aber denkbar für die Performanz noch ungünstigere Binärbäume zu erzeugen. Hierbei stellt ein Binärbaum mit linearer Tiefe zur Anzahl Peers den 'Worst Case' dar. In der Verteilung liegen die Daten nur in den Blättern 0^k1 mit $k = 1..n$ und n Peers. Bei einer idealen Lastverteilung befindet sich in jedem dieser Blätter ein Peer, so dass ein Peer den normalen Pfad 0^n mit Länge n also linearer Tiefe hat. Ist dann das Ziel der Lookup-Operation der Pfad 0^n , kann die Operation bis zu $(n - 1)$ Hops benötigen. Nämlich dann, wenn bei jedem Hop nur eine Baumtiefe mehr erzielt wird. Der anfragende Peer befindet sich dann im normalen Pfad 1 und routet die Anfrage zu einem zufälligen der $(n - 1)$ Peers im Zweig 0. Diese Anfrage erhält zufällig der Peer mit normalen Pfad 01, der die Anfrage nach Zweig 00 weiterleitet usw. Bei dieser Lookup-Operation können so $h = 0..(n - 1)$ Hops benötigt werden mit folgenden Wahrscheinlichkeiten:

$$P(h, n) = \begin{cases} \frac{1}{n} & \text{für } h = 0, \\ \frac{1}{n} \cdot \sum_{i=1}^{n-h} P(h-1, n-i) & \text{für } h > 0, \\ 0 & \text{sonst.} \end{cases}$$

Die Wahrscheinlichkeit für $h = 0$ Hops, dass also der anfragende Peer bereits im Pfad

4. Messungen

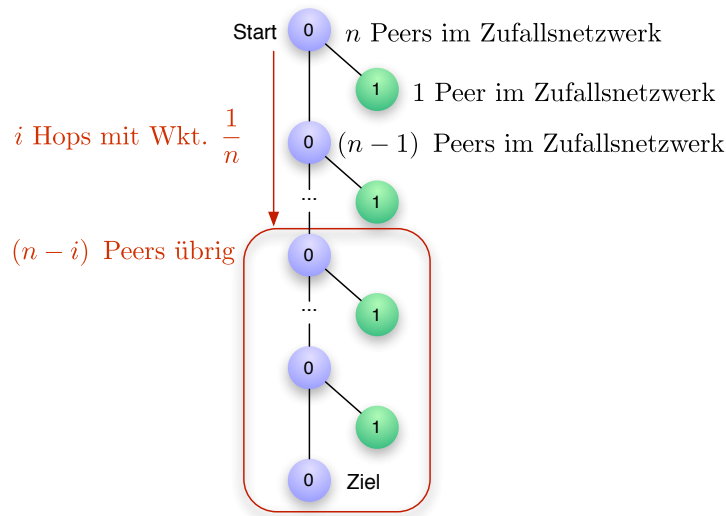


Abbildung 4.7.: Binärbaum mit linearer Tiefe

0^n teilnimmt, ist dabei $\frac{1}{n}$. Die Wahrscheinlichkeit für mindestens einen Hop also $h > 1$ ist rekursiv definiert. Mit den echt zufälligen Links in den Baumzweigen ist die Wahrscheinlichkeit bei noch n verbleibenden Peers in Tiefe $i = 1..n$ zu kommen gleichverteilt $\frac{1}{n}$. Erreicht der Hop dann Tiefe i so bleiben noch $(n - i)$ Peers im Zufallsnetzwerk 0^i übrig, mit denen dann noch rekursiv $(h - 1)$ Hops zu bewältigen sind. Die Verteilung der Hops ist ebenfalls im Graph in Abbildung 4.6 für 2^{14} Peers angegeben. Dabei erreicht der konstruierte Worst-Case mit linearer Tiefe 9, 28 Hops im Erwartungswert. Die anderen beiden Daten-Verteilungen im Binärbaum haben etwa eine durchschnittliche Hop-Zahl von 6, 84 (Wörterbuch 3, 84). Bei den 3 Verteilungen der Binärbäume benötigt der Baum mit linearer Tiefe also nur 35% mehr Hops als Gleich- bzw. Zipf-Verteilung.

Abbildung 4.8 unterstreicht die logarithmische Lookup-Laufzeit abhängig von der Netzwerkgröße, die hier exponentiell aufgetragen ist. Für die Anzahl Hops $c \cdot \log n$ beträgt der Vorfaktor $c \leq \frac{1}{2}$, der mit steigendem Baumgrad sinkt.

4.4. Netzwerklokalität

Aus der Sicht einer Applikation ist bei der Benutzung eines Peer-to-Peer Netzwerks maßgeblich die Anfragezeit entscheidend. Die Applikationen in einem Overlay-Netzwerk laufen typischerweise verteilt im Internet. 3nuts erlaubt eine Optimierung des Overlay-Netzwerkstruktur zugeschnitten auf das physikalische Netzwerk. Dabei stellt sich die Frage, welche Performanz-Verbesserungen sich beim Lookup in der Umgebung des Internets dabei einstellen. Hierzu wird ein Internet-ähnliches physikalisches Netzwerk in der Testumgebung simuliert, wobei hier die Latenz zwischen 2 Peers der entscheidende Hauptfaktor ist. Dann wird ein 3nuts Netzwerk in der Testumgebung

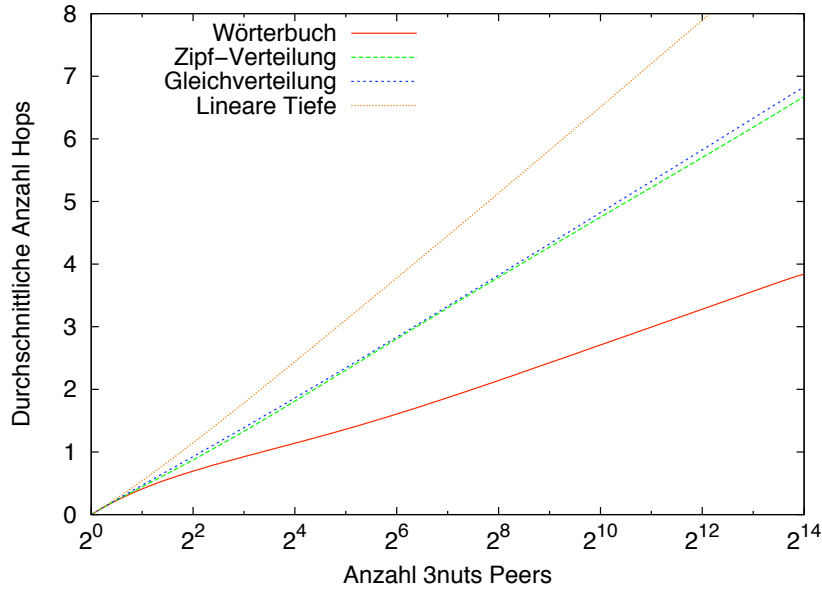


Abbildung 4.8.: Durchschnittliche Hopanzahl in Lookup-Operation

erzeugt und jeder 3nuts Peer wird einem Knoten im simulierten physikalischen Netzwerk zugewiesen. Wird dabei eine Nachricht von einem Peer zu einem anderen geschickt, dann wird die Nachricht über die *Communication* versendet und zugestellt. Im realen physikalischen Netzwerk ergibt sich dabei für jedes Sender-Empfänger Paar eine individuelle Latenz. In der Simulation wird diese dadurch erzielt, dass die Nachrichten entsprechend der Latenz künstlich verzögert werden. Dazu wird ein *TaskScheduler* benutzt, in dessen Zeitplan eine Nachricht mit dem verzögerten Zustelldatum eingetragen wird und zu gegebener Zeit dann wieder entfernt und zugestellt wird.

Für die Verzögerungen zwischen Sender-Empfänger Paaren werden 2 Modelle benutzt, die hier das Internet nachstellen sollen. Diese Modelle sind aber nicht zwingend repräsentativ fürs Internet, sondern sollen hier nur einen Eindruck über die Performanz in solch einer Umgebung vermitteln. Im ersten Modell erhalten alle Peers eine zufällige Position auf der Oberfläche einer Kugel mit zuvor definiertem Radius (siehe Abbildung 4.9(a)). Die Latenz ergibt sich dann durch den Abstand auf der Kugeloberfläche. Die kürzeste Verbindung zwischen 2 Punkten $\vec{p}_1, \vec{p}_2$ auf der Kugeloberfläche berechnet sich dabei durch den eingeschlossenen Winkel mal dem Radius:

$$\text{delay}_{\text{kugel}} := \arccos \left(\frac{\vec{p}_1 \cdot \vec{p}_2}{|\vec{p}_1| \cdot |\vec{p}_2|} \right) \cdot r$$

In einem zweiten Modell – dem Georgia Tech Internetwork Topology Model (kurz *GT-ITM*) [ZCB96] – wird zusätzlich zu einer zufälligen Struktur noch eine hierarchische Note hinzugefügt. Wie in Abbildung 4.9(b) zu sehen, besteht die Hierarchie hier aus 2 Ebenen: Lokale Netzwerke bezeichnet als *Stub-Domänen* mit einer zufälligen internen

4. Messungen

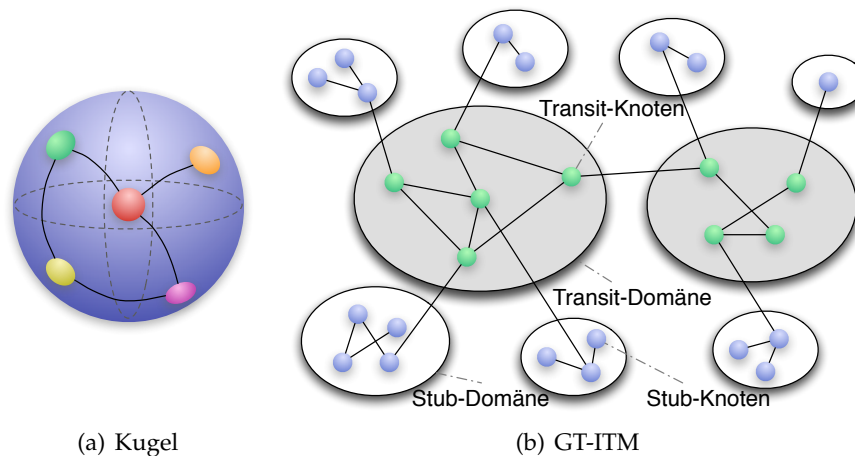


Abbildung 4.9.: Topologie Modelle für das physikalische Netzwerk

Struktur, die in einer Hierarchie darüber an *Transit-Domänen* angeschlossen sind, die ebenfalls eine interne zufällige Struktur aufweisen und als *Backbone* des Netzwerks viele lokale Netzwerke miteinander verknüpfen. Auf der oberen Hierarchieebene können mehrere Transit-Domänen miteinander verbunden sein. Für die Simulation in 3nuts wurde ein Topologiemodell verwendet, in dem es 6 Transit-Domänen gibt, die jeweils intern 15 Transit-Knoten enthalten. Jeder dieser Transit-Knoten hat 14 Stub-Domänen, die jeweils 8 Stub-Knoten haben. Dies ergibt zusammengerechnet 90 Transit-Knoten im Backbone, die ausschließlich zur Übertragung gedacht sind, und 10.080 Stub-Knoten also Endknoten, die zur Belegung durch 3nuts Peers genutzt werden können. Der mit der GT-ITM Software erzeugte Graph hat einen durchschnittlichen Grad 3,64, den Durchmesser 18 Hops, und 13,774 Hops durchschnittlich auf einem Routingpfad.

Der Graph stellt das physikalische Netzwerk dar. Die Routingzeit zwischen zwei Knoten darin ergibt sich durch die Summe der Delays auf dem Pfad vom einen zum anderen Knoten im Graphen. Für ein vollständiges Netzwerkmodell müssen den einzelnen Kanten im Graphen also noch Delays zugeordnet werden. So erhalten hier alle Kanten zwischen zwei Stub-Knoten das einheitliche Delay 5 ms, Kanten von einem Stub-Knoten zu einem Transit-Knoten 20 ms und Kanten zwischen Transit-Knoten das größte Delay mit 100 ms. Typische Werte für den Ping (oder *Round Trip Time*, RTT), der durchschnittlich doppelt so groß wie die Verzögerungszeit zwischen zwei Knoten ist, finden sich in Tabelle 4.1. Um dies in Relation zum Modell zu setzen, kann man die Werte grob in 3 Mengen einteilen: das lokale Netzwerk mit LAN oder WLAN im Modell mit den Stub-Domänen, die Übertragung vom lokalen Netzwerk ins Internet mit den Verbindungen von Stub-Knoten zu Transit-Knoten und schließlich das Internet mit den Transit-Domänen. Für den Graphen und die festgesetzten Delays der einzelnen Kanten wird dann eine Delay-Matrix errechnet, die für den Pfad jedes Knotenpaars die Verzögerungszeit für die Übertragung angibt. Die einzelnen Werte der Matrix werden mit einer kürzesten Wege-Suche bestimmt, die auf dem Algorithmus von *Dijkstra* basiert.

Tabelle 4.1.: Typische RTT-Werte [Wik09c]

Typ	Wert
100BaseT-Ethernet	1 ms
WLAN 802.11b	10 ms
DSL-6000 ohne Fastpath	40 ms
ISDN	200 ms
UMTS	350 ms
Internet innerhalb Deutschlands	<50 ms
Internet nach USA	100-150 ms
Internet nach Fernost	≤300 ms

Dabei sind die Kanten im GT-ITM Graph bereits so gewichtet, dass bei einer kürzesten Wege-Suche nur erlaubte Routingpfade gewählt werden. So führt beispielsweise der kürzeste Routingpfad von einer Transit-Domäne in eine andere Transit-Domäne niemals durch eine Stub-Domäne.

Der Radius des Kugel-Modells wurde hier so gesetzt, dass die durchschnittliche Latenz der zufälligen Links im GT-ITM Modell und Kugel-Modell gleich sind (hier 504 ms pro Hop). Der erwartete Abstand auf einer Kugeloberfläche (= Latenz) errechnet sich wie folgt: alle Punkte auf der Kugeloberfläche sind gleichwahrscheinlich. Geht man von einem festen Punkt aus, so kann ein 2. Punkt den Abstand im Bereich $d \in [0, \pi \cdot r]$ haben. In jedem Abstand d sind mehrere Punkte kreisförmig angeordnet auf der Oberfläche möglich. Integriert über die möglichen Abstände ergibt sich dann der erwartete Abstand zu:

$$E[\text{Abstand auf Kugel}] = \frac{\int_0^\pi \overbrace{r \cdot d\phi}^{\text{Abstand}} \cdot \overbrace{2\pi \cdot r \cdot \sin \phi}^{\text{Ring im Abstand}} \cdot (r \cdot d\phi)}{\underbrace{\int_0^\pi 2\pi \cdot r \cdot \sin \phi \cdot (r \cdot d\phi)}_{=4\pi r^2}} = \frac{1}{2}\pi \cdot r$$

Für eine erwartete Latenz von 504 ms ergibt sich damit ein Radius von $r = 320$.

Zur Messung wird ein 3nuts Netzwerk mit 10.000 Peers und 5 Daten je Peer also insgesamt 50.000 verschiedenen Daten erzeugt. Dabei werden alle 3 Baumverteilungen getestet. Um festzustellen, wie sich die Routinggeschwindigkeit durch die Latenzoptimierung verbessert hat, wird verglichen:

- Latenz eines Routingpfads mit Latenz-optimierten Links
- Latenz eines Routingpfads mit zufälligen Links

4. Messungen

Die Latenz eines Routingpfads ist die Summe der Latenzen auf den einzelnen Hops. Hier muss immer ein gesamter Routingpfad verglichen werden. Denn durch die Auswahl der Links ergeben sich unterschiedliche Routen zum gleichen Ziel mit eventuell unterschiedlich vielen Hops. So kann theoretisch zwar bei der Messung mit Latenz-optimierten Links die Latenz der einzelnen Hops kleiner sein, aber sich durch mehr Hops in der Summe trotzdem eine höhere Gesamtlaufzeit ergeben als mit zufälligen Links.

Abbildung 4.10 zeigt die Messergebnisse für beide Topologiemodelle im Zipf-verteilten Binärbaum. Dabei zeigen beide Modelle ein ähnliches Bild. Die Varianz der einzelnen Kurven ergibt sich durch die Varianz der Hops im Lookup (Vgl. Lookup-Hops in Abb. 4.8). Das Routing mit Latenz-optimierten Links ist beim Kugel-Modell durchschnittlich um Faktor 2,7 und beim Modell GT-ITM um den Faktor 2,4 schneller.

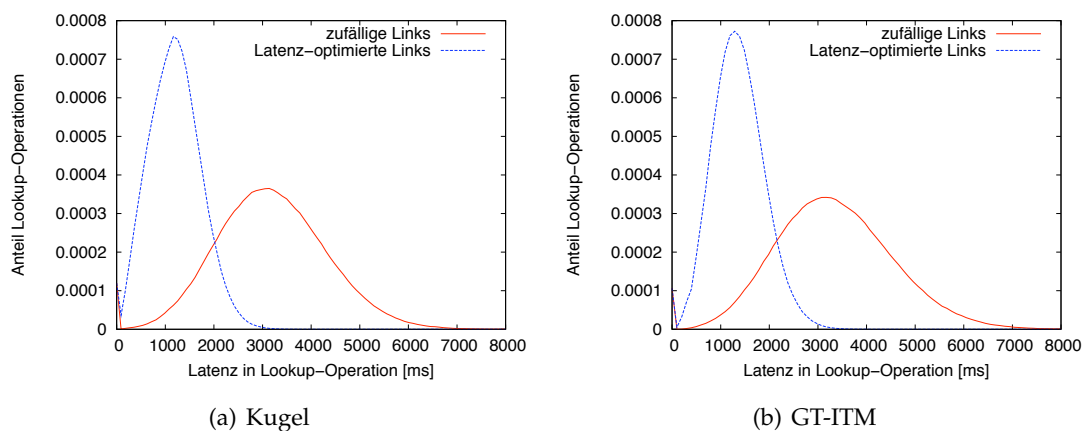


Abbildung 4.10.: Lookup-Latenz bei 10.000 Peers und Zipf-verteilten Binärbaum

Interessant ist hier auch der direkte Vergleich der erzielten Latenzen in den unterschiedlichen Datenbäumen mit Wörterbuch, Gleichverteilung und Zipf-Verteilung in Abbildung 4.11. Hier sind die durchschnittlichen Lookup-Latenzen dargestellt. Werden die zufälligen Links zum Routing benutzt, so hat jeder Lookup-Hop im Mittel die durchschnittliche Latenz zwischen 2 beliebigen Peers im Netzwerk von 504 ms. Nimmt man also die durchschnittliche Anzahl Hops in den 3 Datenbäumen (Abb 4.8) und multipliziert diese mit der durchschnittlichen Latenz zwischen 2 Peers, so kommt man in etwa auf die durchschnittliche Lookup-Latenz in diesen Messungen. Dabei dauert der Lookup in den Binärbäumen wegen dem geringeren Baumgrad und den längeren Routingpfaden um etwa Faktor 1,8 länger. Bei den Latenz-optimierten Links hingegen schrumpft dieser Vorsprung des Wörterbuchs auf nur noch Faktor 1,1. Hier scheint es so zu sein, dass die anfänglichen Hops im Wörterbuch nicht so gut optimiert sind, da sich die Peers durch den hohen Baumgrad sehr schnell in kleine Gruppen aufteilen und es so keine "große Auswahl" mehr für die geringste Latenz gibt. So gibt es bei den Binärbäumen mehr Hops mit dafür kleinerer Latenz pro Hop.

Man kann also mit der Optimierung im physikalischen Netzwerk die Lookup-Latenz

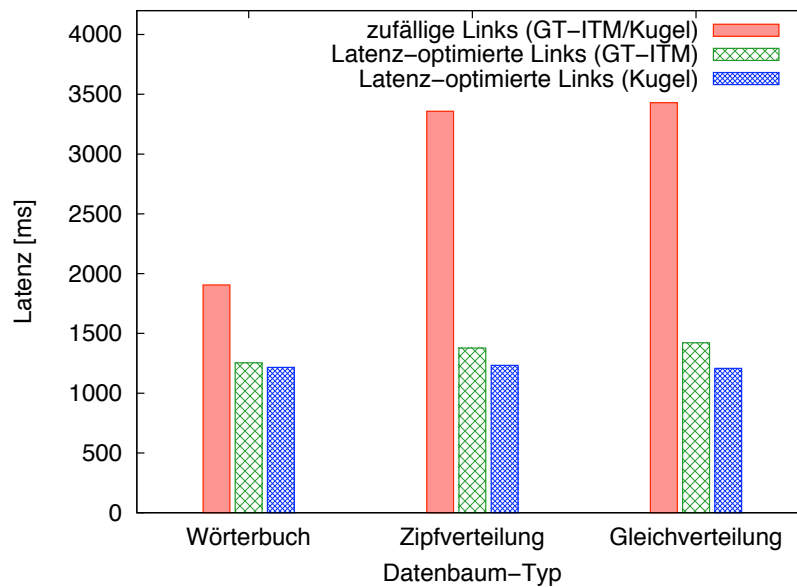


Abbildung 4.11.: Vergleich der Lookup-Latenz im Wörterbuch-Baum und den Binärbäumen

erheblich verringern. Bei den erzielten Latenzen ist hier auch zu beachten, dass die Verwaltung eines Datenpfades mit DHHT einem zufälligen Peer im Overlay-Netzwerk zugeordnet wird. Jeder Datenpfad wird hier also ohne Replikation von genau einem Peer an einer beliebigen Stelle im physikalischen Netzwerk verwaltet. So nehmen die verantwortlichen Peers hier beispielsweise nicht durch Redundanz/Replikation der Verantwortlichkeit "strategisch günstige Positionen" im Netzwerk ein, was die Latenz noch weiter verbessern könnte. Betrachtet man hier alle möglichen anfragenden Peers und möglichen zu suchenden Pfade, so kommt hier ein Routing zwischen allen möglichen Paaren von Peers in Frage.

4.5. Lastverteilung

In 3nuts wird mit Hilfe von heterogenen Hashtabellen (DHHT) aktive Lastbalancierung betrieben. Dabei geht es darum, dass die Verwaltung der einzelnen Daten möglichst gleichmäßig auf die Peers im Netzwerk aufgeteilt wird. In den getesteten Netzwerken mit 16.384 Peers hat jeder Peer selbst 5 Datenelemente und somit sollte jeder Peer im Optimum auch nur genau 5 Datenelemente aus dem Netzwerk verwalten müssen. Abbildung 4.12(a) zeigt eine Verteilung darüber, wie viele 3nuts Peers welches Datenvolumen verwalten müssen. In Abbildung 4.12(b) wird die gleiche Verteilung aus Abbildung 4.12(a) kumulativ dargestellt. Hieraus geht hervor, dass mit 66% die Mehrheit aller Peers weniger gleich der durchschnittlichen Datenlast von 5 Datenelementen haben und bei der 3-fachen Datenlast sind es schon 97%. Darüber hinaus gibt es noch eine

4. Messungen

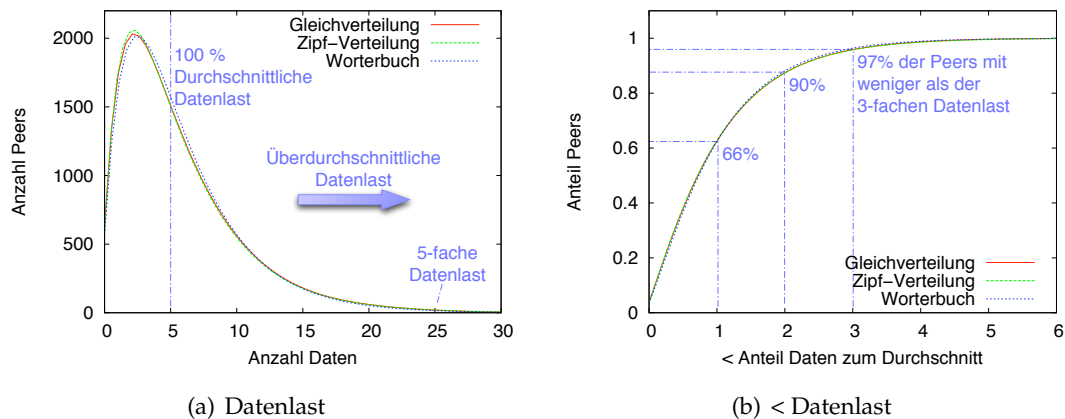


Abbildung 4.12.: Datenlast: Anzahl verwalteter Daten pro Peer

kleine Anzahl von Peers die eine etwas höhere Datenlast haben und die Funktion klingt exponentiell aus. Der höchste gemessene Wert dabei war Faktor 12. Ähnlich zu einer verteilten Hashtabelle (DHT) gibt es in 3nuts mit heterogenem Hashing (DHHT) ebenso eine Varianz in der Verteilung mit der daraus resultierenden Abweichung. Ein gutes Ergebnis ist hierbei auch, dass sich das 3nuts Netzwerk sehr gut an die 3 verschiedenen Datenbäume mit Wörterbuch-Baum und den Binärbäumen mit Gleich- bzw. Zipf-Verteilung anpasst hat. Dabei gibt es fast keine Unterschiede in der Lastverteilung.

Abbildung 4.13 zeigt nochmals anschaulich, wie die Peers auf die inhomogene Datenverteilung in den einzelnen Pfaden abgebildet werden. Dabei werden alle Pfade in einer bestimmten Tiefe miteinander verglichen und nach dem Datengewicht in den Subbäumen dieser Pfade absteigend sortiert. So gibt es wie in Abbildung 4.13(c) zu sehen im Binärbaum in Tiefe 5 genau $2^5 = 32$ Pfade, auf die das Gesamtgewicht des Baums aufgeteilt ist. Wegen eines Speicherengpasses wurde bei der Messung der Zipf-Verteilung die Anzahl Peers von 2^{14} auf 2^{13} reduziert. Diese Anschauung lässt erkennen, dass es in den Datenverteilungen ein großes Ungleichgewicht gibt. So gibt es in der Zipf-Verteilung mit 8.192 Peers und 40.960 Datenelementen einen Präfix mit über 2.800 Daten, dem gegenüber andere Präfixe in der gleichen Tiefe nur 0-1 Datenelemente haben. Die Pfade mit viel mehr Daten verteilen sich allerdings auf die Präfixe weiter oben im Baum, so dass die Ungleichgewichtung nach oben im Baum abnimmt, wenn man Abbildung 4.13(c) mit 4.13(d) vergleicht. Sind die Peers also in jeder Tiefe nur um einen kleinen Faktor ungleichmäßig aufgeteilt, so exponentiert sich dieses Ungleichgewicht über mehrere Baumtiefen hinweg und man kann tief unten im Baum ein sehr großes Ungleichgewicht erzeugen.

Neben der Verteilung der Daten auf die Peers ist auch die Verteilung der Netzwerklast interessant. Denn jeder Peer erzeugt hier nicht nur Netzwerkverkehr, wenn er eine Anfrage stellt oder als Verantwortlicher von Daten eine Antwort schickt, sondern die Peers müssen im Multi-Hop-Routing des Lookups noch zusätzlich Nachrichten weiterleiten, wo sie selbst weder anfragen noch antworten. Dies wurde in einem 3nuts

4.5. Lastverteilung

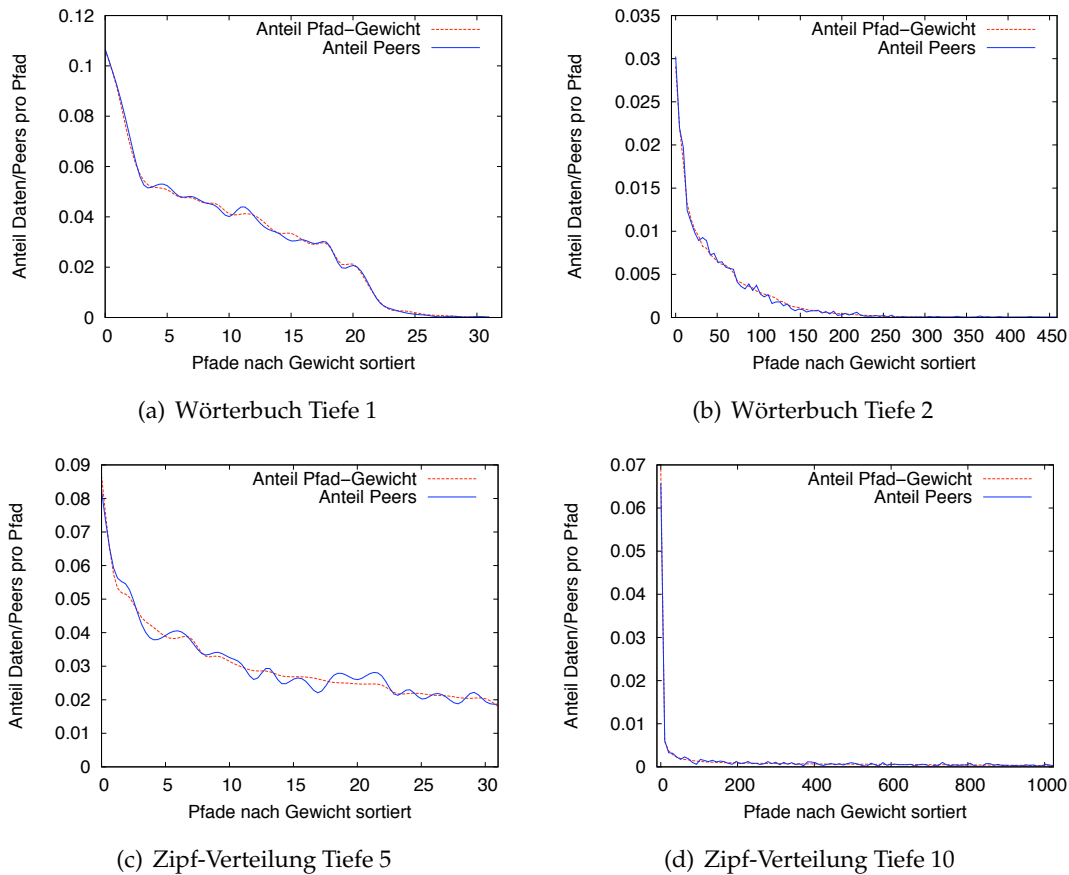


Abbildung 4.13.: Aufteilung von Datengewichten und Peers auf Baumpfade

Netzwerk in Form der Peer Congestion ermittelt. Hierzu wird stichprobenartig eine zufällige Menge von 1 Million Lookups aus dem Kreuzprodukt (Peers $\times$ Datenelemente) durchgeführt. Dabei wird gemessen, wie viele Anfragen jeder Peer bei diesen 1 Million Lookup-Pfaden durchs Netzwerk versenden muss.

Da hier alle 3nuts Peers die gleiche Communication zum Versenden von Nachrichten benutzen, kann an dieser Stelle die Nachrichtenzahl pro Peer ermittelt werden. Intern besteht eine Communication aus einer eingehenden und ausgehenden Pipeline, die die eingehenden bzw. ausgehenden Nachrichten durchlaufen und dort verarbeitet werden. Am Ende werden dann eingehende Nachrichten beim Empfänger zugestellt und ausgehende Nachrichten übers Netzwerk versendet. In die ausgehende Pipeline wird für die Messung ein zusätzliches Element eingesetzt. Darin wird jede Nachricht zum Versenden durch die Pipeline weitergereicht und zusätzlich dabei die Anzahl von versendeten Lookup-Anfragen gezählt. Die Anfragen der Messung benutzen einen speziellen Typ um diese von den Lookup-Operationen der Datenpublikation unterscheiden zu können. Die Anzahl Anfragen wird dann in einer Liste aufgeschlüsselt nach den Sender

4. Messungen

3nuts Peers gespeichert.

Im Schnitt benötigt jeder Lookup etwa 7 Hops, so dass bei 1 Mio. getesteter Lookups 7 Mio. Lookup-Anfragen im Netzwerk versendet werden. Davon versendet jeder der 16.384 Peers im Schnitt 427 Nachrichten. Abbildung 4.14(a) zeigt das Messergebnis, wobei die Anzahl versendeter Nachrichten (Peer Congestion) auf die Anzahl Peers abgebildet ist. Hierbei ist auf der x -Achse die Nachrichtenzahl auf den Schnitt normiert

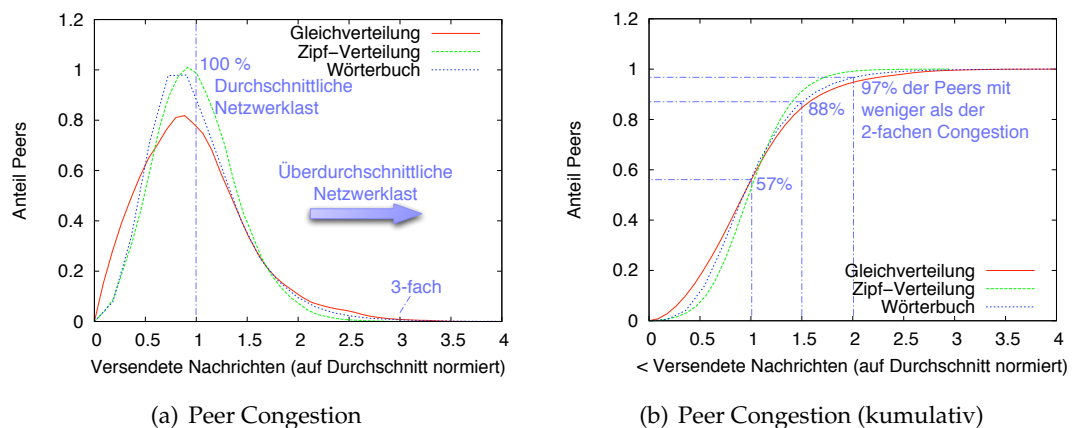


Abbildung 4.14.: Congestion: Anzahl weitergeleiteter Nachricht in Lookup pro Peer

und so haben die Peers für $x = 1$ etwa 427 Nachrichten versendet. Die Anzahl Peers auf der y -Achse ist hier ebenfalls normiert und durch die Anzahl Peers 16.384 dividiert. Abbildung 4.14(b) zeigt zusätzlich die kumulative Verteilung. Die Verteilung zeigt eine ähnliche Varianz wie auch schon die Datenlast-Verteilung. Allerdings fällt hier der erreichte Maximalwert mit einer 5-fachen Netzwerklast bei der Gleichverteilung deutlich kleiner aus ($= 5 \cdot 427$ Nachrichten). Des Weiteren zeigen die unterschiedlichen Daten-Verteilungen verschiedene Ergebnisse. Die inhomogeneren Daten-Verteilungen mit tieferen Bäumen beim Wörterbuch und der Zipf-Verteilung führen gegenüber dem gleichverteilten Binärbaum zu einer besseren Netzwerklast-Verteilung. Da hier in manchen Pfaden mit einer höheren Datenlast entsprechend mehr Peers sind, kann die Last beim Routing offenbar besonders in den anfänglichen Tiefen des Baums besser auf diese Peer-Mengen verteilt werden.

Des Weiteren ist bei den Messergebnissen zu beachten, dass es in 3nuts durchaus möglich ist die Routingpfade zufällig zu halten. So können BranchLinks auf beliebige Peers in den betreffenden Baumzweigen zeigen. Über die Zeit wird der zufällige BranchLink mit Pointer-Push&Pull zufällig gehalten. Genauso könnte man auch bei den Latenz-optimierten einen zufälligen Link aus den Links mit niedrigster Latenz wählen (momentan nicht unterstützt). Auf diese Weise ändern sich dann die Routingpfade ständig. So ist es hier dann nicht mehr so wie bei der Datenlast-Verteilung, dass ein Peer immer die gleiche fest zugewiesene Last tragen muss. Durch die zufällige Änderung der Linkstruktur ändert sich dann die Netzwerklast für einen Peer über die Zeit und kann sich so ausgleichen.

4.6. Robustheit des Netzwerks

Zur Analyse der Robustheit des Netzwerks werden Netzwerke mit 10.000 Peers und jeweils 5 Daten der unterschiedlichen Bäume erzeugt und darin 25% der Peers also 2.500 Peers gelöscht. Dann werden im Wesentlichen 2 Eigenschaften untersucht: Anzahl von Fehlern in den Routingtabellen und daraus resultierend wie viele Daten mit den fehlerhaften Routingtabellen noch erreichbar sind.

Die Analyse beschränkt sich hier auf die BranchLinks, die in der Lookup Operation für das Routing benötigt werden. In jedem Baumzweig nehmen eine Menge Peers teil und ein BranchLink für einen bestimmten Baumzweig enthält einen Link auf einen zufälligen Peer im Baumzweig und einen weiteren Latenz-optimierten Link auf den Peer mit der geringsten Latenz im Baumzweig. Diese beiden Links werden durch Listen mit alten Links erweitert um so die Robustheit zu erhöhen. Diese Listen werden bei der Messung nicht betrachtet.

Zur Erinnerung, die Reparatur von ungültigen Links in den BranchLinks erfolgt durch 2 Mechanismen: einerseits werden in der Pointer-Push&Pull-Operation ständig Links für alle Baumzweige ausgetauscht. Peers, die in einem Baumzweig selbst teilnehmen, publizieren dabei ihren eigenen Link, was Peers, die das Netzwerk verlassen haben, nicht mehr tun. Auf diese Weise werden ungültige Links durch neu publizierte gültige Links "überschrieben". Andererseits werden aber auch Links explizit in der Lookup-Operation überprüft. Wird dabei über einen Link eine Anfrage geschickt, so muss der Empfänger eine Bestätigung schicken. Die Lookup-Operation wird einerseits zur Suche andererseits aber auch in der Publikation der Daten verwendet. In den Applikationen der Peers werden die Daten kontinuierlich neu publiziert, so dass die Routingpfade von Datenbesitzer zu Datenverwalter ständig direkt überprüft werden.

In der Messung bezüglich ungültiger Links werden zuerst 2.500 von 10.000 Peers gelöscht und danach in einem Intervall über die Zeit die Anzahl ungültiger Links in jedem einzelnen Peer berechnet. Alle Peers benutzen zum Nachrichtenaustausch in der Testumgebung dieselbe Communication. Diese Communication enthält eine Adressverwaltung, in der jeder "aktive Peer" mit einer lokalen internen Adresse registriert ist um eingehende Nachrichten zugestellt zu bekommen. Zur Bestimmung der ungültigen Links in einem 3nuts Peer wird dann über die Link-Menge des Peers traversiert und für jeden Link geprüft, ob die Linkadresse noch in der Adressverwaltung der Communication registriert ist und damit der Link auf einen aktiven Peer zeigt und gültig ist. Somit ist die Messung mit den lokalen Operationen recht effizient und die Peers müssen zur Überprüfung nicht etwa angepingt werden oder ähnliches.

Abbildung 4.15(a) zeigt exemplarisch für einen gleichverteilten Binärbaum die Anzahl ungültiger Links aufgeschlüsselt nach zufälligen und lokalen BranchLinks, die über die Zeit repariert werden. Der zeitliche Verlauf der x -Achse ist in durchschnittlichen Pointer-Push&Pull-Operationen pro 3nuts Peer angegeben. Die Anzahl Operationen kann über die dabei entstehende Nachrichtenzahl in der Communication gemessen werden. Das exponentielle Abklingen der zufälligen Links ist dabei damit zu erklären, dass es zu Anfang sehr viele ungültige Links gibt. Wird dann bei einer Pointer-

4. Messungen

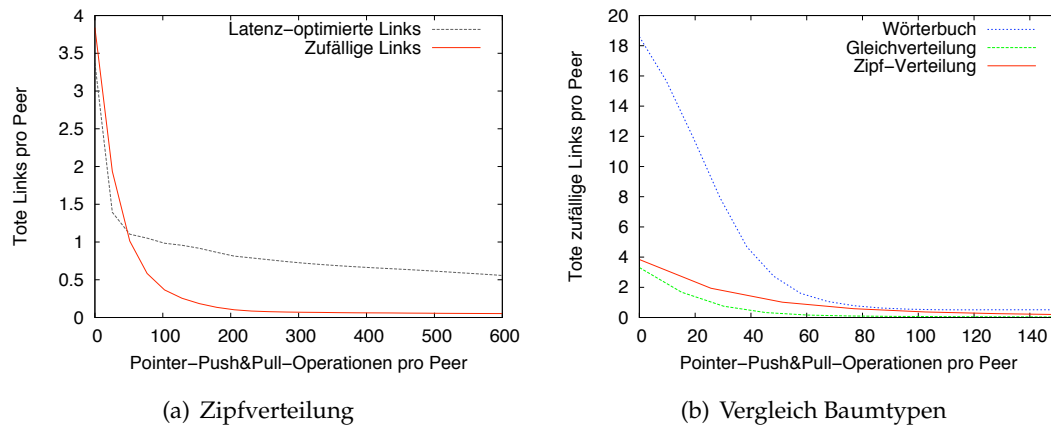


Abbildung 4.15.: Reparatur in einem 10.000er Netzwerk und 25% Ausfall

Push&Pull-Operation zwischen 2 zufälligen Peers ein gültiger Link übertragen, dann ist die Wahrscheinlichkeit recht hoch damit einen ungültigen Link zu überschreiben. Und desto weniger ungültige Links da sind, desto geringer wird diese Wahrscheinlichkeit. Um hierbei sicherzustellen, dass nicht ungültige Links ständig gültige Links überschreiben, sind die Links mit einem Zeitstempel versehen, wobei immer der Link mit dem geringeren Alter gewählt wird. Da ungültige Links nicht erneuert werden, werden diese mit hohem Alter dann irgendwann ausgemustert.

Der Verlauf bei den Latenz-optimierten Links hingegen fällt etwas anders aus. Die zufälligen Links werden ständig durch neue zufällige Links ersetzt und so die Routingtabelle repariert. Ein Latenz-optimierter Link hingegen wird nur ersetzt, wenn es einen besseren Link mit geringerer Latenz gibt, so dass sich dieser Link nicht mehr ändert, wenn das Optimum gefunden wurde. Und dieser optimale Link wird dann immer fürs Routing in den Baumzweig benutzt. Auf diese Weise ergeben sich "feste Routingpfade" von Datenbesitzer zu Datenverwalter, die immer die gleichen Latenz-optimierten Links auf dem Routingpfad verwenden. Da die Daten durch die Applikation kontinuierlich neu publiziert werden, werden diese festen Routingpfade auch ständig durch die Lookup-Operation direkt überprüft. Auf diese Weise werden so zu Anfang recht schnell alle fehlerhaften Einträge auf diesen Pfaden erkannt und korrigiert. Dann gibt es allerdings auch Latenz-optimierte Links, die sich nicht auf diesen Pfaden befinden und daher im Lookup der Datenpublikation nie benutzt werden. Daher werden hier die fehlerhaften Einträge nicht erkannt und die Anzahl fehlerhafter Latenz-optimierter Links stagniert abrupt bei etwa 40 Operationen pro Peer (im Beispiel ist die Anzahl von Pointer-Push&Pull bzw. Lookup Nachrichten in etwa gleich groß). Diese fehlerhaften Links sind aber auch mit einem Zeitstempel versehen und werden mit zu hohem Alter entfernt. Auf diese Weise sind so oft benutzte Routingwege sehr gut gewartet. Selten benutzte Latenz-optimierte Links allerdings können eher fehlerhaft sein. Hier kann der Peer fürs Routing im Fehlerfall dann aber auf den zufälligen Link bzw. alte noch gespeicherte Links für den Zweig zurückgreifen, so dass das Routing trotzdem gewährleistet

ist.

Abbildung 4.15(b) zeigt einen Vergleich für die Reparatur der Routingtabellen bei den unterschiedlichen Baumtypen. Hier fällt direkt auf, dass die Anzahl an ungültigen Links beim Wörterbuch sehr viel größer ist. Das liegt daran, dass zwar der Grad der Peers hier insgesamt zwar nicht viel größer ist, aber der Anteil der BranchLinks ist durch den höheren Baumgrad größer (Vgl. Abb. 4.2(d)). Da die BranchLinks hier gemessen werden und anteilig ausfallen, fällt so die absolute Zahl an ungültigen Links hier größer aus.

Beim Ausfall von Peers stellt sich auch die Frage welcher Anteil Daten mit dem Multi-Hop Routing im Lookup noch erreichbar ist. Eine solche Messung wurde hier ebenfalls angestellt, wobei zu Anfang von 10.000 Peers 2.500 gelöscht werden. Dann wird in einem Intervall der zeitliche Verlauf der Datenverfügbarkeit gemessen. Eine einzelne Bestimmung der Datenverfügbarkeit erfolgt über 100.000 Stichproben der möglichen Lookup-Operation aus $(\text{Peers} \times \text{Daten})$. Erreicht dabei eine Anfrage in der Lookup-Operation startend bei einem bestimmten Peer ihr Ziel, so zählen die Daten dort als verfügbar. Der Anteil aller erfolgreichen Lookups definiert damit die Datenverfügbarkeit im Intervall $[0, 1]$. Da durch den Ausfall der Peers und deren Daten die Last im Netzwerk neu balanciert werden muss, kommen so ständig kleine Änderungen im Netzwerk vor, so dass die Messung bei 99% verfügbarer Daten anhält. Abbildung 4.16 zeigt das Ergebnis.

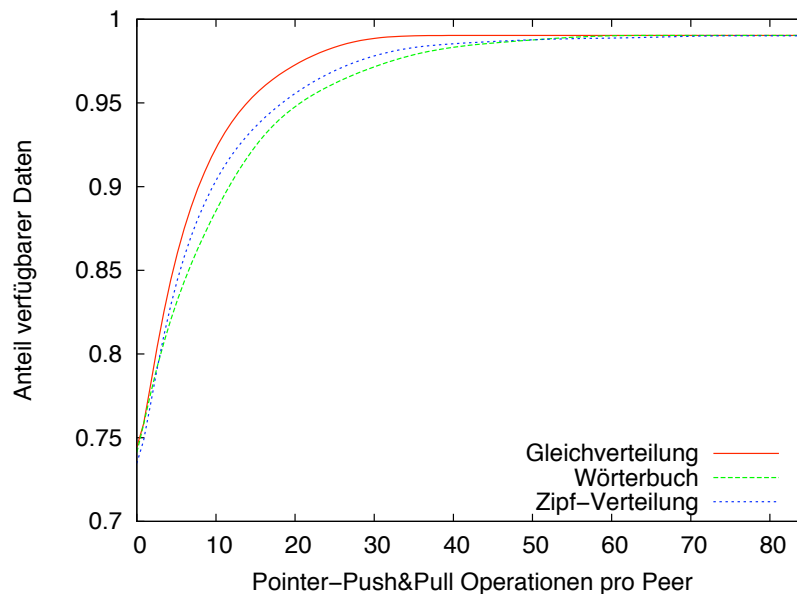


Abbildung 4.16.: Datenverfügbarkeit nach Ausfall von 25% der 10.000 Peers im Netzwerk

Da die Lookup-Operation die benutzen Links auf dem Routing-Pfad überprüft, indem bei jedem Hop eine Bestätigung erwartet wird, können hier ungültige Links kompensiert werden. Dazu wird im Fehlerfall die Reparatur des ungültigen Links angesto-

4. Messungen

ßen und der Lookup wird über einen anderen Link fortgeführt. So kann zum Messen die echte Lookup-Operation nicht benutzt werden, da dadurch der Reparaturmechanismus angestoßen würde und das Ergebnis verfälscht wäre. Deshalb wird hier auf eine Simulation des Lookups zurückgegriffen, die den exakt gleichen Routingpfad über die internen Baumstrukturen der Peers nachverfolgt, dabei aber keine Nachrichten zwischen den Peers verschickt und im Fehlerfall keine ungültigen Links löscht um sie dann zu reparieren. 100.000 Lookup-Überprüfungen pro Messschritt nehmen ihre Zeit in Anspruch. Damit sich während der Messzeit das Netzwerk nicht ändert und jeder Messschritt in sich konsistent ist, wird das Netzwerk bei jedem Messschritt in einer Art Pausefunktion angehalten und nach der Messung fortgesetzt. 3nuts Peers und deren Applikationen haben zwar komplexe interne Prozesse, aber all diese Prozesse oder Aufgaben werden einheitlich über TaskScheduler verwaltet und angestoßen. So ist es möglich im TaskScheduler allgemein eine Pausefunktion einzubauen. Dabei wird der Zeitplan eingefroren und die Threads werden gelöscht, so dass keine weiteren Tasks mehr angestoßen werden. Wird die Ausführung dann fortgesetzt, so wird der Zeitplan einfach um die Pausezeit nach hinten verschoben und die Threads neugestartet, so dass die Ausführung zeitlich versetzt fortgeführt wird. Hierzu sind alle TaskScheduler virtuell in einem großen TaskScheduler vereint, so dass hier das gesamte 3nuts System pausiert werden kann (in Testumgebung bezeichnet als "All TaskScheduler in one Array"). Zusätzlich müssen hier nochmals alle Zeitstempel in den Daten-Links zeitlich versetzt werden, die auf Applikationsebene verwaltet werden. Diese dienen als Timeout in der rundenbasierten Datenpublikation und nicht neu publizierte Daten werden nach dem Timeout gelöscht. Mit dieser Pausefunktion wird dann in jedem Zeitschritt der Messung das Netzwerk angehalten, die Messung durchgeführt und das Netzwerk wieder fortgesetzt.

Beim Messergebnis ist der zeitliche Verlauf in durchschnittlichen Pointer-Push&Pull-Operation pro 3nuts Peer angegeben. Jeder 3nuts Peer hat im verbleibenden Netzwerk etwa 10-25 Zufallnetzwerk-Peers abhängig vom Baumtyp. Rechnet man die Zeitskala hier auf die einzelnen Zufallsnetzwerk-Peers herunter, so müssen hier faktisch in jedem Zufallsnetzwerk nicht mehr als 5 Pointer-Push&Pull-Operationen aufgerufen werden, so dass die Daten fast vollständig wieder verfügbar sind.

Zusammenfassung und Ausblick

In dieser Masterarbeit wurde die Implementierung und Evaluation des semantischen Peer-to-Peer Netzwerks 3nuts vorgestellt. Das Netzwerk ist in der Lage Daten oder Ressourcen geordnet in einem Suchbaum zu verwalten. Dies ermöglicht neben der Lookup-Funktion für Punktanfragen auch Bereichsanfragen in einer Such-Funktion um Daten oder Ressourcen im Netzwerk zu finden. Hier wurde bewiesen, dass die Funktionen mit hoher Wahrscheinlichkeit nur $\mathcal{O}(\log n)$ Hops (Pfadlänge im Overlay-Netzwerk) benötigen. Durch Berücksichtigung der *Netzwerklokalität* wird in 3nuts die Overlay-Netzwerkstruktur auf die Latenzen zwischen den Peers im physikalischen Netzwerk (z.B. Internet) optimiert, um so die Routingzeiten im Lookup bzw. der Suche zu minimieren. Des Weiteren besteht hier in einer *Interessenlokalität* die Möglichkeit, dass Peers zusätzlich in weiteren Pfaden des Suchbaums teilnehmen um hier eigene Daten selbst verwalten zu können oder schneller in die Pfade routen zu können.

Um die Verwaltung eines geordneten Suchbaums im 3nuts Netzwerk zu realisieren, wählt jeder Peer einen Pfad im Datenbaum, den er dann verwaltet. Um hierbei eine gute Lastbalancierung zu erzielen, werden den Peers die Pfade mit heterogenem Hashing (DHHT) zugewiesen, so dass Teilbäume mit mehr Daten oder Last entsprechend mehr Peers zur Verwaltung zugewiesen bekommen. Damit sich die Peers über die Struktur des dynamischen Datenbaums gegenseitig informieren können, formen die Peers im selben Baumknoten ein Zufallsnetzwerk. Über die Zufallsnetzwerke werden dann Informationen über die Gewichte der Baumknoten und Links in die Baumzweige ausgetauscht. Dazu wird die *Pointer-Push&Pull*-Operation benutzt, die die Struktur der Zufallsnetzwerke zufällig hält. Hierbei können gleichzeitig die Bauminformationen an die Nachrichten der Operation angehängt werden um sie so im Zufallsnetzwerk zu verbreiten. Für die Netzwerklokalität von 3nuts werden mit der Pointer-Push&Pull-Operation auch nebenbei die Latenzen für die Links in die Baumzweige ermittelt.

Die Implementierung des 3nuts Netzwerks erfolgte in der Programmiersprache *Java*. Hierbei sind die Zufallsnetzwerk-Peers unabhängig implementiert. Ein 3nuts Peer nutzt dann mehrere Zufallsnetzwerk-Peers um in den einzelnen Baumknoten teilzunehmen. Alle diese Peers nutzen gemeinsam eine *Communication*, die hinter einem Netzwerk-

5. Zusammenfassung und Ausblick

Port den einzelnen Peers eine zusätzliche lokale Adresse zuweist und so alle Peers unabhängig voneinander Nachrichten im Netzwerk versenden und empfangen können. Alle Prozesse der Peers wurden vollständig parallel und asynchron definiert und mit Hilfe von TaskSchedulern zeitlich koordiniert. So war es hier auch möglich große 3nuts Netzwerke für die Messungen mit bis zu 20.000 Peers mit einer festen Anzahl Threads auf einem Rechner zu erzeugen. Bei der Implementierung der 3nuts Peers erhielt die Problemstellung mit dynamischen Netzwerken und Daten eine besondere Beachtung. So wird hier versucht auch bei noch sehr wagen Informationen über die gesamten Daten im Netzwerk trotzdem einen stabilen gewichteten Netzwerkbaum zu erzeugen. Denn darauf muss dann eine möglichst korrekte Lastbalancierung mit DHHT durchgeführt werden, so dass man dann zu einem stabilen Netzwerkzustand gelangen kann. Die Peers eines 3nuts Netzwerks passen bei einem dynamischen Datenbaum ihre Pfade ständig an, um die Last wieder gleichmäßig zu balancieren. Dabei werden bestehende Netzwerkstrukturen zerstört um sie anzupassen. Dementsprechend wurden die Lookup- und Such-Funktion mit entsprechenden Fehlerkorrekturen ausgestattet um auch bei stark fehlerhaften Routingtabellen ein Routing zu gewährleisten.

In den Messungen wurden Netzwerke mit $2^{14}=16.384$ Peers je 5 Daten untersucht, um bei möglichst großen und realistischen Netzwerken die Praxistauglichkeit von 3nuts zu überprüfen. Dabei wurden auch die Effekte verschiedener Datenbäume auf das Netzwerk getestet. Dazu gehörten ein gleichverteilter bzw. Zipf-verteilter Binärbaum und ein Baum mit höherem Baumgrad, wobei jeder Baumknoten ein Zeichen eines Worts aus einem Wörterbuch war. Bei der Untersuchung des Grads der Peers (=Anzahl Links auf andere Peers) stellte sich heraus, dass eine ungleichmäßigere Datenverteilung im Baum zu einem höheren Grad bei einigen Peers führt, da sich die Peers entsprechend unbalanciert auf den Baum für eine gute Lastverteilung aufteilen. Des Weiteren hängt hier der Peer Grad auch vom Baumgrad ab und wie gut der Datenraum mit möglichst kurzen Pfaden ausgenutzt wird. Messungen der Anzahl Hops in der Lookup-Funktion bestätigten für n Peers eine logarithmische Laufzeit $\mathcal{O}(\log n)$. Die Simulation eines Internet-ähnlichen physikalischen Netzwerks ergab, dass die Latenz in der Lookup-Funktion mit der Netzwerklokalität von 3nuts um bis zu Faktor 2,7 verbessert werden kann. Zusätzlich wurde hier festgestellt, dass sich durch die Netzwerklokalität die Routingzeiten bei verschiedenen Baumgraden weiter annähern, auch wenn beim Wörterbuch-Baum wegen der hohen Expansion weniger Hops im Overlay-Netzwerk benötigt werden. Bei der Lastverteilung wurde ermittelt, dass sowohl die Daten als auch die Netzwerklast bei Anfragen gut auf die Peers verteilt ist. Zu letzt wurde noch gezeigt, dass ein 3nuts Netzwerk auch mit dem Ausfall von vielen Peers (hier 25%) gut umgehen kann und die Netzwerkstruktur schnell korrigiert wird, damit alle Daten im Netzwerk wieder verfügbar sind.

Wie die Messungen gezeigt haben, hat der Baumgrad und die Ausnutzung des Datenraums einen starken Einfluss auf den Grad der Peers (= Anzahl Links auf andere Peers). Der Grad eines Peers ist ein wichtiger Faktor bei der Skalierbarkeit eines P2P-Netzwerks. Hierbei sollte bei großen Netzwerken die Last zur Verwaltung der Netzwerkstruktur bei den einzelnen nicht übermäßig ansteigen. Für eine weiterführende

Reduzierung des Grads gibt es 2 Ansatzpunkte: einerseits die Kodierung der Datenpfade im Datenbaum als auch die Abbildung des Datenbaums auf die 3nuts Netzwerkstruktur.

Bei den Datenpfaden ist nur die Identifizierung und Ordnung im Baum entscheidend. Darüber hinaus können die Pfade in beliebigen Kodierungen abgebildet werden (z.B. Zeichenstring oder Binärstring). Hier ist aber auch z.B. die *Huffman-Kodierung* [Huf52] denkbar, die einzelne Zeichen der Datenschlüssel mit unterschiedlich vielen Bits darstellt um so möglichst kurze Pfade und einen kompakten Datenbaum zu erzielen. Hierzu muss allerdings schon vorher der eingeschränkte Datenraum genau bekannt sein.

Weitaus variabler wäre hier, die Repräsentation des Datenbaums im 3nuts Netzwerk selbst dynamisch zu optimieren. So muss hier nicht zwangsweise jeder Knoten im Datenbaum als ein Knoten im Netzwerkbaum abgebildet werden. Genauso könnte der Datenbaum auch als Patricia-Trie abgebildet werden. Darin werden Pfadstücke im Baum ohne weitere Abzweige einfach als ein Knoten mit der Zeichenkette des Pfadstücks dargestellt und so der Baum komprimiert. In solchen Pfadstücken ohne weitere Abzweige befinden sich dann nämlich in den Zufallsnetzwerken jedes Baumknotens die gleichen Peers, so dass man diese dann auf nur ein Zufallsnetzwerk reduzieren kann und so den Grad der Peers und die Kosten für die Instandhaltung der Zufallsnetzwerke spart.

Da sich bei den Messungen gezeigt hat, dass der optimale Baumgrad für einen minimalen Grad der Peers nicht bei binär 2 liegt sondern etwas größer ist, könnte man hier auch noch einen Schritt weitergehen und den Baumgrad im Netzwerkbaum dynamisch anpassen. So besteht hier immer die Möglichkeit z.B. $2^3 = 8$ Pfade als Binärbaum der Tiefe 4 darzustellen oder aber auch direkt alle 8 Pfade als Kindknoten mit 3 Bit IDs unter einen Wurzelknoten mit Tiefe 2 und Baumgrad 8 zu setzen. Nach diesem einfachen Prinzip könnte man mehrere Baumtiefen dynamisch zusammenziehen und so den Baumgrad in einem möglichst optimalen Bereich halten.

Literaturverzeichnis

- [ACMD⁺03] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Puceva, and Roman Schmidt. P-Grid: A self-organizing structured p2p system. In *SIGMOD Record*, page 5, 2003.
- [ADH05] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. Multifaceted simultaneous load balancing in dht-based p2p systems: A new game with old balls and bins. In *Self-* Properties in Complex Information Systems, "Hot Topics" Series, Lecture Notes in Computer Science, LNCS 3460, Springer Verlag*, page 19, 2005.
- [BAS04] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. In *In SIGCOMM*, pages 353–366, 2004.
- [Gnu01] Clip2: The gnutella protocol specification v0.4 (document revision 1.2). http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf, 2001.
- [HHK⁺02] Kirsten Hildrum, Kirsten Hildrum, John D. Kubiatowicz, John D. Kubiatowicz, Satish Rao, Satish Rao, Ben Y. Zhao, and Ben Y. Zhao. Distributed data location in a dynamic network. In *In Proc. of ACM SPAA*, pages 41–52, 2002.
- [Huf52] D.A. Huffman. A method for the construction of minimum-redundancy codes. In *In: Proceedings of the I.R.E.*, pages S. 1098–1101, Sept 1952.
- [JOV05] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *In VLDB*, pages 661–672, 2005.
- [KK03] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table, 2003.
- [KS01] Ley Richard Karp and Scott Shenker. A scalable content-addressable network. In *In Proc. ACM SIGCOMM 2001*, pages 161–172, 2001.

Literaturverzeichnis

- [KSSV00] Richard Karp, Scott Shenker, Christian Schindelhauer, and Berthold Vöcking. Randomized rumor spreading. In *FOCS'00: 41st Symposium on Foundation on Computer Science*, page 10, 2000.
- [LLS06] Mei Li, Wang-Chien Lee, and Anand Sivasubramaniam. Dptree: A balanced tree based indexing framework for peer-to-peer systems, 2006.
- [MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. pages 183–192, 2002.
- [MS06] Peter Mahlmann and Christian Schindelhauer. Distributed random di-graph transformations for peer-to-peer networks. In *SPAA'06: Proceedings of the eighteenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, page 10, 2006.
- [NW02] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. pages 50–59, 2002.
- [RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Lecture Notes in Computer Science*, pages 329–350, 2001.
- [Rie] Simon Rieche. Range queries and load balancing in a hierarchically structured p2p system.
- [RRHS04] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. Technical report, 2004.
- [SMK⁺01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160, 2001.
- [SS05] Christian Schindelhauer and Gunnar Schomaker. Weighted distributed hashing. In *SPAA'05: Proceedings of the seventeenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, page 10, 2005.
- [Wik09a] Wikipedia. Inversionsmethode — wikipedia, die freie enzyklopädie. <http://de.wikipedia.org/w/index.php?title=Inversionsmethode&oldid=58744013>, 2009. [Online; Stand 1. Juni 2009].
- [Wik09b] Wikipedia. Logarithmus — wikipedia, die freie enzyklopädie. <http://de.wikipedia.org/w/index.php?title=Logarithmus&oldid=60989497#Potenzreihe>, 2009. [Online; Stand 15. Juni 2009].

- [Wik09c] Wikipedia. Round trip time — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=Round_Trip_Time&oldid=59010570, 2009. [Online; Stand 13. Mai 2009].
- [Wik09d] Wikipedia. Zipfsches gesetz — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=Zipfsches_Gesetz&oldid=60635362, 2009. [Online; Stand 1. Juni 2009].
- [ZCB96] Ellen Zegura, Kenneth Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *In Proceedings of IEEE INFOCOM*, pages 594–602, 1996.
- [ZSLS06] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over dht. In *In Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, 2006.

Glossar

Baumzweig	oder auch einfach Zweig bezeichnet einen Teilbaum in einem Baum, der durch einen eindeutigen Pfad von der Wurzel identifiziert ist. In einem gewichteten Baum ist das Gewicht eines Baumzweigs gegeben durch die Summe aller Knoten in diesem Baumzweig.
Bereichsanfrage	(engl.: range query) bezeichnet die Suche nach allen Elementen in einem eingeschränkten Datenbereich (z.B. alle Namen von 'm' bis 'z' aus dem Telefonbuch).
Congestion	(engl. "Überlastung", "Stau") Die Congestion eines Peers bezeichnet die Anzahl an Routingpfaden, die im Lookup-Algorithmus durch den Peer gehen.
DHHT	Verteilte heterogene Hashtabellen (Distributed Heterogeneous Hash Tables) bilden Daten auf Knoten eines Netzwerks mit einer heterogenen Kapazität ab. Hierbei erhalten Knoten mit einer höheren Kapazität mehr Daten (siehe Kapitel 2.1.1).
DHT	(engl.: Distributed Hash Table) eine verteilte Hashtabelle ist eine verteilte Datenstruktur, bei der im verteilten System ähnlich wie bei einer Hashtabelle für einen Schlüssel effizient der assoziierte Wert ermittelt werden kann. Dabei werden die Daten möglichst gleichmäßig auf die Knoten verteilt.
Hashfunktion	(engl. "zerstückeln") bezeichnet eine Funktion, die aus einer Eingabe einen Hashwert fester Länge (hier 128-bit) erzeugt, der eine möglichst eindeutige Kennzeichnung der Eingabe darstellt. Eingaben sind hier Schlüssel der Daten oder eine Identifikation eines Peers (z.B. Netzwerkadresse).
Hashtabelle	ist eine Indexstruktur, die eine Hashfunktion zur effizienten Abbildung von Schlüsseln auf assoziierte Werte benutzt.
Hop	(engl. "Hopser") bezeichnet den Weg im Netzwerk von einem Peer zum nächsten. Ein P2P-Netzwerk ist ein Multi-Hop-Netz, in dem man meist einen Pfad mit mehreren Hops von einem Peer zu einem anderen Peer benötigt.

ID	bezeichnet eine eindeutige Identifizierung (Schlüssel) eines Objekts wie einem Peer oder einem Datenelement.
Join	(engl. "beitreten", "verbinden") bezeichnet hier die Operation zur Verbindung eines neuen Peers mit einem bestehenden Peer-to-Peer-Netzwerk
Latenz	bezeichnet die Übertragungszeit eines Datenpakets im Netzwerk von einer Quelle zu einem Ziel. Dies ist im Durchschnitt der halbe Ping.
Link	(engl. "Zeiger") bezeichnet hier eine gerichtete Kante von einem Peer auf einen anderen. ein Link enthält die Netzwerkadresse des Peers, über die man die Kommunikation zum Peer aufbauen kann.
Lookup	(engl. "nachschiagen") ist eine Funktion, die für eine ID den zuständigen Peer in einem P2P-Netzwerk zurückgibt.
Peer	(engl. "Gleichgestellter", "Ebenbürtiger") bezeichnet einen Endpunkt einer Kommunikation in einem Computernetzwerk. Jeder Peer bietet dabei seine Dienste an und nutzt die Dienste anderer Peers.
Peer-to-Peer	bezeichnet die Kommunikation unter gleichberechtigten Netzwerkpartnern.
Ping	siehe RTT
Routing	(engl. "Lotsen", "Wegewahl", "Verkehrslenkung") bezeichnet das Festlegen des Pfads, der von einem Peer zu einem anderen im Netzwerk führt.
Routingtabelle	bezeichnet eine Tabelle mit Links auf Peers im Netzwerk. Ein Peer benutzt die Einträge seiner Routingtabelle um beim Routing eine Nachricht zu einem bestimmten Peer weiterzuleiten.
Routingpfad	bezeichnet den Pfad von einem Peer zu einem bestimmten anderen Peer im Netzwerk.
RTT	die Round-Trip-Time oder auch Ping bezeichnet die Umlaufzeit, die ein Datenpaket von einer Quelle zu einem Ziel und wieder zurück zur Quelle benötigt.
Suchbaum	bezeichnet eine auf Bäumen basierende Datenstruktur, in der man in einer geordneten Menge von Elementen effizient suchen kann.
Trie	bezeichnet einen Präfixbaum. Darin repräsentiert jede Kante ein Zeichen. Jeder Knoten entspricht dann der Zeichenkette, die aus der Verkettung aller Kantenzeichen von der Wurzel entsteht. Die Wurzel ist dann eine leere Zeichenkette.

A

Screenshots der Implementierung

In diesem Kapitel werden abschließend Screenshots der Implementierung gezeigt. Eine Liste aller Netzwerkeinstellungen aus den Dialogen in Abbildung A.1 gibt es noch mal im nächsten Kapitel B für die Messungen. Die Messwerkzeuge befinden sich alle im Menü "Measurements" im Hauptfenster der Testumgebung (siehe Abb. A.1).

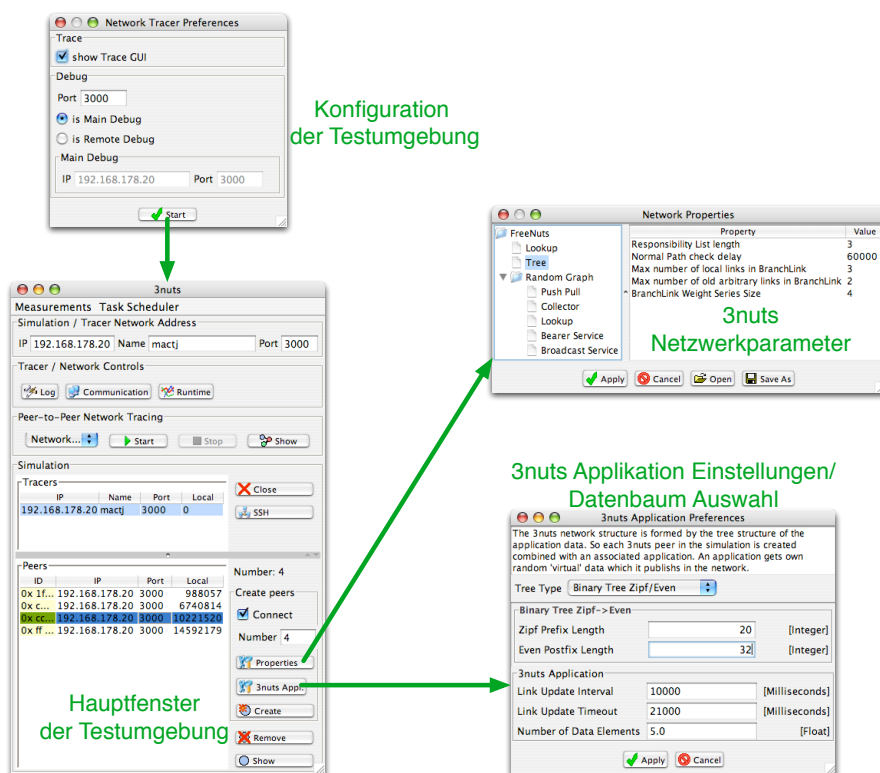


Abbildung A.1.: Konfiguration eines 3nuts Netzwerks

A. Screenshots der Implementierung

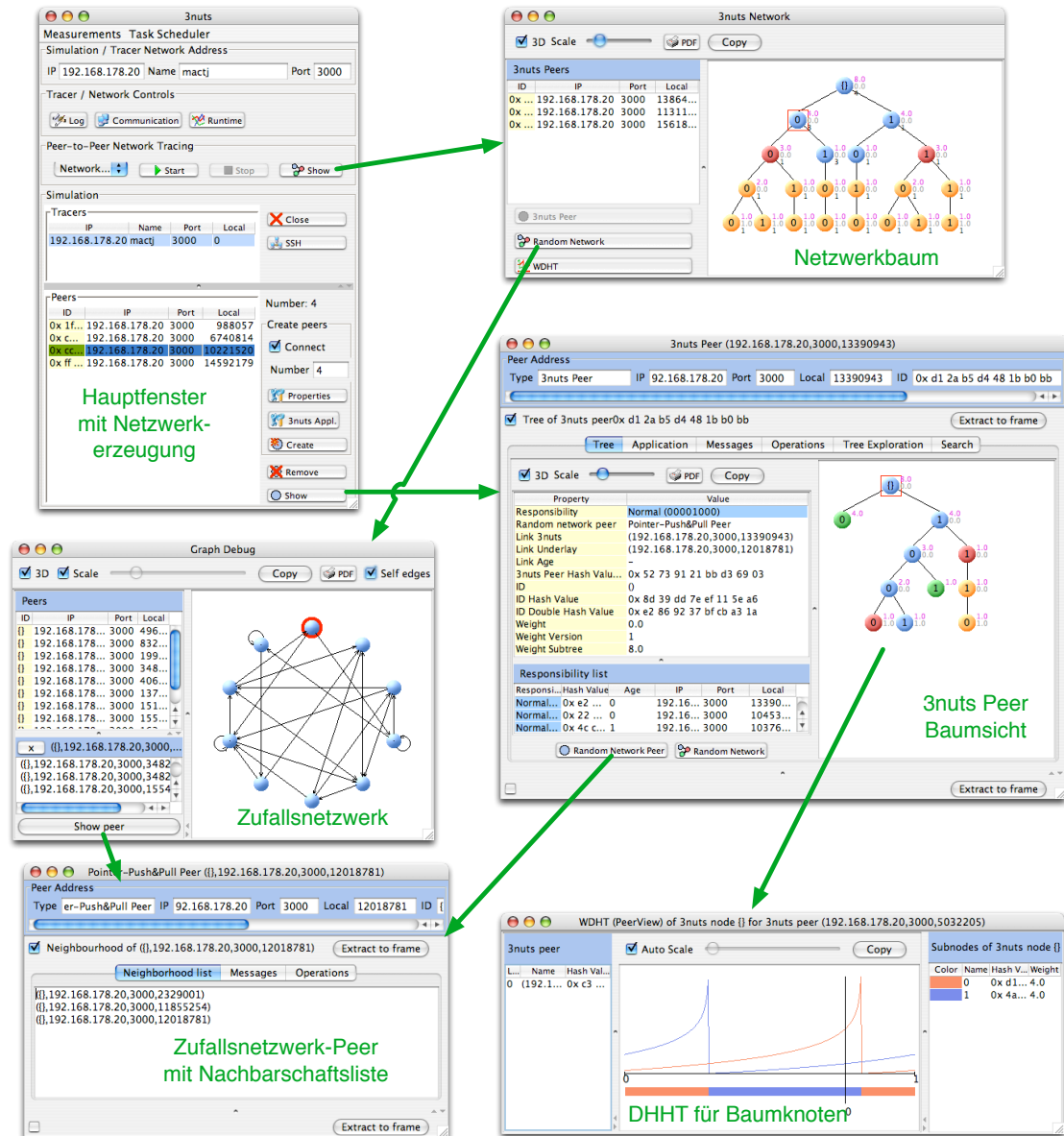


Abbildung A.2.: Erzeugung/Ansicht eines 3nuts Netzwerks

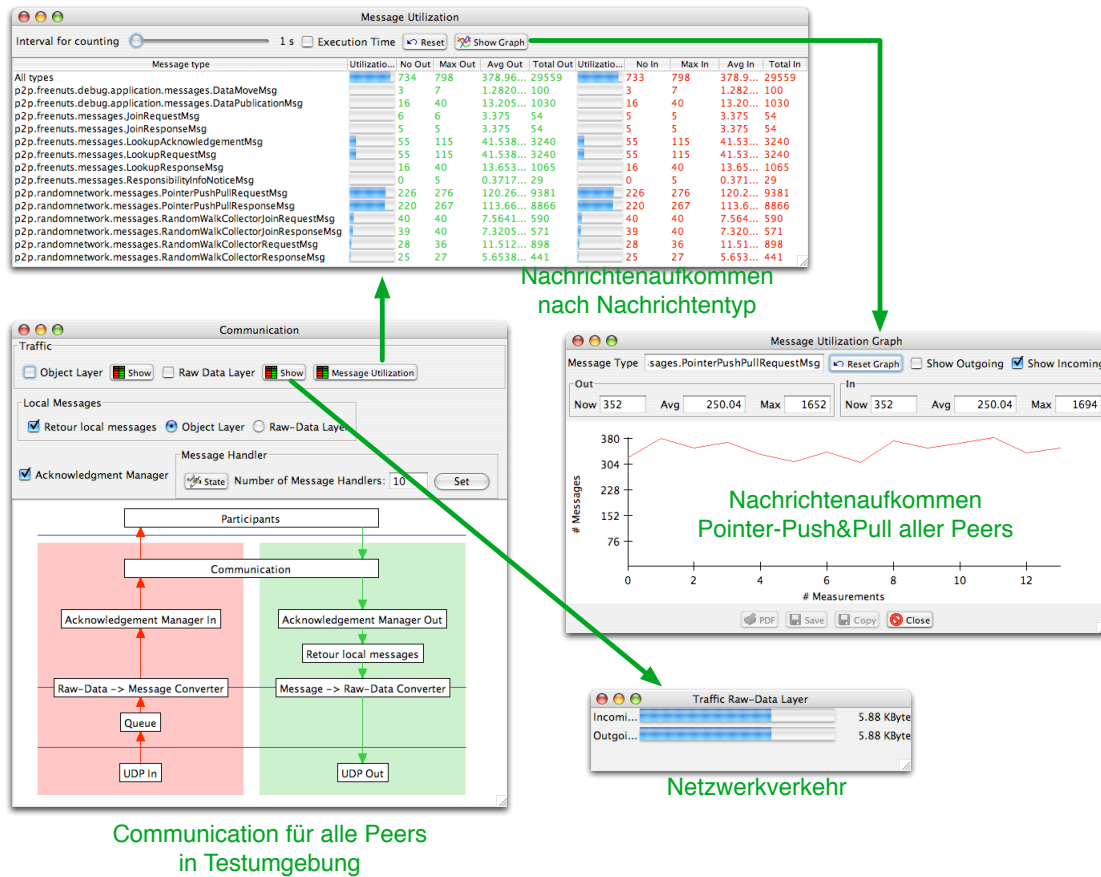


Abbildung A.3.: Ansicht der Netzwerkschicht mit der Communication

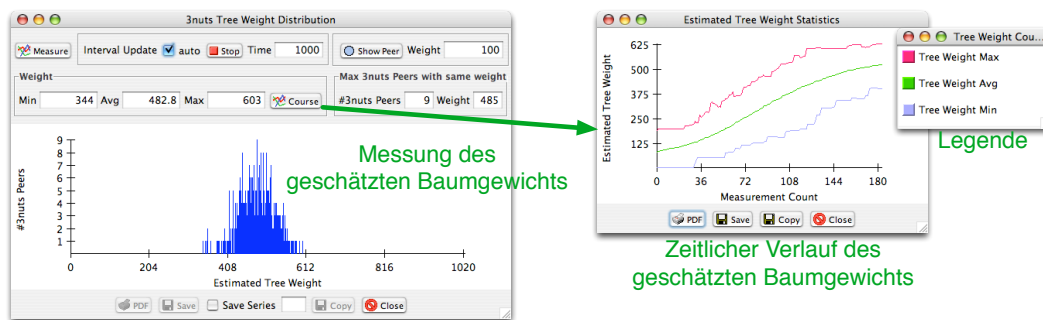


Abbildung A.4.: Messung des Baumgewichts

A. Screenshots der Implementierung

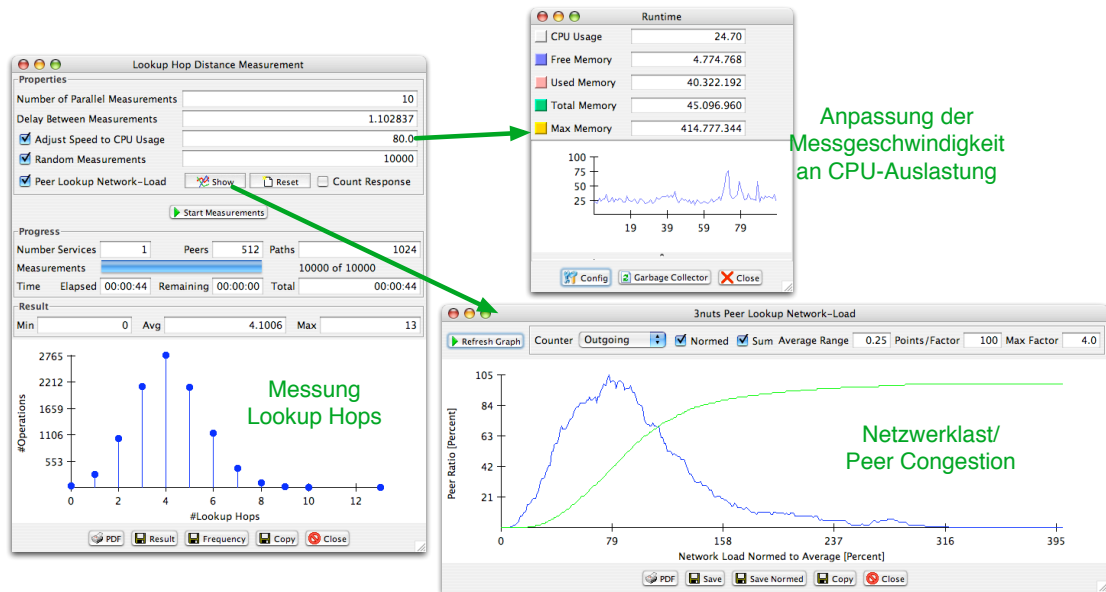


Abbildung A.5.: Messung der Lookup Hops und Peer Congestion

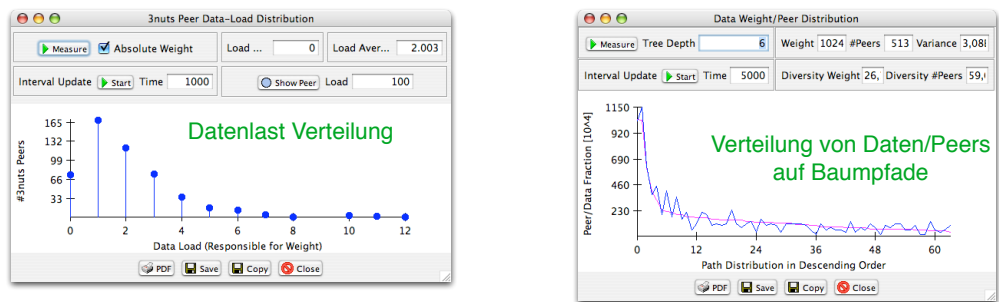


Abbildung A.6.: Messung der Datenlast bzw. Aufteilung der Daten/Peers auf Baumfäde

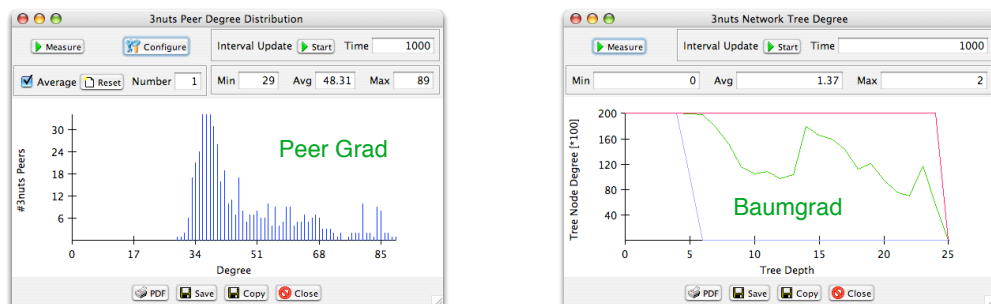


Abbildung A.7.: Messung des 3nuts Peer Grads und dem Baumgrad

B

Messparameter

In diesem Kapitel werden kurz alle Einstellungen der 3nuts Netzwerke in den Messungen präsentiert. Abbildung B.2 zeigt die Konfiguration der Testumgebung, welche über Konfigurationsdateien vor dem Start erfolgt. Die einstellbare Startdatei der Testumgebung für 3nuts heißt `run_3nuts` und liegt im Hauptverzeichnis. Alle weiteren Konfigurationsdateien befinden sich im Unterverzeichnis `preferences/`. Da bei den

TaskScheduler Größen

3nuts (preferences/TaskScheduler_3nuts.txt)

3nuts Peer: Array Size Lookup Acknowledgment	200
3nuts Peer: Array Size Join Timeout	4
3nuts Peer: Max Threads Move Data	50
TrunkNode: Max Threads Responsibility Changed	100
TrunkNode: Max Threads Move Data	50
TrunkNode: Max Threads Move Data	50

Random Network (preferences/
TaskScheduler_3nuts_PointerPushPull.txt)

Use own Threads	false
Array Size Lost Connection Timeout	4
Array Size Ping	4
Array Size Pointer-Push&Pull	250
Array Size Locked Addresses Timeout	250
Array Size Random Walk Timeout	4

3nuts Application (preferences/
TaskScheduler_3nuts_Application.txt)

Data Publication: Array Size	4
Data Publication: Max Threads	256
Data Link Check: Array Size	50

3nuts Debug Parameter

Java Startup (run_3nuts)

Extended Memory	-mx2000m
Classpath	-classpath bin
Main Class	-p2p.freenuts.debug. FreenutsDebug
Network Properties	-configFile=preferences/ 3nuts.xml
Debug Properties	-debug(3000,isSim,Main)
Message Queue	-messageQueue=500000

Verzögerung zwischen Peererzeugung

(preferences/
PeerCreationPerformance_3nuts.txt)

bis 1.000 Peers	10 ms
bis 5.000 Peers	150 ms
bis 10.000 Peers	180 ms
bis 100.000 Peers	180 ms
ab 100.000+ Peers	180 ms

Abbildung B.1.: Konfiguration der Testumgebung

Messungen viele tausend Peers die TaskScheduler gemeinsam benutzen, sind hier die TaskScheduler in große Arrays TaskScheduler.Array mit bis zu 250 kleinen TaskScheduler unterteilt, wo die Verwaltung der kleineren Zeitpläne dann performanter ist (Vgl.

B. Messparameter

Kapitel 3.1.2). So ergibt sich hier auch eine hohe Nebenläufigkeit. Nach der Konfiguration kann die Testumgebung dann mit dem Skript `run_3nuts` gestartet werden.

Die Konfiguration des Netzwerks erfolgt in der Testumgebung selbst. Abbildung B.2 zeigt in einer Übersicht alle Einstellungen. Die Einstellung des 3nuts Netzwerkpara-

3nuts Netzwerk Eigenschaften

3nuts

Lookup Timeout	16.000 ms
Lookup Acknowledgment	800 ms
Responsibility List Length	3
Normal Path Check Delay	300.000 ms
BranchLink: Max Local Links	3
BranchLink: Max Old Arbitrary	2
BranchLink: Weight Series Size	4

Zufallsnetzwerk

Regular Degree	3
Neighbors Old Tested	3
Neighbors Old Untested	1
Pointer-Push&Pull Interval	16-20.000ms
Pointer-Push&Pull Timeout	10.000 ms
Ping (deaktiviert)	0 ms
Collector: Number per Hop	2
Collector: Fail Number Join Max	4
Collector: Timeout	4.000 ms
Lookup: Hops (nicht benutzt)	10
Lookup: Timeout (nicht benutzt)	1.000 ms
Bearer: Lost Connection Timeout	300.000 ms
Broadcast Interval (deaktiviert)	0 ms

3nuts Applikation Eigenschaften

3nuts Application

Link Update Interval	90.000 ms
Link Update Timeout	190.000 ms
Number of Data Elements	5

Verwendete Datenbäume

Zipf-verteilter Binärbaum

Tree Type	Binary Tree Zipf->Even
Zipf Prefix Length	20
Even Postfix Length	32

Gleichverteilter Binärbaum

Tree Type	Binary Tree Thin and Deep
Path Number	40

Wörterbuch-Baum mit Zeichen als Knoten

Tree Type	File List Tree
Filename	Dictionary_EN_DE -> 129.872
Tree Type	Char

Netzwerk Topologie

Delay Msg	only-Pointer-Push&Pull
Sphere	Radius 320
GT-ITM	ts11025-0.delay

Abbildung B.2.: Messparameter des 3nuts Netzwerks und deren Applikationen

meter erfolgt über einen Dialog, der über den Knopf "Preferences" im Hauptfenster geöffnet werden kann. Die Timeout-Parameter sind bei den Netzwerk Eigenschaften wegen der großen 3nuts Netzwerke in nur einer Java Virtual Machine gesetzt. Netzwerke mit bis zu 20.000 Peers benötigen zusammen einen großen Heap-Speicher und der Java Garbage Collector benötigt dann kurzzeitig viel Leistung um den Speicher aufzuräumen. Durch die hohen Timeouts soll vermieden werden, dass die Operationen hier fälschlicherweise die Fehlerkontrolle anstoßen. In einem anderen Dialog, erreichbar über den Knopf "3nuts Appl." im Hauptfenster, können die Applikationen konfiguriert werden, die die 3nuts Peers benutzen. Dabei ist einer der 3 Datenbäume Wörterbuch, gleich- bzw. Zipf-verteilter Binärbaum auszuwählen. Die Konfiguration und Aktivierung der Simulation des physikalischen Netzwerks mit einer Netzwerk-Topologie erfolgt direkt im Fenster der Latenzmessungen. Das Fenster der Latenzmessungen ist erreichbar über das Menü "Measurements" im Hauptfenster und dem Menüpunkt "Network Locality Performance".