Lookup Performanz von Verteilten Hashtabellen

Martin Christian <martin_at_christianix_dot_de>

Leipzig, 19. März 2006

1 Einleitung

Das Thema dieses Vortrags ist die Besprechung des Artikels *Improving Lookup Performance over a Widley-Deployed DHT* von Daniel Stutzbach und Reaz Rejaie. Zur Einführung werde ich Verteilte Hashtabellen anhand von Chord, CAN und Kademlia vorstellen, wobei Aufbau und Funktion von Letzterem ausführlicher behandelt wird. Anschließend stelle ich den Artikel selbst vor.

2 Verteilte Hashtabellen

Peer-to-Peer (P2P) Systeme lassen sich nach Art der Ressourcenverwaltung in folgende Kategorien einteilen:

- Hybride P2P-Systeme
  - zentralistisch: ein zentraler Server verwaltet die Zuordnung von Ressourcen zu Peers, z. B. Napster
  - hierarchisch: eine Hierarchie von Servern übernimmt die Verwaltung der Ressourcen, z. B. DNS

- Reine P2P-Systeme
  - unstrukturiert: zwischen Nachbarn gibt es keine Unterschiede, z. B. Gnutella
  - strukturiert: Nachbarn stehen in einem Verhältnis zueinander, z. B. Verteilte Hashtabellen

Die *Distributed Hashtables (DHT)* oder zu deutsch Verteilten Hashtabellen gehören zu den reinen P2P-Systemen mit strukturierter Nachbarschaft. Reine P2P-System haben den Vorteil, dass die Funktionsfähigkeit nicht von einzelnen Server abhängt (Flaschenhals, Single-Point-of-Failure). Dagegen skalieren reine, unstrukturierte P2P-Systeme schlechter. Das liegt daran, dass sie beim Routing von Anfragen nicht zwischen geeigneten und nicht geeigneten Knoten unterscheiden können. Sie müssen Anfragen also an alle ihre Nachbarn *broadcasten*. Reine, strukturierte P2P-Systeme kennen dagegen ein Maß für die Nützlichkeit eines Knotens bezüglich einer vorliegenden Anfrage. Bei DHTs wird ein Distanzmaß definiert, welches auf der Struktur des Netzes beruht. Zum Suchen (Lookup) eines Schlüssels stellen DHTs eine Schnittstelle, bestehend aus der Funktion `lookup(key)`, zur Verfügung [**?**]. Das Auffinden von Daten in einem DHT läuft folgendermaßen ab:

- Generiere den Hashwert des Schlüssels (key). Als Schlüsselwerte kommen nur allgemein bekannte Eigenschaften der Daten in Betracht, z. B. der Dateiname.
- Suche den Knoten der diesen Schlüssel[1] verwaltet.
- Entweder kann diesem Knoten jetzt das Tupel `(key, data)` zur Speicherung gesendet oder es können die Daten angefordert werden.

Das Verwalten der (key, data) Tupel in den Knoten gehört nicht zu den Aufgaben eines DHT. Er stellt nur die Mittel bereit, gegeben einen Schlüssel, den verantwortlichen Netzwerkknoten zu finden.

[1] Bei der Suche nach dem passenden Knoten wird das Bild des Schlüssel unter der Hashfunktion verwendet.

Der Einfachheit halber werden die Knoten in den Schlüsselraum eingebettet. D. h. die Hashfunktion erzeugt für Schlüssel und Knoten numerische Werte gleicher Länge. Das vereinfacht die Zuordnung von Schlüsseln zu Knoten, da der Abstand eines Schlüssels zu einem Knoten direkt berechnet werden kann. In [**?**] wird sowohl der Schlüssel als auch sein Bild unter der Hashfunktion als Schlüssel bezeichnet. Gleiches gilt für Knoten. Dem schließe ich mich an.

Auf Basis der in [**?**] genannten Entwurfsmerkmale von DHTs habe ich folgende Vergleichskriterien entwickelt:

1. Struktur des (gehashten) Schlüsselraums
2. Distanzmaß im Schlüsselraum
3. Datenstruktur zum Weiterleiten (Routing) von Anfragen

2.1 Chord

Bei Chord [**?**] werden Schlüssel und Knoten in einem Restklassenring $\mathbb{Z}/2^m$ angeordnet, dabei ist m die Schlüssellänge in Bit. Eine geeignete Hashfunktion verteilt die Schlüssel auf $[0,\ 2^m - 1]_{\mathbb{Z}}$ gleichmäßig. Sei k ein beliebiger Schlüssel, dann soll gelten:

succ(k) bezeichnet denjenigen Knoten n mit: $n \geq k$ und es gibt keinen anderen Knoten zwischen n und k.

Aus Effizienzgründen speichert ein Knoten keine Liste aller Knoten, sondern eine Sprungliste (skiplist) mit maximal m Einträgen. Die Einträge der Sprungliste werden $finger$ genannt. Jeder $finger$ deckt einen exponentiell wachsenden Bereich des Schlüsselrings ab:

Position	finger.start	finger.end	Nachfolger
1	$n+1$	$< n+2$	succ$(n+1)$
...			
i	$(n+2^{i-1})\ mod\ 2^m$	$< (n+2^i)\ mod\ 2^m$	succ(finger$[i]$.start)
...			
m	$(n+2^{m-1})\ mod\ 2^m$	$< n$	succ(finger$[m]$.start)

2.1.1 Eigenschaften

1. Schlüsselraum: Restklassenring $\mathbb{Z}/2^m$ mit m Bit Schlüsseln
2. Distanzmaß: $d(x,y) = 2^m - (x-y)\ mod\ 2^m$
3. Routing: Sprungliste mit $1 \leq i \leq m$ Einträgen $succ(\ (n+2^{i-1})\ mod\ 2^m\ )$

2.2 CAN

Das *Content Adressable Netzwork* (CAN) [**?**] verwendet einen d-dimensionalen kartesischen Schlüsselraum. Jede Dimension besteht aus dem Zahlenbereich $[0,1]$. Der Koordinatenraum wird in Hyperrechtecke eingeteilt, den so genannten Zonen. Eine Zone wird von genau einem Knoten verwaltet. Alle Schlüssel in einer Zone haben einen minimalen euklidischen Abstand $d(\vec{x},\vec{y})$ zum Knoten der Zone. Kommt ein neuer Knoten hinzu, wir eine bestehende Zone aufgeteilt und die Schlüssel entsprechend verteilt.

Das Suchen eines Knotens läuft derart ab, dass jeder Knoten die Anfrage an die benachbarte Zone weiterleitet, die dem Schlüssel am nächsten ist. Der Weg einer Anfrage ist die Näherung einer gerade Linie zwischen Start- und Zielknoten.

2.2.1 Eigenschaften

1. Schlüsselraum: $[0,\ 1]^d \subset \mathbb{R}^d$
2. Distanzmaß: $d(\vec{x},\vec{y}) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$

3. Routing: Nachbarzone mit dem geringsten Abstand zum gesuchten Schlüssel

2.3 Kademlia

Kademlia [**?**] ordnet die Schlüssel nicht in einem Ring an, sondern nach Ähnlichkeit der Bitfolge. Die Ähnlichkeit ist definiert als bitweise XOR-Verknüpfung: $d(x, y) = x \oplus y$. Je mehr höherwertige Bits zwei Schlüssel gemeinsam haben, desto näher liegen sie beisammen. Als Hashfunktion wird SHA-1 mit einem 160 Bit großen Bildbereich vorgeschlagen.

2.3.1 Routingtabellen

Wie in Chord verwaltet jeder Knoten nur eine kleine Untermenge der Knoten im Netz in einer Routingtabelle. Jeder Eintrag in der Routingtabelle beinhaltet eine Liste von maximal k Knoten:

Intervall	k-Bucket
$[2^0, 2^1[$	$1 \leq r \leq k$ Knoten
...	...
$[2^i, 2^{i+1}[$	$1 \leq r \leq k$ Knoten
...	...
$[2^{159}, 2^{160}[$	$1 \leq r \leq k$ Knoten

Diese sogenannten k-Buckets sind notwendig, um im Netz trotz Zu- und Abwanderungen mit hoher Wahrscheinlichkeit den gesuchten Knoten zu finden. Die Suche nach Knoten läuft bei Kademlia iterativ ab. D. h. die Suchanfragen werden nicht von Knoten zu Knoten weitergeleitet, sondern jede Antwort geht an den Suchenden zurück. Auf eine Anfrage antwortet ein Knoten entweder mit dem Schlüssel oder mit einer Liste von Knoten die näher am Ziel liegen.

2.3.2 Routingtabellen als Präfixbaum

Die Routingtabelle kann man auch als Baum auffassen, bei dem jeder innere Knoten eine Bitposition im Schlüssel repräsentiert und die k-Buckets die Blätter sind. Bei jedem inneren Knoten wird getestet, ob die Bits übereinstimmen. Bei Übereinstimmung geht der Algorithmus zum nächsten Knoten, sonst in das k-Bucket. Der Wurzelknoten repräsentiert das höchstwertige Bit. Stimmt das nicht überein, ist der gesuchte Knoten maximal weit weg. Danach wird die Wertigkeit der Bits in jedem Schritt verringert. D. h. der Abstand hängt von der Gleichheit des Bitpräfixes ab, daher auch der Name Präfixbaum.

2.3.3 Subnetze

Knoten mit den gleichen x höchstwertigen Bits, bilden das $/x$ Subnetz. Z. B. befinden sich die Knoten $n_1 = 1011$ und $n_2 = 1001$ im $/2$ Subnetz. Diese Notation stammt vom Internetprotokoll (IP). Bei Kademlia gibt es 161 Subnetze ($[0, 160]$), wobei in Subnetz $/0$ kein Bits und in Subnetz $/160$ alle Bits übereinstimmen[2]. Die Knoten eines k-Buckets bilden in diesem Sinne ein Subnetz:

Subnetz	Intervall
$/159$	$[2^0, 2^1[$
$/i$	$[2^i, 2^{i+1}[$
$/0$	$[2^{159}, 2^{160}[$

2.3.4 Eigenschaften

1. Schlüsselraum: Symmetrische Maschen mit 160 Bit Schlüsseln
2. Distanzmaß: $d(x, y) = x \oplus y$
3. Routing: Präfixbaum mit 160 inneren Knoten und k-Buckets an den Blättern

[2] Wobei fürs Routing nur die ersten 160 Bit relevant sind.

2.3.5 Weitere Eigenschaften

1. Die k-Buckets erhöhen die Wahrscheinlichkeit, dass der gesuchte Knoten gefunden wird, da mehrere Knoten aus dem selben Subnetz zur Auswahl stehen.

2. Die Pflege (Stabilisation) der Routingtabelle erzeugt nur wenig Overhead an Nachrichten, denn die Routingtabelle wird durch Arbeitskontakte gefüllt und aktualisiert. D. h. wenn ein Knoten a von einem Knoten b das erste Mal kontaktiert wird, trägt a den Knoten b in das entsprechende k-Bucket ein. Enthält das k-Bucket bereits k Einträge, wird der älteste (unterste) Kontakt angepingt. Antwortet er nicht, so wird er gelöscht und b eingetragen. Falls er antwortet, wird b verworfen.

3. Dies hat den weiteren Vorteil, dass k-Buckets nicht *vergiftet* werden können, denn aktive Knoten werden nicht gelöscht.

3 Lookup Effizienz

Die Autoren von [**?**] sehen den Nachteil bisheriger Forschungsergebnisse darin, dass sie sich auf DHTs unter Laborbedingungen beschränkt haben. Außerdem würde die überwiegende Mehrheit nur das Worst-Case Verhalten der Lookup-Operation betrachten. Daher haben sie das durchschnittliche Laufzeitverhalten der Lookup Operation im Kad Netz von eMule untersucht. Kad ist eine im Feldeinsatz befindliche Implementierung eines Kademlia DHT mit knapp einer Million Nutzer und einigen Bugs.

Die Performanz eine Lookup Operation wird an drei Faktoren gemessen:

1. Wieviele Schritten (Hops) braucht eine Anfrage bis sie zum Ziel kommt?

2. Wie lange Dauert eine Anfrage (Latenzzeit)?

3. Mit welcher Sicherheit findet ein Lookup vorhandene Daten?

Daher geht es den Autoren darum,

- zuerst ein theoretisches Modell zur Abschätzung der mittleren Anzahl Hops in DHTs zu entwickeln, dann
- verschiedene Routing-, Lookup- und Replikationsstrategien empirisch zu vergleichen, um schließlich
- fundierte Vorschläge zur Verbesserung von eMule und anderen Präfix-basierte DHTs unterbreiten zu können.
- Dabei sollen neue Werkzeuge zur Untersuchung realer DHTs entstehen.

Für die Anzahl Hops ist die Qualität der Routingtabelle ausschlaggebend. Eine Verbesserung der Lookup Performanz kann über die Routingtabelle auf zwei Arten erreicht werden:

- Man erhöht die Anzahl Einträge in den k-Buckets.
- Man erhöht die Anzahl Einträge (k-Buckets) in der Routingtabelle. Dies kann auf zwei Arte geschehen:
  - Die Anzahl Blätter wird erhöht indem größere Bitblöcke (Symbole) verglichen werden.
  - Die Anzahl Bits, die pro Knoten verglichen werden (Auflösung), wird erhöht.

Diese beide Ansätze sollen nun miteinander verglichen werden.

3.1 Redundanz bringt Performanz

Neben der Redundanz besitzen k-Buckets einen weiteren Vorteil: Sie erhöhen die Chance zufällig näher ans Ziel zu kommen. Seien z. B. x und y Zahlen in Binärdarstellung, dann ist die Wahrscheinlichkeit, dass das obere Bit übereinstimmt

$\frac{1}{2}$, dass die beiden oberen Bits übereinstimmen $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. Übertragen auf Kademlia entspricht das der Wahrscheinlichkeit, dass mehr Bits in einem k-Bucket übereinstimmen, als die Netzmaske des Buckets. Seien die Schlüssel vier Bit lang und das k-Bucket $/2$ von Knoten 0000 gegeben. Dann gibt es es höchstens vier unterschiedliche Einträge im k-Bucket:

0000	0001	0010	0011

Die Wahrscheinlichkeit, dass man dem Ziel um mindestens δ Bits näher kommt ist also:

$$Pr[X \geq \delta] = \frac{1}{2^{\delta}}$$

In einem vollen k-Bucket hat man k-Mal die Chance δ Bits näher ans Ziel zu kommen. Pro Knoten können auch mehr Bits gleichzeitig verglichen werden. Die Anzahl Bits, die pro Knoten verglichen werden, ist r. Die Wahrscheinlichkeit, in einem k-Bucket mindestens δ Routingschritte näher ans Ziel zu kommen ist:

$$F(\delta, r, k) = Pr[X \geq \delta] = 1 - \left(1 - \frac{1}{2^{r\delta}}\right)^{k}$$

D. h. je größer k ist, desto größer ist die Wahrscheinlichkeit, dass man sich mindestens δ Routingschritte dem Ziel annähert, denn man hat mehrmals die Chance, sich mindestens um die gewünschte Anzahl Bits zu verbessern. Pech hat man nur, wenn alle k Ziehungen unter der gewünschten Verbesserung liegen. Aus diesem Grund wird die Wahrscheinlichkeit über das Komplement berechnet.

Die Wahrscheinlichkeit genau δ Routingschritte näher ans Ziel zu kommen ist:

$$f(\delta, r, k) = Pr[X = \delta] = F(\delta, r, k) - F(\delta + 1, r, k)$$

Der Erwartungswert für die zufällige Bitverbesserung ist:

$$m(r, k) = r \cdot \sum_{\delta=0}^{\infty} \delta \cdot f(\delta, r, k)$$

Durch die Symbolgröße b, d. h. der Anzahl k-Buckets pro Knoten im Routingbaum, nähert man sich mindestens b Bits pro Routingschritt dem Ziel an. Die gesamte Bitverbesserung pro Schritt ergibt sich aus der Summe von minimalem (garantiertem) und zufälligem Bitgewinn:

$$t(b, r, k) = b + m(r, k)$$

Mit Hilfe des Bitgewinns läßt sich die mittlere Anzahl Hops für einen Lookup abschätzen:

$$\text{steps}_{\text{avg}} = \frac{\log_2(n)}{t(b, r, k)}$$

3.2 Symbolgröße und Performanz

Kademlia nutzt in der Originalversion von [**?**] 1-Bit große Symbole. Dies lässt sich jedoch zu b-Bit großen Symbolen erweitern. Je größer b ist, desto weniger Schritte sind notwendig um ans Ziel zu kommen: $\log_{2^b} n$ Hops. Stellt man sich die Routingtabelle als Baum vor, werden pro innerem Knoten b-Bit verglichen und $2^b - 1$ k-Buckets verwaltet. Will man die Anzahl Blätter reduzieren, kann man pro innerem Knoten nur r-Bit vergleichen, mit $r \leq b$. Mit Hilfe dieser Parameter lassen sich Kademlia System klassifizieren: $\mathcal{D}(b, r, k)$ bezeichnet ein System mit b-Bit Symbolen, einer Auflösung von r-Bit pro innerem Knoten und k Einträgen pro Buckets. Damit lassen sich drei Grundsysteme beschreiben:

Basic Kademlia $\mathcal{D}(1, 1, k)$ bedeutet, dass jeder innere Knoten 1-Bit vergleicht und 1 Blatt mit k-Einträgen hat.

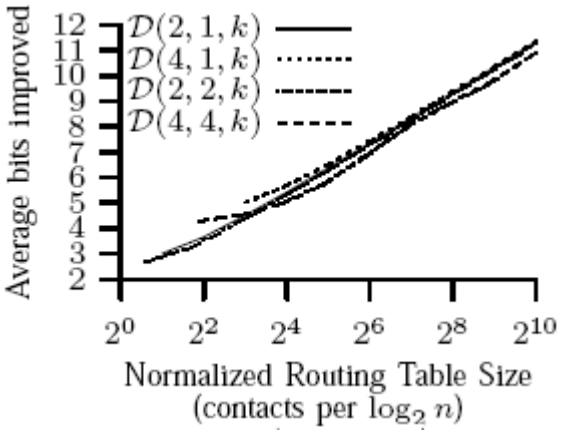
Discrete Symbols $\mathcal{D}(b, b, 1)$ bedeutet, dass jeder innere Knoten b-Bits vergleicht und $2^b - 1$ Blätter mit jeweils einem Eintrag hat.

Split_Symbols $\mathcal{D}(b, 1, 1)$ bedeutet, dass jeder innere Knoten 1-Bit vergleicht und 2^{b-1} Blätter mit jeweils einem Eintrag hat.

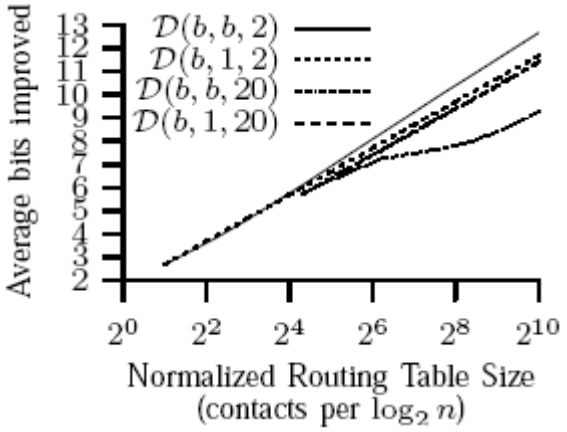
Eine Routingtabelle enthält $k \cdot (2^b - 2^{b-r}) \cdot \log_{2^r} n$ Einträge. Normalisiert man die Größe der Routingtabelle mit dem Faktor $\log_2 n$, erhält man die um n bereinigte Größe: $k \cdot \frac{2^b - 2^{b-r}}{r}$.

3.3 Ergebnisse

Anfangs wurde die Frage gestellt, wie sich Veränderungen der Routingtabelle oder der k-Buckets auf die Hopanzahl auswirken. Variiert man Symbolgröße und Auflösung ergibt sich keine wesentliche Bitverbesserung im Vergleich zum deutliche höheren Platzbedarf der Routingtabellen:



Auch die Kombination von k-Buckets und diskreten Symbolen bringt keine Steigerung im Vergleich zur reinen Redundanz:



Fasst man die Ergebnisse dieser Untersuchung zusammen, lässt sich folgendes feststellen:

- Größere Symbole bringen eine konstante Verbesserung der Laufzeit im schlechtesten Fall.
- Neben der Redundanz erhöhen k-Buckets auch die durchschnittliche Performanz und sind einfach zu implementieren.
- Größere Symbole mit k-Buckets zu mischen bringt keine Performanzgewinne.

3.4 Kad

Kad hat Routingtabellen in der Form $\mathcal{D}(3.25, 1, 10)$. Die Symbolgröße $b = 3.25$ kommt daher, dass Kad an jedem Knoten fünf Blätter hat: `1000, 1001, 101, 110, 111`. Daraus ergibt sich eine mittlere Verbesserung von $t(3.25, 1, 10) = 3.25 + m(1, 10) = 6.98$ Bits pro Schritt. Der Wurzelknoten hat sogar eine vollständige 4 Bit Verästelung und somit eine mittlere Verbesserung von $t(4, 1, 10) = 4 + m(1, 10) = 7.73$ Bits im ersten Schritt. Damit ergibt sich die mittleren Anzahl Hops für Kad:

$$\text{steps}_{\text{avg}} = 1 + \frac{\log_2(n) - t(4, 1, 10)}{t(3.25, 1, 10)} = 1 + \frac{\log_2(n) - 7.73}{6.98}$$

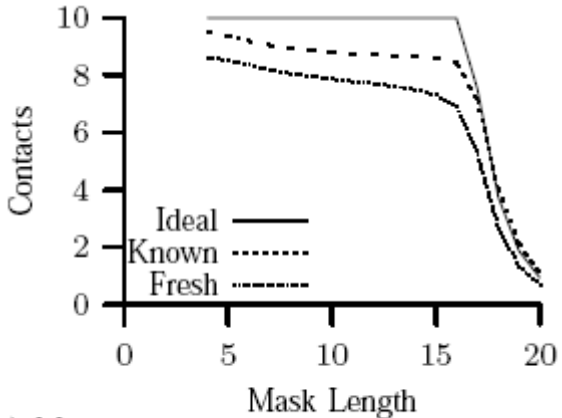
Diese Formel hängt nur noch von der Größe des Netzes n ab. Aufgrund der ständigen Zu- und Abwanderungen läßt sich die Größe von Kad nicht direkt feststellen. Da aber die Knoten ihre Hashwerte in Kad zufällig bestimmen, ist es möglich die Gesamtgröße über die mittlere Größe von Subnetzen abzuschätzen. Mit diesem Verfahren kommen die Autoren auf eine geschätzte Größe von ca. 980 000 Knoten im Kad Netz. Die erwartete Anzahl Hops liegt damit bei: $\text{steps}_{\text{avg}} = 2.7$ (perfekte Routingtabellen vorausgesetzt). Perfekte Routingtabellen werden in der Realität aber nicht vorkommen, weswegen im nächsten Schritt die Qualität der Routingtabellen untersucht wird.

3.5 Qualität von Routingtabellen

Die Qualität einer Routingtabelle wird durch zwei Indikatoren bestimmt:

- **Vollständigkeit**: wird durch das Verhältnis von IST- zu SOLL-Einträgen bestimmt. SOLL-Einträge ergeben sich aus: $\min(k, \frac{n}{2^x})$, wobei x die Bitanzahl der Subnetzmaske angibt.
- **Aktualität**: wird durch den Anteil aktiver Knoten in einem k-Bucket bestimmt.

Mit dem Tool k-Fetch wurden die Routingtabellen von 80 000 Kad Knoten extrahiert und analysiert. Die Auswertung hat ergeben, dass im Durchschnitt 1.5 SOLL-Plätze in der Routingtabelle leer sind und 1 Eintrag nicht aktuell ist. Folgende Abbildung veranschaulicht das Ergebnis:



(a) Mean number of contacts in a routing table bucket

Damit verringert sich die Anzahl effektiv nutzbarer Plätze auf $k = 10 - 1.5 - 1 = 7.5$. Womit sich ein neuer durchschnittlicher Hopwert ergibt:

$$\mathbf{steps_{avg}} = \mathbf{1} + \frac{\log_{\mathbf{2}}(\mathbf{n}) - \mathbf{t}(\mathbf{4}, \mathbf{1}, \mathbf{7.5})}{\mathbf{t}(\mathbf{3.25}, \mathbf{1}, \mathbf{7.5})} = \mathbf{2.91}$$

3.6 Lookup-Strategien

Parallelität verkürzt die Dauer von Lookups, da die Suche weitergehen kann, während auf das Timeout eines inaktiven Knotens gewartet wird. Im Folgenden geht es darum, die verschiedenen Parameter zu vergleichen.

3.6.1 Freie Parameter

1. Der **Parallelisierungsgrad α** setzt den parallel laufenden Anfragen eine Schranke.

2. Wie strikt diese Schranke ist, bestimmen die Parallelisierungsstrategien:

   Strikte Parallelität: „A new request is issued only when a pending request times out or a response is received." [**?**]

   Lockere Parallelität: „[...] a client can issue a lookup request to a contact that is among the top α contacts as soon as such a contact is identified [...]" [**?**]

3.6.2 Abhängige Parameter

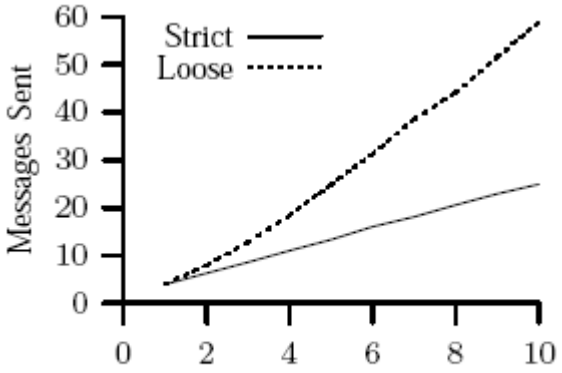
Hopzahl misst die Anzahl der Schritte, um von einem Quell- zu einem Zielknoten zu gelangen.

Latenzzeit misst die Zeit zwischen dem Start der Suchanfrage und einer positiven Antwort.

Overhead misst die Anzahl gesendeter Nachrichten.

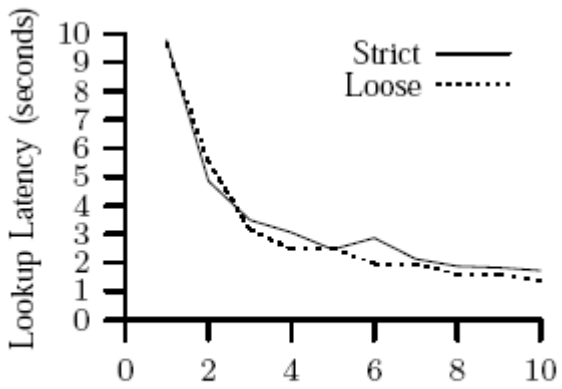
3.6.3 Ergebnisse

1. Der Overhead nimmt bei lockerer Parallelität stärker zu als bei strikter:



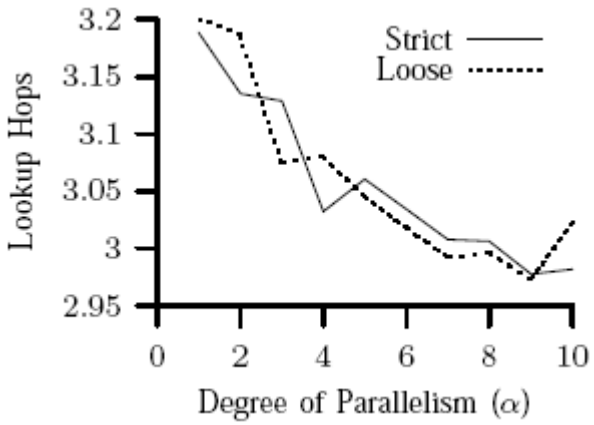
(b) Mean packets sent as a function of α

2. Lockere und strikte Parallelität verhalten sich bezüglich der Latenzzeit gleich:



(a) Mean lookup latency as a function of α

3. Auch die Hopzahl ist zwischen lockerer und strikter Parallelität nahezu gleich:



Nebenbei kann damit die obige Vorhersage empirisch bestätigt werden: Bei $\alpha = 1$ liegt die mittlere Hopzahl bei 3.2, was nahe an der vorrausgesagten Hopzahl von 2.91 liegt. Bei $\alpha > 1$ sinkt die Differenz noch weiter.

4. Da sich der Overhead bei lockerer Parallelität weder in kürzerer Latenzzeit noch in weniger Hops auszahlt, lohnt sich diese Strategie nicht. Die effizienteste Lookup-Methode ist demnach strikte Parallelität mit $\alpha = 3$.

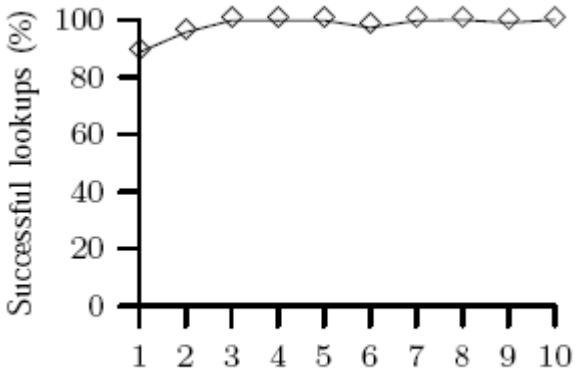
3.7 Replikationsstrategien

Um Datenverlust in einem DHT zu verhindern, müssen die Daten repliziert werden. Nur so kann das Netz mit einer Zuverlässigkeit von p garantieren, dass Daten die im Netz gespeichert wurden, auch gefunden werden können. Dazu sichert jeder Knoten seine Daten auf c benachbarten Knoten. Eine beliebige Replikation der Daten ist aus Effizienzgründen nicht sinnvoll. Daher sollen minimale Werte für c gefunden werden, die eine Zuverlässigkeit p garantieren.

Zu verschiedenen Werten $c \in [1, 10]$ wurde die Zuverlässigkeit p gemessen. Die Messung erfolgte in zwei Schritten:

1. Ein Wert wurde in c benachbarten Knoten gespeichert.
2. Gleich darauf wurde aus 50 zufälligen Knoten nach diesem Wert gesucht.

Im folgenden Schaubild ist die Zuverlässigkeit gegen den Replikationsfaktor abgetragen:



(a) Lookup consistency as a function of c

Daraus ist bereits ersichtlich, dass mit $c = 3$ bereits eine Zuverlässigkeit von nahezu 100 % erreicht wird - genauer von über 99.9 %.

4 Zusammenfassung

4.1 Ergebnisse

- k-Buckets bringen Redundanz UND Performanz
- k-Buckets UND komplexe Routingtabellen bringen keinen Vorteil
- Hopzahl: SOLL = 2.91 vs. IST = 3.2 (bei $\alpha = 1$) vs. anderen Autoren = 6.3
- effizienteste Lookup-Strategie: strikte Parallelität mit $\alpha = 3$
- p = 99,9 % wird erreicht mit: c = 3

4.2 eMule

- Lookup Strategie: lockere Parallelität mit $\alpha = 3$
- Routing Strategie: k-Buckets und Split Symbols
- Replikationsfaktor: c = 19 (im Durchschnitt)

Literatur