Technische Universität Kaiserslautern
FB Informatik
Lehrgebiet Datenverwaltungssysteme
AG Datenbanken und Informationssysteme
(Prof. Dr. Dr. h.c. Härder)
AG Heterogene Informationssysteme
(Prof. Dr. Deßloch)

Integriertes Seminar Datenbanken und Informationssysteme

Thema Nr. 4

Peer-To-Peer Computing

Betreuer: Jernej Kovse

Björn Jung

Bachwiese 18, 55494 Dichtelbach
Angewandte Informatik
MatNr.: 346148
b_jun@informatik.uni-kl.de

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

In jüngster Vergangenheit sind viele Peer-to-Peer (P2P) Systeme bekannt geworden, hauptsächlich in Bezug auf Datei-Austausch-Systeme. Beispiele hierfür sind Napster oder Gnutella. Diese Systeme sind vergleichsweise einfach, da sie oft nur einen Zugriff auf die Daten über den Dateinamen ermöglichen. Das Seminar beschäftigt sich nun damit, welche Netzwerkstrukturen solchen Systemen zugrunde liegen und zeigt Ansätze, wie komplexere Anfragen über beliebige Objekte darauf ausgeführt werden können, so dass ein P2P-Netz als Grundlage für eine verteile Datenbankanwendung nutzbar ist.

In Kapitel 2 wird eine Einführung in P2P Systeme gegeben und eine Abgrenzung gegenüber Client- / Server Systemen. Das Kapitel 3 zeigt wie P2P-Netze aufgebaut sein können, unstrukturiert oder strukturiert, und welche Auswirkung dies auf Suchanfragen innerhalb der Netze hat. In Kapitel 4 werden die zwei bekanntesten Systeme, Chord und CAN, zu strukturierten P2P-Netzen vorgestellt. Zu CAN werden weiterhin Verbesserungen gezeigt, welche die Anfragedauer optimieren. In Kapitel 5 werden zwei Frameworks behandelt, die sich mit der Behandlung komplexerer Anfragen, wie sie z.B. mit SQL möglich sind, beschäftigen.

2 Klassifikation von Netzstrukturen

Die Client-/Server Struktur gehört nicht zu den Peer-to-Peer Systemen. Sie sind hier aufgelistet, um den Unterschied, vor allem in Hinblick auf die Vor- und Nachteile zu verdeutlichen.

Schollmeier definiert Peer-to-Peer-Systeme wie folgt: "Eine verteile Netzwerk Architektur kann Peer-to-Peer Netz genannt werden, wenn die Teilnehmer einen Teil ihrer Ressourcen[1] zur Verfügung stellen. Die Ressourcen können von anderen Teilnehmern direkt genutzt werden, ohne dazwischengeschaltete Instanzen. Jeder Teilnehmer ist sowohl Anbieter der eigenen Ressourcen, als auch Anforderer anderer Ressourcen." (nach [Scho01])

2.1 Zentralisiert - Client/ Server Netze

Diese Struktur zeichnet sich dadurch aus, dass zwischen Server und Clients unterschieden wird. Es gibt nur einen Server, welcher viele Clients bedient. Der Server dient als zentrale Kontrollinstanz und bietet Dienste wie Datenabfrage oder Anfrageausführung an. Ein Client fragt nur Inhalte oder bestimmte Dienste an, stellt aber selbst keinerlei Ressourcen zur Verfügung. Wie Abbildung 1 zeigt, findet Kommunikation ausschließlich zwischen Client und Server statt.

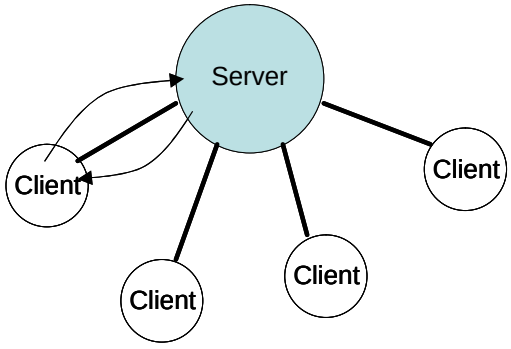


Abb. 1. Client- / Server Struktur

Der Vorteil der Struktur ist, dass diese Systeme einfach zu erstellen und einfach zu warten sind. Ein Nachteil ist, dass der Server einen *Single Point of Failure* bildet - wenn der Server ausfällt, ist das gesamte Netz nicht benutzbar. Des Weiteren vermag der Server nur eine bestimmte Anzahl an Clients zu bedienen. Das System ist schlecht skalierbar: Sollten mehr Clients als geplant im Netz sein, so muss der Server erweitert werden.

[1] Der Begriff Ressourcen ist allgemein gefasst. Dies kann bereitgestellter Speicherplatz sein; Objekte, auf die zugegriffen werden können oder zur Verfügung gestellte Rechenleistung. Im Weiteren werden unter Ressourcen bereitgestellte Objekte verstanden.

2.2 Semi-Dezentralisiert - Hybride P2P-Systeme

Bei hybriden P2P-Systemen gibt es immer noch eine Unterscheidung zwischen Server und Clients. Allerdings dient der Server nur noch als Kontrollinstanz, nicht mehr als Datenspeicher. Die Clients bieten nun selbst Ressourcen an und können miteinander kommunizieren. Alle Knoten bilden zusammen einen gemeinsamen Datenspeicher

Ein typischer Vorgang sieht so aus, dass ein Client eine Anfrage an den Server richtet (linker grau hinterlegter Client in Abbildung 2), dieser ermittelt einen Ziel-Client, welcher die Anfrage bedienen kann (rechter Grau hinterlegter Client), woraufhin die beiden Clients nun direkt miteinander kommunizieren (gestrichelte Pfeile).

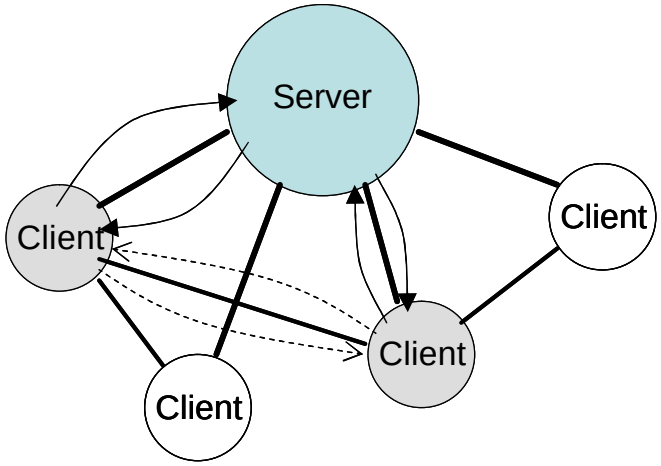


Abb. 2. Struktur hybrider P2P-Systeme

Die Vorteile dieser Struktur sind zum einen die leichte Realisierbarkeit durch den zentralen Server, zum anderen wird der Server im Gegensatz zu reinen Client- / Server Strukturen stark entlastet und kann somit eine weitaus höhere Anzahl an Clients bedienen.

In dieser Struktur ist immer noch der Server der kritische Punkt. Wenn dieser ausfällt, können die Clients keine neuen Kommunikationspartner finden. Auch hier ist das Netz, trotz Server-Entlastung, nicht beliebig erweiterbar.

2.3 Dezentralisiert - Reine P2P Systeme

In reinen P2P Systemen besteht keine Unterscheidung mehr zwischen Server und Clients. Schollmeier ([Scho01]) führt hier den Begriff "Servent" ein, abgeleitet von *Serv*-er und Cli-*ent*.

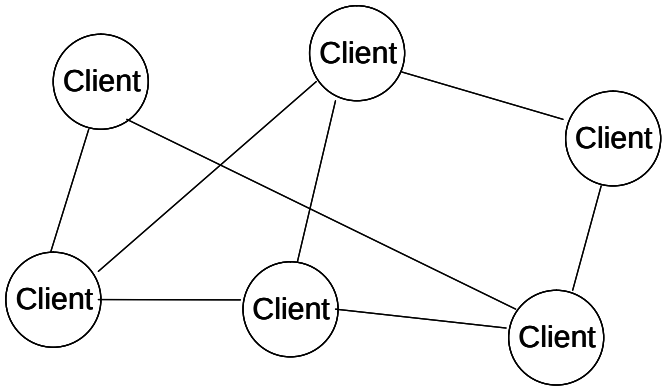


Abb. 3. Struktur reiner P2P-Systeme

Es gibt keinen Knoten mehr, der die zentrale Kontrolle übernimmt, genauso wenig eine zentrale Datenbank. Anfrageverarbeitung und Datenspeicher werden dezentral von den einzelnen Knoten übernommen. Jeder Knoten hat eine bestimmte Anzahl Verbindungen zu anderen Knoten, sog. Nachbarknoten. Daraus resultiert, dass kein Knoten eine globale Sicht über das System hat, sondern nur einen kleinen Teil des Netzes einsehen kann. In Abbildung 3 ist jeder Client ein solcher Knoten mit einer variierenden Anzahl an Verbindungen zu Nachbar-Clients.

Vorteile der P2P-Struktur sind die leichte Skalierbarkeit, da es keinen Zentralen Server mehr gibt. Des Weiteren können Peers einfach angefügt oder entfernt werden, ohne die Leistung des Netzwerkes zu beeinträchtigen. Dies ist notwendig für eine dynamische Umgebung, da andauernd bestehende Verbindungen wegfallen, sobald ein Knoten das Netz verlässt, und neue Verbindungen aufgebaut werden müssen, wenn ein Knoten hinzukommt.

Der Nachteil dieser Struktur im Gegensatz zu den anderen beiden Systemen ist das Einstiegsproblem: Da es keinen zentralen Server gibt, hat ein Knoten der neu in das Netz hinzukommt noch keine Nachbarknoten und muss diese erst finden.

3 Ressource Discovery

Da die Daten in P2P Systemen dezentralisiert auf allen Knoten verteilt vorliegen, bedarf es Suchalgorithmen, um bei einer Suche nach einem bestimmten Objekt diejenigen Knoten zu bestimmen, welche das Objekt liefern können. Es gibt zwei verschiedene Arten von P2P-Topologien, die unstrukturierten Systeme und die strukturierten Systeme, welche die Suche innerhalb des Netzes auf unterschiedliche Weisen realisieren.

3.1 Unstrukturierte Systeme

Bei unstrukturierten Systemen sind die Daten eines Knotens nur diesem Knoten selbst bekannt. Der intuitive Ansatz der Suche ist das Fluten des Netzes: Eine Anfrage an einem Knoten wird von diesem an alle benachbarten Knoten geleitet (was auch Broadcast genannt wird). Diese leiten die Anfrage wiederum an alle ihre Nachbarn weiter, bis schließlich jeder Knoten im Netz mindestens einmal erreicht wird. Durch redundante Pfade wird jedoch eine sehr hohe Netzwerkauslastung erzeugt. Wenn eine Anfrage über 7 Knoten verfolgt wird (7 *hops*), und jeder Knoten 4 Verbindungen C zu seinen Nachbarknoten besitzt (über eine Verbindung kommt die Anfrage, über 3 wird sie weitergereicht), so kann die Menge der entstehenden Nachrichten wie folgt berechnet werden (nach [AbHa02][2]):

$$Nachrichten = 2 * \sum_{i=0}^{hops} C * (C-1)^i \qquad (1)$$

In diesem Fall wären das 26240 erzeugte Nachrichten.

Da bei vielen Clients das Netz schnell überlastet werden kann, gibt es verschiedene Ansätze, das Datenaufkommen zu reduzieren.

- Jede Anfrage bekommt einen TTL (Time-To-Live) - Wert. Dieser Wert wird nach jedem hop um eins verringert, wenn der Wert Null erreicht, wird die Anfrage verworfen. Dies hat jedoch den gravierenden Nachteil, dass potentiell im Netz vorhandene Daten unter Umständen nicht gefunden werden können, da der minimale Pfad zwischen Anfrageknoten und Zielknoten größer sein kann, als der angegebene TTL-Wert.

- Schrittweise Erhöhung der TTL: Eine Anfrage bekommt zunächst einen sehr kleinen TTL-Wert, so dass die Suche auf die unmittelbare Nachbarschaft beschränkt wird. Bleibt die Suche ergebnislos, so wird der Wert um eins erhöht, und die Anfrage erneut gestellt, solange, bis ein Ergebnis gefunden wird, oder eine gegebene Höchstgrenze für den TTL-Wert erreicht wird. Wenn Ergebnisse in der Nachbarschaft eines Knotens gefunden werden können, so bleibt die Netzauslastung sehr gering. Allerdings kann es auch hier vorkommen, dass im Netz vorhandene Daten aufgrund der TTL Beschränkung nicht gefunden werden können.

- Neustrukturierung der Nachbarschaft: Unter der Annahme, dass Knoten, die in der Vergangenheit gute Ergebnisse geliefert haben, dieses auch in Zukunft machen werden, wird zu diesen Knoten eine direkte Verbindung aufgebaut. Dies hat den Vorteil, dass gute Ergebnisse schnell gefunden werden. Auch können sich Cluster bilden, deren Knoten einen hohen Verbindungsgrad untereinander haben und ein gemeinsames Interessensgebiet teilen, was sowohl die Art der Anfragen betrifft, als auch die vorhandenen Datensätze. Allerdings können Knoten, die allen gute Ergebnisse liefern, schnell

[2] Die Gleichung in [AbHa02] bezieht sich auf das Gnutella Protokoll. Sie ist aber auch beim hier gezeigten Ansatz gültig.

überlastet werden.

– Unicast: Statt eine Anfrage an alle benachbarten Knoten zu senden, wird diese nur an einen Knoten geschickt. Dieser kann zufällig gewählt werden oder mittels eines Ratings, das angibt, bei welchem Knoten ein Ergebnis am wahrscheinlichsten ist (Dieses Beruht meist auf der Historie des Knotens: Derjenige Knoten, der in der Vergangenheit gute Resultate geliefert hat, wird bei neuen Anfragen zuerst angesprochen). Auch hier bekommt die Anfrage einen TTL-Wert, jedoch ist dieser bedeutend höher als bei Broadcast. Bei einer Anfrage sind nur sehr wenige Knoten beteiligt, jedoch können potentiell vorhandene Daten wieder nicht gefunden werden. Auch muss der Weg einer Anfrage protokolliert werden, da sonst wenige Knoten, welche sich gegenseitig bevorzugen, sich diese Anfrage immer wieder gegenseitig zuschicken, bis der TTL-Wert erreicht, und die Anfrage verworfen wird.

3.2 Strukturierte Systeme

In strukturierten P2P-Systemen wird jeder Speicherort eines Objekts in einer dezentral verteilten Hashtabelle (Distributed Hash Table, DHT) hinterlegt. Die Informationen über die Daten eines Knotens werden nicht mehr wie im unstrukturierten Ansatz von diesem Knoten erfragt, sondern werden aus der Hashtabelle ausgelesen. Dies bedeutet auch, dass die Informationen über die Daten eines Knotens nicht unbedingt auf diesem Knoten gespeichert werden.

Jeder Knoten innerhalb des Netzes ist für einen Teil der DHT zuständig. Wenn eine Anfrage an einem Knoten vorliegt, so kann dieser genau bestimmen, zu welchem Knoten er die Anfrage weiterleiten muss, so dass sie ohne Umwege am für den entsprechenden Hash-Bereich zuständigen Knoten ankommt. Die Netzauslastung ist dementsprechend gering. Außerdem ist es immer möglich, Daten zu finden, sofern die im Netz vorhanden sind.

Das DHT Interface bietet drei Funktionen:

– `put(key, value):` Mit put können neue Einträge in die DHT hinzugefügt werden. Key ist der Hashschlüssel für das zu speichernde Objekt, value ist das Objekt selbst, oder eine Referenz auf den Knoten, der es enthält. Mittels einer Hash-Funktion wird von dem Objektbezeichner ein Schlüssel (key) erstellt und dann an entsprechender Stelle der Wert (value) eingetragen.
– `delete(key, value):` Mit delete können bestehende Einträge wieder aus der Tabelle gelöscht werden, z.B. wenn der entsprechende Knoten das Netz verlässt.
– `lookup(key)` → `value:` Mittels lookup kann nach einem Objekt gesucht werden. Dazu wird mittels der Hash-Funktion von der Objektbezeichnung der entsprechende key errechnet. An der entsprechenden Stelle der DHT kann nun das Objekt oder eine Referenz auf einen Knoten mit dem entsprechenden Objekt ausgelesen werden.

Für die Hash-Funktion kann entweder die Bezeichnung eines Objekts benutzt werden, wenn der Inhalt unbekannt ist, oder bestimmte Teilen des Inhalts, sofern diese bekannt sind. Dies muss in der Hash-Tabelle einheitlich geregelt sein. Des Weiteren bietet DHT nur Suche nach exakter Übereinstimmung, da ähnliche Objektbezeichnungen total unterschiedliche Hashschlüssel haben können. Aus diesem Grund eignen sich DHT Systeme zwar gut für die Suche nach z.B. Dateinamen, komplexere Anfragen mit Operatoren (z.B. SQL Anfragen) lassen sich mit DHT aber nur schwer umsetzen. [HHH+02]

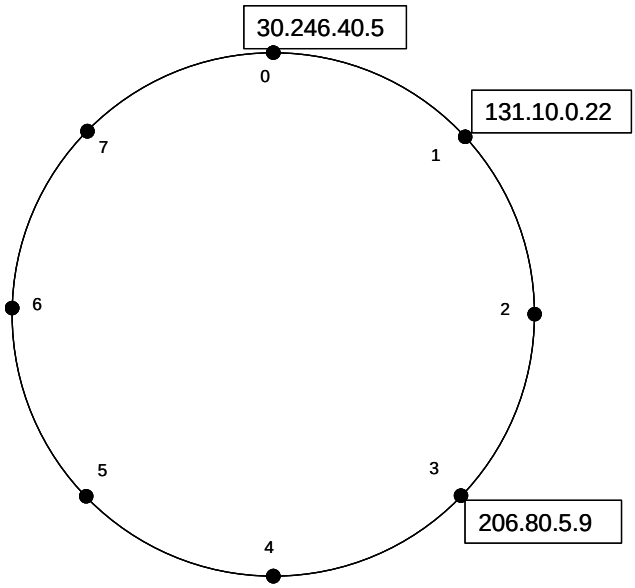
4 Strukturierte P2P-Architekturen

Die Architekturen Chord [SMD+01] und CAN [Ratn02] sind die beiden Ansätze, die die größte Verbreitung unter den strukturierten Architekturen haben. Beide sind reine Routing - Architekturen, die von Programmen zur Datenanfrage genutzt werden können. Eine Datenverarbeitung findet nicht statt.

4.1 Chord

Chord benutzt als Hashtabelle einen Identifier-Ring modulo 2^m. Dazu werden mittels der Hash-Funktion sowohl Objektbezeichner (key) als auch IP der Knoten (value) in denselben m-Bit Raum abgebildet. Der Wert von m muss so groß gewählt werden, dass die Wahrscheinlichkeit, dass mehrere Knoten oder mehrere gehashte Schlüssel denselben Identifier haben ausreichend gering ist, damit dieser Fall in der Praxis nicht auftaucht. Basisoperationen von Chord sind insert(key, value), delete(key, value) und lookup(key) → value.

Um innerhalb des Chord-Rings suchen zu können, wird jedem gehashten Schlüssel eines Objektbezeichners der entsprechende Nachfolgeknoten successor(key) im Ring zugeordnet. Dieser kennt dann die Zuordnung key → value (Schlüssel → Objekt oder Referenz auf das Objekt). Abbildung 4 verdeutlicht dies. Es liegt ein 3-Bit Ring vor, mit 3 Knoten. Knoten 131.10.0.22 hat selbst den key 1 und ist zuständig für den key 1, Knoten 206.80.5.9 hat selbst den key 3 zuständig für key 2 und 3, Knoten 30.246.40.5 hat selbst den key 0 und ist zuständig für die keys 4,5,6,7 und 0. Anfragen können nun im Uhrzeigersinn im Ring weitergereicht werden, bis der zuständige Knoten erreicht wird.



Knoten 131.10.0.22

Eintrag	succ	Knoten (IP)
1	2	206.80.5.9
2	3	206.80.5.9
3	5	30.246.40.5

Knoten 206.80.5.9

Eintrag	succ	Knoten
1	4	30.246.40.5
2	5	30.246.40.5
3	7	30.246.40.5

Knoten 30.246.40.5

Eintrag	succ	Knoten
1	1	131.10.0.22
2	2	206.80.5.9
3	4	30.246.40.5

Abb. 4. 3-Bit Chord Identifier-Ring und die entsprechenden Finger-Tabellen

Da bei einem Ring mit N Knoten der Suchpfad auf diese Weise O(N) beträgt, hat jeder Knoten in Chord zusätzliche Informationen über andere Knoten. Diese liegt in Form einer *Finger-Tabelle* mit m Einträgen vor (bei einem m-Bit Ring). Dabei enthält der i-te Eintrag dieser Tabelle des Knotens n die Adresse des ersten Knotens s, der n in mindestens 2^{i-1} Schritten folgt. Knoten s = successor($n+2^{i-1}$) (mit $1 \leq i \leq m$, modulo 2^m). Der erste Eintrag der Finger-Tabelle ist gleichbedeutend mit dem direkten Nachfolger (successor). Die Abstände zwischen den referenzierten Knoten nehmen exponentiell zu.

Die Finger-Tabelle des Knoten 3 aus Abbildung 4 berechnet sich wie folgt: Der erste Eintrag verweist auf den Nachfolger des keys $(0+2^0) = 1$, der zweite auf den Nachfolger des keys $(0+2^1) = 2$ und der dritte Eintrag verweist auf den Nachfolger des keys $(0+2^2) = 4$. Successor(1) ist Knoten 131.10.0.22, successor(2) ist Knoten 206.80.5.9 und successor(4) ist Knoten 30.246.40.5.

Es ergeben sich zwei Eigenschaften durch diese Finger-Tabellen: Jeder Knoten kennt nur eine geringe Anzahl von weiteren Knoten (genauer log N, bei N maximal möglichen Knoten) und von diesen Knoten kennt er seine im Ring direkt folgende „Nachbarschaft" besser, als im Ring weit entfernte Knoten. Die zweite Eigenschaft ist, dass kein Knoten genug Informationen besitzt, um jeden Hash-key direkt lokalisieren zu können. Liegt nun an einem Knoten eine Anfrage für einen key k an, den dieser Knoten nicht direkt kennt, so sucht er sich aus seiner Finger-Tabelle einen Knoten, dessen Identifier möglichst nah an k liegt. Aus Eigenschaft eins folgt, dass dieser so gefundene Knoten mehr über k weiss, als er selbst. Dieser Suchweg wiederholt sich rekursiv, bis der zu k gehörige Knoten, der successor(k), gefunden ist.

Um das Hinzufügen neuer Knoten in das Chord-Netz zu vereinfachen, erhält jeder Knoten die Kenntnis über seinen direkten Vorgänger (predecessor). Wenn nun ein Knoten n neu in das Netz kommt, so werden 3 Schritte durchlaufen. Zuerst benötigt n die Kenntnis von einem anderen Knoten n' der bereits im Netz ist. Dieser kann ihm dann entsprechend dem Identifier-key von n dessen Finger-Tabelle und seinen Vorgänger mitteilen. Als nächstes müssen die Finger-Tabellen der bereits vorhandenen Knoten aktualisiert werden. Der neue Knoten n wird der i-te Eintrag in der Tabelle eines anderen Knotens p, wenn erstens p mindestens der 2^{i-1}-te Vorgänger von n ist und zweitens der i-te Finger von p ein Nachfolger von n ist. Dies trifft auf denjenigen Knoten zu, der der direkte Vorgänger von $n-2^{i-1}$ ist (Dafür die Kenntnis des predecessors). Der neue Knoten n muss also für genau m andere Knoten den Identifier-key ermitteln und diese von seinem Hinzukommen in Kenntnis setzen, was einer Komplexität von O(m log N) Suchanfragen entspricht. Im letzten Schritt muss der Nachfolger von n, der nun für einen kleineren Teil des Ringes zuständig ist als vorher, die entsprechenden (key, value)-Paare an n übergeben. Als praktische Optimierung kann ein neuer Knoten die Finger-Tabelle und den Vorgänger seines direkten

Nachbarn übernehmen. Diese Tabelle kann er als Hilfe benutzen, um seine eigene Finger-Tabelle zu erstellen, da beide Tabellen sehr ähnlich sind. Auf diese Weise lässt sich die Komplexität auf annähernd O(log N) reduzieren.

Wenn ein Knoten n das Netz verlässt, so werden diese Schritte in umgekehrter Reihenfolge vollzogen. Zuerst übergibt der Knoten seine (key, value) - Paare an seinen Nachfolger, danach müssen alle anderen Knoten von seinem Gehen informiert werden, bei denen n in der Finger-Tabelle auftaucht und mit dem Nachfolger von n ersetzt werden.

Die durchschnittliche Pfadlänge einer Suche ist O(log N) bei N Knoten im Netz. Wenn ein Knoten neu ins Netz kommt, oder das Netz verlässt, so müssen auch O(log N) Knoten benachrichtigt werden.

4.2 CAN

Als Hashraum benutzt CAN einen d-Dimensionalen Koordinatenraum. Jeder Knoten ist verantwortlich für eine Zone dieses Raumes, die von allen anderen Zonen disjunkt ist. In Abbildung 5 ist ein Beispiel-Koordinatenraum mit 5 Knoten. Verbindungen zwischen Knoten bestehen überall dort, wo 2 Zonen aneinandergrenzen. Knoten 2 hat z.B. Verbindungen zu den Knoten 1,3 und 4. Zwischen den Knoten diagonal liegender Zonen, z.B. Zone 1 und 3, besteht keine direkte Verbindung.

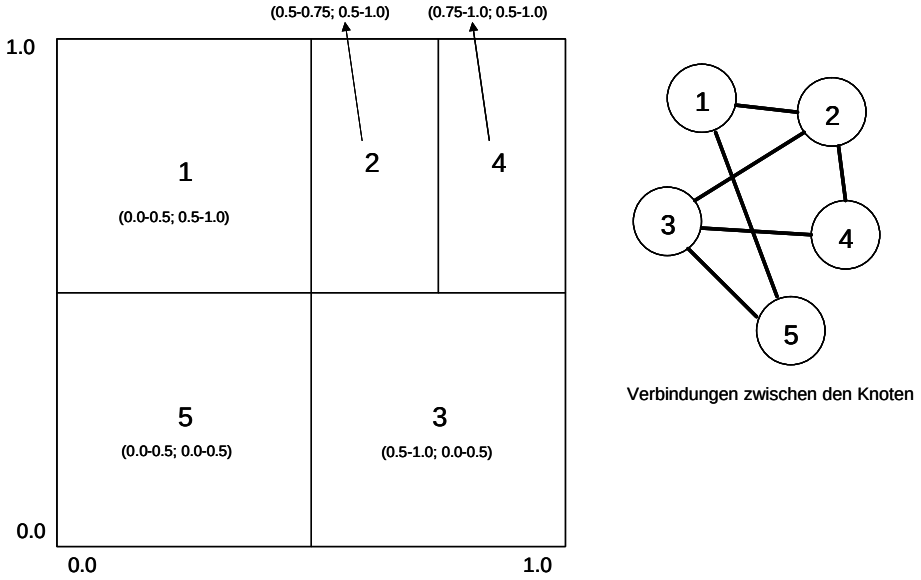


Abb. 5. 2-Dimensionaler Koordinatenraum mit 5 Knoten

Basisoperationen sind wieder `insert(key, value)`, `delete(key, value)` und `lookup(key)` → `value`. Die Hashfunktion ordnet jedem key einen Punkt p in diesem Koordinatenraum zu (h(key) p(x,y)). Derjenige Server, in dessen Zone p liegt, ist dann für die Verwaltung des entsprechenden (key, value)-Paares zuständig.

Wenn bei einem Knoten eine lookup-Anfrage erfolgt, so errechnet der Knoten zuerst den Koordinatepunkt p für den gesuchten key. Liegt dieser innerhalb der eigenen Zone, so kann die Anfrage direkt beantwortet werden. Liegt p außerhalb der eigenen Zone, so kann der Knoten über den Koordinatenraum einen direkten Nachbarn bestimmen, dessen Zone näher an p liegt, als seine eigene. Der so gefundene Nachbar wiederholt die Prozedur, bis der Knoten gefunden wird, der für die Zone zuständig ist, in der p liegt.

Wenn ein neuer Knoten n das CAN Netz hinzukommen will, so braucht er dazu auch die Hilfe eines ihm bekannten Knotens n' welcher schon im Netzwerk ist. Zuerst sucht sich n einen zufällig gewählten Punkt innerhalb des Koordinatenraums. Dann wird die Join-Anfrage mithilfe von n' an den Knoten geschickt, der für die Zone zuständig ist, in der der gewählte Punkt liegt. Dieser Knoten halbiert nun seine Zone, und gibt die Hälfte der Zone an den neuen Knoten ab. Danach werden die (key, value)-Paare an den neuen Knoten übertragen und die Verbindungen zwischen den Knoten der Nachbarzonen aktualisiert. Der neue Knoten baut eine Verbindung zu allen Nachbarknoten auf, und diese wiederum übernehmen den neuen Knoten in ihre Verbindung und lösen gegebenenfalls die Verbindung zum ehemaligen Knoten ihrer Nachbarzone.

Wenn ein Knoten das Netz verlässt, so muss ein anderer Knoten seine Zone übernehmen. Wenn Knoten y in Abbildung 6 Teil 1 das Netz verlässt, so kann Knoten x dessen Zone einfach durch vergrößern seiner eigenen Zone übernehmen. Dies ist der Fall, wenn y nach x in das Netz kam, und von x

seine Zone zugewiesen bekam. Wenn allerdings eine Situation wie in Abbildung 6 Teil 2 vorliegt, so kann x seine eigene Zone nicht einfach vergrößern, sondern muss beide Zonen getrennt voneinander verwalten. Falls irgendwann ein neuer Knoten eine Zone von x zugewiesen bekommen soll, so halbiert dann x seine Zone nicht, sondern gibt eine der beiden Zonen komplett ab.

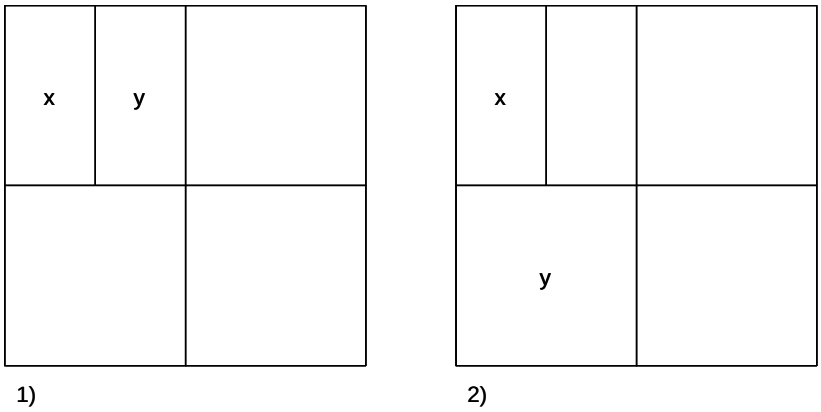


Abb. 6. Verlassen eines Knotens im CAN-Netz

Die durchschnittliche Pfadlänge einer Suchanfrage in einem d-dimensionalem CAN-Netz mit N Knoten beträgt $O(dN^{1/d})$, wobei jeder Knoten im Schnitt Informationen über 2d andere Knoten speichern muss. Für den Fall, dass d = log(N) ist, so hat CAN gleich hohe Suchkosten wie Chord.

4.3 Verbesserungen bei CAN

Zu CAN gibt es eine Reihe von Verbesserungen ([RFH+01]). Ziel dieser Verbesserungen ist es, die Anfragedauer (latency) der Suche zu minimieren. Wichtig ist jetzt die Unterscheidung zwischen hops auf Applikationsebene (CAN) und hops auf IP-Ebene. Die Anfragedauer auf IP-Ebene wird mit der *round-trip-time* (RTT) gemessen. Dies ist die Dauer, die ein Datenpaket das zwischen 2 Knoten direkt ausgetauscht wird für den Hin- und Rückweg benötigt. Ein hop auf Applikationsebene entspricht normalerweise mehreren hops auf IP-Ebene. Um die Anfragedauer zu verringern kann jetzt die Länge des Suchpfades (die Anzahl der hops auf Applikationsebene) verringert werden, oder die hops zwischen zwei Knoten auf IP-Ebene können optimiert werden. Zuerst wird gezeigt, wie sich die Länge des Suchpfades minimieren lässt; danach, wie die Anzahl der hops auf IP-Ebene optimiert werden kann.

Mehrdimensionale Koordinatenräume
Die Länge des Suchpfades hängt direkt ab von der Anzahl der Dimensionen d mit $O(dN^{1/d})$ (N ist die Anzahl der Knoten im Netz). Deswegen führt eine Erhöhung der Dimensionsanzahl direkt zu kürzeren Suchpfaden.

Mehrere Realitäten
Es werden mehrere unabhängige Koordinatenräume (Realitäten) eingeführt. Jeder Knoten bekommt in jeder Realität eine andere Zone zugewiesen. Wenn es r Realitäten gibt, ist jeder Knoten für r Zonen zuständig und besitzt r Mengen von Nachbarn. Der Inhalt des Hash-Raumes wird für jede Realität kopiert. Wenn ein bestimmtes Objekt z.B. den Punk (x,y) zugewiesen bekommt, so ist in jeder Realität ein anderer Knoten dafür verantwortlich.

Wenn nun eine Suchanfrage anliegt, so kann jeder Knoten diejenige Realität wählen um die Anfrage weiterzuleiten, bei der der verbleibende Pfad minimal ist.

Vergleich Mehrere Dimensionen gegenüber mehreren Realitäten
Wie Abbildung 7 zeigt, verkürzt sich die Pfadlänge durch mehrere Dimensionen schneller, als durch mehrere Realitäten.

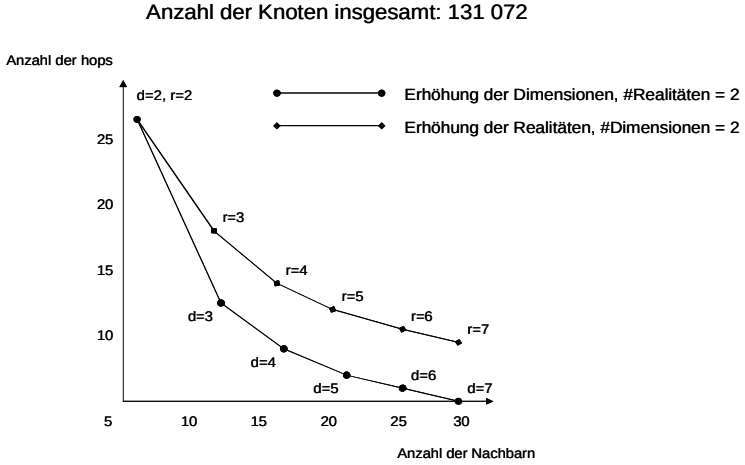


Abb. 7. Pfadlänge in Bezug auf die Anzahl der Nachbarn

Allerdings haben mehrere Realitäten einen positiven Nebeneffekt auf die Stabilität des Systems. Wenn ein Knoten ausfällt, ohne sich ordentlich aus dem CAN-Netz abzumelden, dann verschwindet mit ihm die Information über alle (key, value)-Paare seiner Zone. Da bei mehreren Realitäten die entsprechenden Einträge für jede Realität kopiert werden, hilft die daraus resultierende Redundanz solche Ausfälle besser zu kompensieren.

Verbesserung des Routing-Verfahrens
Während die beiden obigen Verbesserungen nur auf Applikationsebene liefen, wird hier versucht, die IP-Topologie bei der Berechnung des Suchpfades zu berücksichtigen. Wenn ein Knoten einen Nachbar auswählt, um die Anfrage weiter zu leiten, dann schickt er die Anfrage nicht mehr an denjenigen Knoten, der den Suchpfad direkt minimieren würde, sondern er erstellt eine Liste von allen Nachbarknoten, bei denen der die Länge des Suchpfads potentiell verringert wird. Aus dieser Liste wählt er dann denjenigen Knoten heraus, zu dem er die geringste RTT hat. Dies minimiert nicht unbedingt die Länge des Suchpfades, sondern verbessert die Zeitdauer der Anfrage. Eine höhere Anzahl an Dimensionen erhöht die Anzahl der Nachbarknoten, so dass jeder Knoten eine höhere Wahlmöglichkeit hat, zu welchen Nachbar er die Anfrage weiterschickt. Daraus resultiert, dass mit mehr Dimensionen sich die Anfragedauer verringert. Die Autoren sprechen von einer Verbesserung in einem Testsystem[3] von 24% bis 40%, abhängig von der Anzahl der Dimensionen.

Überladen der Zonen
Während bisher jeder Zone genau ein Knoten zugeordnet war, so können sich jetzt mehrere Knoten eine Zone teilen. Hier werden auch wieder die Inhalte dieser Zone für jeden Knoten kopiert. Es wird weiterhin der Parameter *maxpeers* eingeführt, welcher angibt, wie viele Knoten pro Zone erlaubt sind (der Wert ist eher klein, z.B. maxpeers = 3 oder 4). Wenn nun ein neuer Knoten in das CAN-Netz hinzukommt, so wird ihm keine eigene Zone zugewiesen, sondern muss sich eine Zone mit anderen Knoten teilen und der Wer für maxpeers wird erhöht. Erst wenn die Grenze für maxpeers erreicht wird, wird die Zone halbiert, und die Knoten der entsprechenden Zone werden auf die zwei neu entstandenen Zonen aufgeteilt.

Jeder Knoten muss nun zusätzlich eine Referenz auf die anderen Knoten speichern, die in derselben Zone liegen. Für die Nachbarzonen behält er weiterhin je nur eine Referenz, auch wenn für die Nachbarzone mehrere Knoten zuständig sind. Jeder Knoten misst in gewissen Zeitabständen die RTT zu allen in den Nachbarzonen liegenden Knoten. Dazu schickt er an die Nachbarknoten, die er in seiner Liste hat eine Nachricht, dass diese veranlassen, dass alle anderen Knoten, die auch in der entsprechenden Zone liegen, die RTT mit dem Ursprungsknoten messen sollen. Der ursprünglich anfragende Knoten bekommt so eine Liste mit RTT-Werten von allen Knoten einer Nachbarzone. Aus dieser Liste wählt er sich nun den Nachbarknoten mit der geringsten RTT heraus, und ersetzt die Referenz auf den alten Nachbarknoten durch den neu bestimmten.

Dies hat zwei Auswirkungen auf die Dauer der Suchanfragen: Dadurch, dass die Knoten ihre Nachbarn anhand der RTT-Werte bestimmen, wird die Anfragedauer auf IP-Ebene verringert. Da jetzt allerdings

[3] Ein simuliertes Netzwerk mit in Baumstruktur angeordneter IP-Hierarchie. Knoten auf der obersten Hierarchieebene haben eine Verzögerung von 100ms zueinander. Knoten der zweiten Ebene haben nur eine Verbindung zu ihren Väterknoten mit einer Verzögerung von 10ms. Knoten der dritten Ebene haben auch nur eine Verbindung zu ihren Väterknoten mit einer Verzögerung von 1ms. Die Ergebnisse sind gemittelt über die Anzahl der Knoten zwischen 2^8 und 2^{18}.

mehrere Knoten für dieselbe Zone verantwortlich sind, gibt es insgesamt weniger Zonen. Deshalb verringert sich auch die Pfadlänge auf Applikationsebene. Ein weiterer Nebeneffekt ist, dass durch die Redundanz die dabei entsteht auch wieder Knotenausfälle besser abgefangen werden können.

Bei 4 Knoten pro Zone kann im Testsystem eine durchschnittliche Verbesserung von 45% der Anfragedauer auf IP-Ebene erreicht werden.

Verwendung mehrerer Hash-Funktionen

Hauptsächlich auf erhöhte Datenverfügbarkeit durch Redundanz ausgerichtet, werden mehrere Hash-Funktionen benutzt, um einen Schlüssel auf mehrere Punkte abzubilden. Bei k Hash-Funktionen gibt es entsprechend k (key, value)-Kopien im System. Ein solches (key, value)-Paar ist erst dann nicht verfügbar, wenn alle Kopien gleichzeitig ausfallen.

Diese Kopien kann man ausnutzen, um schneller Suchanfragen beantworten zu können: Eine Anfrage nach einen (key, value)-Paar wird gleichzeitig an alle k Punkte geschickt. So kann der Anfragende Knoten sich der zuerst eintreffenden Antwort bedienen.

Topologie-sensitiver Aufbau des CAN-Netzwerks

Da bisher die die Knoten per Zufall den entsprechenden Zonen zugeteilt wurden, können die direkten Nachbarn eines Knotens weit entfernt liegen. So kann es z.B. vorkommen, das ein Knoten eine Suchanfrage stellt, die ein anderer Knoten, der in unmittelbarer Nachbarschaft zum fragenden Knoten liegt beantworten kann. Da die Nachbarschaft aber nicht berücksichtigt wird, ist es durchaus möglich, dass der Anfragepfad dorthin sehr lang ist, und auch über weit entfernt liegende Knoten führt.

Um jetzt solche geographischen Gegebenheiten berücksichtigen zu können wird ein neues Topologie-Modell eingeführt. Um den Standort eines Knotens zu bestimmen, werden feste Referenzpunkte hinzugezogen (z.B. die DNS-Root-Server), welche als Landmarken dienen. Ein Knoten bestimmt nun zu jedem dieser Landmarken seinen RTT-Wert und ordnet die Ergebnisse ansteigend. Bei m solcher Landmarken sind m! verschiedene Ordnungen möglich. Dementsprechend wird auch der Koordinatenraum in m! Partitionen aufgeteilt. Dies geschieht folgendermaßen: Die Dimensionen der Raumes werden zyklisch aneinandergereiht (Bsp.: Bei einem 3-dimensionalen Raum mit den Dimensionen x,y,z wird die zyklische Reihenfolge (xyzxyzxyz...) erstellt). Die erste Dimension dieser Reihenfolge wird dann in m Teile geteilt. Jedes dieser m Teile wird in der zweiten Dimension in m-1 Teile geteilt, und diese wiederum in der dritten Dimension in m-2 Teile. Für jede der m! Ordnungen entsteht so ein genau festgelegter Bereich des Koordinatenraumes. Wenn nun ein Knoten neu in das Netz hinzukommt, so bekommt er einen zufälligen Punkt innerhalb der Zone, die seiner Landmarken-Ordnung entspricht.

Ein Nachteil dieses Modells ist, dass jetzt die Gleichverteilung der Knoten im Koordinatenraum nicht mehr gegeben ist. Dies kann man kompensieren, wenn man eine Möglichkeit einführt, dass ein Knoten, der ausgelastet ist, Teile seiner Zone an andere Knoten abgeben kann. Ziel zukünftiger Forschungen ist es hier, ein Modell zu entwickeln, dass sowohl die Geographische Verteilung beachtet, als auch die Gleichverteilung im Koordinatenraum.

Ansatz zur Gleichverteilung

Dieser Ansatz sieht es vor, dass wenn ein Knoten neu in das CAN-Netz hinzukommt, der entsprechende Zielknoten nicht direkt seine Zone teilt und die eine Hälfte an den neuen Knoten n abgibt, sondern, dass der Zielknoten erst die Größe seiner Zone mit der Größe der Nachbarzonen vergleicht, und derjenige Knoten mit der größten Zone kann nun diese halbieren und die eine Hälfte dem neuen Knoten zuweisen. Unter der Vorraussetzung, dass die Hash-Funktion die (key, value)-Paar gleichmäßig im Koordinaten-raum abbildet, führt dies zu einer Gleichverteilung über die Anzahl der Paare, für die die Knoten zuständig sind.

Dies führt aber nur bedingt zu einer Entlastung der Knoten. Da es durchaus Vorkommen kann, dass ein bestimmtes (key, value)-Paar sehr oft angefragt wird, verursacht dies immer noch eine hohe Auslastung am entsprechenden Knoten (das sogenannte *hot-spot* Problem).

Caching und Replikation von (key, value)-Paaren

Um jetzt auch solche hot-spots zu entlasten, werden Caching- und Replikationstechniken eingeführt.

Das Caching sieht vor, dass jeder Knoten alle Anfrageergebnisse, die er bekommen hat in seinem *Cache* zwischenspeichert. Liegt nun eine neue Anfrage vor (von diesem Knoten selbst, oder nur weitergeleitet) so durchsucht er erst seinen Cache nach dem Ergebnis. Findet er es dort, so kann er die Anfrage direkt beantworten. Nur wenn das Ergebnis nicht lokal im Cache vorliegt, wird die Anfrage entsprechend dem Suchpfad weitergeleitet. (key,value)-Paare, die oft angefragt werden, sind auch entsprechend oft im Cache der einzelnen Knoten enthalten.

Die Replikation ist gegensätzlich zum Caching. Beim Caching geschieht das Zwischenspeichern eher passiv (sozusagen als Nebeneffekt einer Anfrage). Bei der Replikation veranlasst der überlastete Knoten (unabhängig von einer bestimmten Anfrage), dass er eine Kopie dieses Paares an alle seine Nachbarn verteilt. Diese können nun ihrerseits abhängig von ihrer Auslastung bestimmen, ob die die Kopie selbst speichern, oder wiederum an einen weiteren Nachbar weitereichen. Als Folge der Replikationen ist ein häufig angefragtes (key, value)-Paar oft in der Nachbarschaft des überlasteten Knotens vertreten, so dass weitere Anfragen nicht mehr den überlasteten Knoten erreichen.

Um die Konsistenz der Daten zu gewährleisten, erhalten sowohl gecachte als auch replizierte Paare einen time-to-live Wert, welcher veranlasst, dass diese Kopien nach einer bestimmten Zeit gelöscht werden.

Ergebnis der Verbesserungen

Um die Auswirkungen der Verbesserungen beurteilen zu können wurden zwei Testsysteme mit jeweils 2^{18} Knoten verglichen. Das erste Testsystem (*bare bones* genannt) besitzt 2 Dimensionen und keine der vorgeschlagenen Verbesserungen. Das zweite Testsystem (*knobs-on-full*) besitzt 10 Dimensionen und jede Zone ist mit bis zu 4 Knoten überladen. Des Weiteren enthält es zusätzlich das Verbesserte Routing-Verfahren, und eine verbesserte Gleichverteilung der Zonen. Die Ergebnisse sind in Abbildung 8 dargestellt. Dabei gibt die Pfadlänge die durchschnittliche Anzahl der benötigten hops zwischen zwei Punkten in Koordinatenraum auf Applikationsebene an. #Nachbarn gibt an, wie viele Nachbarn jeder Knoten im Schnitt hat. #Knoten/Zone gibt an, wie viele Knoten sich durchschnittliche eine Zone teilen. Die IP latency gibt die durchschnittliche Verzögerung auf IP-Ebene zwischen zwei zufälligen gewählten Knoten an. CAN path latency gibt die Dauer einer gesamten Suchanfrage an. Die Topologie dieser Testsysteme ist genauso, wie beim obig angegebenen Testsystem.

	„bare bones“ CAN	„knobs-on-full“ CAN
Pfadlänge	198,0	5,0
#Nachbarn	4,75	27,1
#Knoten/Zone	1	2,95
IP latency	115,9ms	82,4ms
CAN path latency	23008ms	135,29ms

Abb. 8. Ergebnisse der Verbesserungen

Die Dauer der Suchanfrage ist sehr stark zurückgegangen. Dies liegt hauptsächlich an der Erhöhung der Dimensionen und daraus resultierend eine starke Verkürzung des Suchpfades. Dadurch, dass für jede Zone mehrere Knoten zuständig sind, gibt es insgesamt weniger Zonen, was wiederum zu einem kleineren Suchpfad führt. Durch die hohe Anzahl der Nachbarn eines Knotens (27,1 + 2,95) wird die Verzögerung auf IP-Ebene geringer. Ohne eine Heuristik, welche als nächsten Knoten denjenigen mit minimalen RTT-Wert wählt wäre die Dauer der Suchanfrage im Bereich $5 \cdot 82{,}4 = 412$. Mit dieser Heuristik kann die Anfragedauer weiter auf 135,29ms verringert werden. Der unterschiedliche Wert der IP latency erklärt sich dadurch, als dass beim knobs-of-full CAN jeder Knoten zwischen mehreren Nachbarn wählen kann, die für die gleiche Zone zuständig sind.

5 Komplexe Anfragen in strukturierten Systemen

Da DHT basierte Systeme sehr effizient in der Suche nach Objekten sind, liegt es nahe, diese auch für Datenbanken zu nutzen. Da wie oben geschrieben, DHTs nur die Suche nach exakter Übereinstimmung unterstützen, bedarf es einiger Erweiterungen des Protokolls, um auch komplexere Anfragen ausführen zu können. Hier werden zwei Ansätze vorgestellt, um auch solche Anfragen beantworten zu können.

5.1 Näherungslösung für Anfragen über Wertebereiche

Gupta, Agrawal und Abbadi stellen in [GuAA03] ein Framework vor. Dieses ermöglicht Anfragen über Wertebereiche in einem strukturierten System. Grundlegender Gedanke ist hier, dass der Nutzer eine

breite Anfrage an das System stellt, aber nicht die exakte Antwort seiner Anfrage benötigt. Aus diesem Grund wurde eine Technik entwickelt, die Näherungslösungen liefert.

Das System basiert auf Chord, bedarf jedoch zweier Erweiterungen. Die erste Erweiterung ist, die Hash-Funktion einen einzigen Schlüssel für einen Bereich (low, high) erzeugt. Die zweite Erweiterung ist, dass Ergebnisse auf den Knoten zwischengespeichert werden (Caching). Ein Knoten, der ein Ergebnis für Bereich (low, high) zwischenspeichert, ist mit hoher Wahrscheinlichkeit auch für den Bereich (low-ε, high+ε) zuständig. Die Idee ist also, dass ähnliche Bereiche mit hoher Wahrscheinlichkeit auf demselben Knoten zwischengespeichert werden.

Um dies zu gewährleisten, muss eine bestimmte Familie H der Hash-Funktion benutzt werden, das so genannte *Locality Sensitive Hashing*. Wenn zwei Mengen A und B mit der Domäne D gegeben sind, und $h \in H$, dann ist h lokalitätserhaltend, wenn

$$\text{Wahrscheinlichkeit } [h(A) = h(B)] = \text{sim}(A, B). \quad (2)$$

Dabei ist sim(A, B) ein Maß für die Ähnlichkeit der Mengen A und B. Wenn Q der Wertebereich der Anfrage ist, und R der enthaltene Bereich der gegebenen Antwort, so wird das Maß der Ähnlichkeit definiert durch

$$sim(Q, R) = \frac{|Q \cap R|}{|Q \cup R|} \quad (3)$$

Um weiterhin sicherzustellen, dass ähnliche Bereiche auf dem selben Knoten gespeichert werden, und andersartige Bereiche auch auf unterschiedlichen Knoten, wird folgende Methode angewandt: Aus den Eigenschaften der Hash-Funktion folgt, dass zwei Mengen Q und R mit der Wahrscheinlichkeit p = sim(Q, R) den selben Bezeichner zugeordnet bekommen. Sei $g = \{h_1, h_2, .., h_k\}$ eine Gruppe von k Hash-Funktionen zufällig aus der Familie der lokalitätserhaltenden Hash-Funktionen gewählt. Dann ist die Wahrscheinlichkeit, dass zwei Mengen Q und R mit allen k Funktionen auf denselben Schlüssel abgebildet werden gleich Wahrscheinlichkeit $[g(Q) = g(R)] = p^k$. Wenn nun *l* solcher Gruppen $g_1, g_2, .. g_l$ von Hash-Funktionen genommen werden, dann ist die Wahrscheinlichkeit, dass Q und R in Gruppe g_i nicht übereinstimmen gleich $1 - p^k$ und die Wahrscheinlichkeit, dass sie in allen Gruppen nicht übereinstimmen $(1 - p^k)^l$. Daraus folgt, dass die Wahrscheinlichkeit, dass Q und R in wenigstens einer dieser Gruppen übereinstimmen $1 - (1 - p^k)^l$. Wenn nun *l* Gruppen mit jeweils k Hash-Funktionen genommen werden, so ergeben sich für einen Bereich *l* verschiedene Schlüssel, so dass jedes Tupel *l* mal in den Hash-Raum eingefügt werden kann. Anhängig von k und *l* ist die Wahrscheinlichkeit, dass wenigstens ein Knoten (von den insgesamt *l* möglichen Knoten) eine Übereinstimmung der Bereiche liefert sehr hoch.

Dazu wird folgender Algorithmus auf dem Knoten von dem die Suche ausgeht, verwendet:

```
INPUT Q: Der gesuchte Bereich (low, high)
OUTPUT: Die IP vom Knoten mit der besten Übereinstimmung
BEGIN
for each g[l] do
    // g[l] ist eine Gruppe von Hash-Funktionen
    identifier[l] = 0
    for each h[i] in g[l] do
    // h[i] ist eine Hash-Funktion der Gruppe g[l]
        identifier[l] ^= h[i](Q)
    done
done
for each identifier[l] do
    sende eine Anfrage an den Knoten, der für identifier[i]
    zuständig ist. //Dieser liefert einen Wert für seine
                   //Übereinstimmung
done
Wähle aus allen Antworten diejenige mit der größten Übereinstimmung
Wenn keine Übereinstimmung genau genug ist, speichere das selbst
berechnete Ergebnis auf jedem Knoten identifier[l]
```

Dabei wird für jede Gruppe ein Identifier errechnet, indem alle Ergebnisse der Hashfunktionen einer Gruppe mit XOR verknüpft werden. Dies dient dazu, die anfangs gegebene Anforderung, ähnliche Bereiche auf demselben Knoten zu speichern, und entsprechend andersartige Bereiche auf verschiedenen Knoten.

Der Such-Algorithmus sieht nun vor, dass entsprechend der Hash-Funktionen aller Gruppen die Anfrage an alle *l* Knoten weitergereicht wird. Jeder Knoten kann dann den Wert der Übereinstimmung

errechnen und an den anfragenden Knoten zurückschicken. Der anfragende Knoten kann sich nun die beste Übereinstimmung auswählen und die Anfrage dann von diesem Knoten beantworten lassen. Die Anfragen an die Knoten können parallel ausgeführt werden. Auf diese Weise werden für jede Anfrage O(log N) Suchschritte im Netz benötigt, wobei N die Anzahl der Knoten im Netz ist.

Nur wenn keine Übereinstimmung genau genug ist, muss die Anfrage zeitaufwändig selbst beantwortet werden (Wie dies geschieht wird noch nicht behandelt). Das Ergebnis dieser Anfrage kann dann in den Speicher jedes Knotens, der mit identifier[*l*] bestimmt wurde, geschrieben werden und steht für zukünftige Anfragen zur Verfügung

5.2 Komplexe Anfragen

Triantafillou und Pitoura stellen in [TrPi03] ein anderes Framework vor, das aufsetzend auf Chord, komplexe Anfragen ermöglicht. Im Gegensatz zu dem eben genannten ermöglicht es viele Anfragetypen, wie sie mit einer Sprache wie SQL formuliert werden können.

Es wird zwischen zwei Anfragetypen unterschieden, einfache Anfragen und komplexe Anfragen. Einfache Anfragen werden direkt umgesetzt, komplexe Anfragen werden dadurch gelöst, dass sie auf einfache Anfragen abgebildet werden. Folgende Kürzel werden dabei benutzt: SR für Single Relation; MR für Multi Relation; SA für Single Attribute; MA für Multi Attribute.

Einfache Anfragen sind:

- [SR, SA, =]: Eine Anfrage über eine Relation mit einem Vergleich eines Attributs mit einem Zahlenwert.
- [SR, SA, <>]: Anfrage über eine Relation mit einem Attribut in einem Intervall.
- [MR, Ma, join]: Der Join zweier Relationen über jeweils ein Attribut.

Komplexe Anfragen sind:

- [SR, MA, =]: Anfrage über eine Relation und Vergleich mehrerer Attribute mit Zahlenwerten.
- [SR, MA, <>]: Anfrage über eine Relation mit mehreren Attributen jeweils in einem Intervall.
- [MR, MA, =]: Anfrage über mehrere Relationen und Vergleich mehrerer Attribute mit Zahlenwerten.
- [MR, MA, <>]: Anfrage über mehrere Relationen mit mehreren Attributen jeweils in einem Intervall.
- [MR, MA, =, sf]: Anfrage über mehrere Relationen und Vergleich mehrerer Attribute mit Zahlenwerten und einer speziellen Funktion wie *min*, *max*, *sum* , *order by* oder *group by*.

Alle Objekte werden in einer Relation $R(DA_1, DA_2, .. , DA_k)$ gespeichert, wobei A der Name des Attributs ist und DA die Domäne des Attributs. Bisher können nur ganze Zahlen als Domäne genommen werden. Weitere Datentypen wie String oder Date müssen noch implementiert werden.

Gemäß dem Chord-Protokoll erhält jeder Knoten einen m-Bit Bezeichner, der dessen Position im Ring angibt. Dazu wird die IP-Adresse des Knotens gehasht. Auf ähnliche Weise bekommt jedes Tupel einen eindeutigen m-Bit Bezeichner. Dazu erhalten die Tupel jeweils einen eindeutigen Schlüssel t, basierend auf den Werten der Attribute, der als Parameter der Hash-Funktion h dient. Es entsteht die Zuordnung (h(t), Tupel) und jedes Tupel wird auf dem Knoten succ(h(t)) gespeichert. Um eine Suche nach jedem Attribut des Tupels zu ermöglichen, werden bei k Attributen k Kopien erzeugt, wobei jedes Mal das entsprechende Attribut A_k der Hash-Funktion zugrunde liegt mit $h(a_1), h(a_2), .. , h(a_k)$. Der Speicherort des Tupels ist dann der Knoten $succ(h(a_i))$ für $i \in \{1, .. k\}$.

Um viele Anfragen leichter durchführen zu können, werden besondere Anforderungen an die Hash-Funktion gestellt. Die erste Anforderung ist, dass sie Reihenfolge-erhaltend (monoton) ist.

$$\forall v1, v2 \in DA_i : v1 < v2 \Rightarrow h(v1) < h(v2) \qquad (4)$$

Die zweite Anforderung ist, dass eine Gleichverteilung der Werte einer Domäne auf den Ring erfolgt. Dazu wird die Domäne auf den Bereich 2m skaliert. Vorraussetzung hierfür ist, dass |DA| < 2m.

Sei DA={low, .. , high}, dann wird der Hash-Raum in 2m/s Bereiche geteilt, wobei s definiert wird als

$$s = \frac{2^m}{(high - low + 1)} \qquad (5)$$

Für jeden Wert $a \in DA$ wird dann die Hash-Funktion wie folgt angewandt:

$$h:DA \rightarrow \{0, .. 2^m-1\} \quad h(a) = \lceil (a - low) \cdot s \rceil \qquad (6)$$

5.2.1 Einfache Anfragetypen

Mit diesen Anforderungen ist das System in der Lage, Anfragen auszuführen.

Anfragetyp [SR, SA, =]
Eine mögliche SQL Anfrage für diesen Anfragetyp sieht wie folgt aus:

```
SELECT *
FROM R
WHERE R.A = a
```

Der entsprechende Pseudocode sieht folgendermaßen aus (mit n ist der Knoten bezeichnet, der die Anfrage stellt):

```
INPUT: a_i   DA_i, for i ∈ {1, 2, .. k}
OUTPUT: Eine Tupel-Liste, mit Wert a_i im Attribut A_i
BEGIN
calculate h_i(a_i) //unter Benutzung der für die Domäne DA_i
                   //entsprechenden Hash-funktion h_i
n_target = Chord_lookup h_i(a_i) ausgeführt auf Knoten n
Abfrage-und-Entgegennahme der gewünschten Tupel von n_target
END
```

Gesucht wird nur nach dem einen Attributwert a_i. Es wird mittels der von Chord zur Verfügung gestellten Methode lookup nach $h_i(a_i)$ gesucht. Chord findet dann den Knoten succ($h_i(a_i)$). Dabei ist h_i die hash-Funktion für die entsprechende Domäne DA_i des Attributs A_i. N_target ist der Knoten, der eine Kopie des gesuchten Tupels enthält. Dieser Knoten führt eine lokale Suche nach den gewüschten Tupel durch und schickt alle Tupel die der Anfrage entsprechen zurück zu n.

Da diese Anfrage auf der lookup Methode von Chord beruht, kann die Anfrage mit O(log N) routing-Schritten ausgeführt werden.

Anfragetyp [SR, SA, <>]
Eine mögliche SQL Anfrage dieses Typs sieht wie folgt aus:

```
SELECT *
FROM R
WHERE R.A IN [low, high]
```

Der Algorithmus ist in zwei Teile gegliedert.

```
INPUT: (low, high) ⊆ DA_i, for i ∈ {1, 2, .. k}
OUTPUT: Eine Tupel-Liste, mit Attribut A_i
          im Bereich (low, high)
BEGIN
calculate h_i(low) und h_i(high) //mittels der für DA_i entsprechenden
          Hash-Funktion h_i
n_start = Chord_lookup h_i(low) // ausgehend von Knoten n
n_end = Chord_lookup h_i(high) //ausgehend von Knoten n
forward (query, n, n_end) zu Knoten n_start
Entgegenahme der angefragten Tupel
END
```

Es werden zuerst die beiden Knoten succ(h_i(low)) und succ(h_i(high)) ermittelt. Da wieder dazu die lookup-Methode von Chord genutzt wird, kann die Anfrage in O(log N) Suchschritten ausgeführt werden. Da die Hash-Funktion Reihenfolge erhaltend ist, liegen alle gesuchten Tupel in Knoten, die im Chord-Ring zwischen succ(h_i(low)) und succ(h_i(high)) liegen. Die Anfrage wird zuerst an den Knoten succ(h_i(low)) geschickt, welcher die Anfrage an seinen Nachfolger im Ring weitergibt, bis schließlich succ(h_i(high)) erreicht wird.

```
Forward (query, n, n_end), ausgeführt auf Knoten n_j
INPUT: (low,high)   DA_i for i   {1, 2, .. k}
OUTPUT: Eine Tupel-Liste, mit Attribut A_i
          im Bereich (low, high)
BEGIN
local_find(R, A_i, a_i), mit a_i ∈ (low, high)
Sende Ergebnisse der Anfrage an Knoten n
IF n_j ≠ n_end
```

```
    forward (query, n, n_end) zu Knoten succ(n_j)
END
```

Der Algorithmus fängt bei n_start an. Zuerst wird eine lokale Suche ausgeführt nach allen Tupeln bei denen das Attribut A_i im entsprechendem Bereich (low,high) liegt. Die Ergebnisse werden direkt zu n zurückgegeben. Wenn der aktuelle Knoten nicht n_end ist, so wird die Anfrage an den direkten Nachfolger weitergeschickt. Dies wiederholt sich, bis der Knoten n_end erreicht wird,

Die Kosten der Anfrage entsprechen der Anzahl an Knoten, welche zwischen succ(h_i(low)) und succ(h_i(high)) liegen. Dies kann im besten Fall O(1) sein, im schlechtesten Fall, wenn der gesuchte Bereich die komplette Domäne DA_i umfasst, sind N Anfrageschritte nötig. Auch der Durchschnitt liegt im Bereich O(N).

Aufgrund der schlechten Performanz der Anfrage wird eine besondere Klasse von Knoten, Range Guards eingeführt (kurz RG). Jeder RG ist für einen bestimmten Bereich des Ringes zuständig und enthält Kopien aller Tupel, die ein Attribut besitzen, dass in diesen Bereich fällt. Dabei werden nur Attribute berücksichtigt, von denen erwartet wird, dass Bereichsanfragen über diese Attribute gestellt werden,

Die Domäne von A wird in *l* disjunkte Zonen unterteilt, wobei die Größe einer solchen Zone gleich |DA|/*l* ist. Für jede Zone ist ein anderer RG zuständig. Auf gleiche Weise werden die Domänen der anderen Attribute in *l* Zonen geteilt. Da die Hash-Funktion die Domäne auf den Chord-Ring skaliert, sind die Range Guards für die jeweils gleichen Zonen unterschiedlicher Attribute identisch. Jeder Range Guard hat eine Referenz auf den RG, der für die nachfolgende Zone zuständig ist. Des Weiteren besitzt jeder Knoten in Chord eine Referenz auf den jeweils für die entsprechende Zone zuständigen RG.

Wenn nun eine Bereichsanfrage anliegt, so kann diese von den RG beantwortet werden, deren vereinigten Zonen mit dem Bereich LOW, HIGH umfassen. Dabei ist der Bereich (low,high) der Anfrage eine Teilmenge von (LOW,HIGH) der entsprechenden Range Guards. Eine anliegende Anfrage wird wieder zuerst an succ(h(low)) gesendet. Dieser Knoten kann nun die Anfrage an den zuständigen RG weiterleiten, welcher eine lokale Suche nach den angefragten Tupeln ausführen und die Ergebnisse an den anfragenden Knoten zurückschicken kann. Danach wird geprüft, ob high < HIGH ist. Wenn ja, so liegen alle angefragten Tupel im Bereich der bisher durchsuchten Range Guards. Im anderen Fall wird die Anfrage an den nachfolgenden RG weitergereicht.

In Abbildung 9 ist ein solcher Ring mit Range Guards dargestellt. Jeder Knoten besitzt eine Referenz auf den jeweils zuständigen Range Guard. Von einem Knoten sind zusätzlich alle Einträge aus der Finger-Tabelle als Referenz auf andere Knoten dargestellt. Die Range Guards haben nur Referenzen auf ihren direkten Nachfolger.

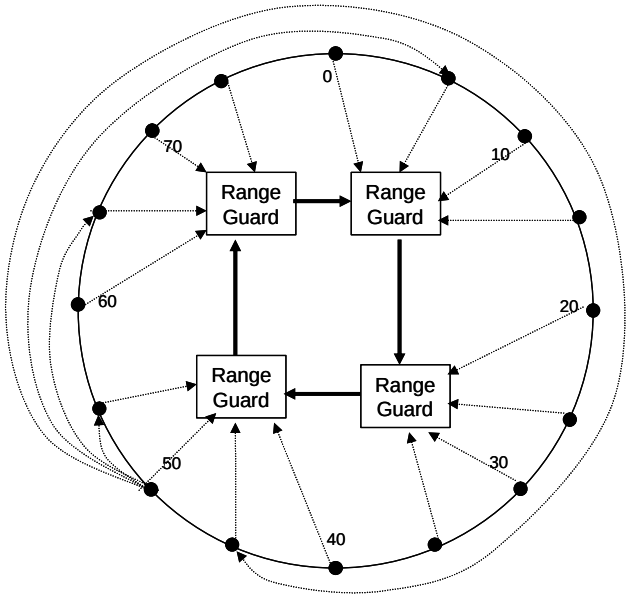


Abb. 9. Chord-Ring mit Range Guards

Auf diese Weise braucht die Bereichsanfrage O(log N) Schritte, um den Knoten succ(h_i(low)) zu erreichen und weitere O(l) Schritte (bei l verwendeten Range Guards) um die Anfrage innerhalb der RG zu beantworten. Da l weitaus kleiner ist als N, erhöht die Erweiterung des Algorithmus durch die Range Gards die Effizienz bedeutend. Wenn l = log N gesetzt wird, so kann die komplette Anfrage in O(log N) + O(log N) = O(log N) Schritten ausgeführt wird. Alternativ kann l = $\sqrt{N}$ gesetzt werden, was die

Auslastung der einzelnen Range Guards verringert, allerdings auch die Komplexität der Anfrage auf O($\sqrt{N}$) erhöht.

Anfragetyp [MR, MA, <>]
Eine mögliche SQL Anfrage dieses Typs ist

```
SELECT *
FROM R, M
WHERE R.a = M.b
    //mit R(DA1, ... , DAk) und M(DB1, ..., DBl)
    //und DAi {1, ..., n}, DBj   {1, .., m}
```

Sofern R.a und M.b vom gleichen Typ sind, gilt folgende Gleichung:

$$R.a = M.b \Longrightarrow h(R.a) = h(M.b) \Longrightarrow succ(h(R.a)) = succ(h(M.b)) \quad (7)$$

Dies bedeutet, dass keine Tupel zwischen den Knoten transferiert werden müssen, da bedingt durch die Hash-Funktion, die Tupel, die im entsprechenden Attribut den selben Wert haben auf dem selben Knoten gespeichert werden. Daher kann der Join lokal auf dem jeweiligen Knoten ausgeführt werden, und die Ergebnisse zum anfragenden Knoten zurückgeschickt werden.

Da die Hash-Funktion allerdings die Domäne des Attributs auf den gesamten Ring skaliert, muss die Join-Anfrage an alle Knoten im Ring geschickt werden, um alle möglichen Lösungen zu finden. Deshalb sind für die komplette Anfrage O(N) Suchschritte nötig, einhergehend mit einer hohen Netzauslastung und einer hohen Auslastung jedes einzelnen Knotens, da jeder Knoten eine Join-Anfrage stellen muss.

Auch hier können wieder Range Guards eingesetzt werden, um die Effizienz zu erhöhen. Dazu wird eine Hash-Funktion benutzt, die die Tupel in eine festgelegte Anzahl disjunkter Partitionen (Mengen) einteilt. Innerhalb einer Partition sind genau die Tupel, bei denen der Wert des Join-Attributs in einen bestimmten Bereich fällt. Nun ist es möglich den Join innerhalb einer solchen Partition unabhängig der restlichen Partitionen auszuführen (oder auch parallel dazu). Wenn nun *l* Partitionen erzeugt werden, so kann jeweils genau eine Partition einem der *l* Range Guards zugeordnet werden. Danach kann die Join-Anfrage von diesen *l* Range Guards beantwortet werden. Es werden nicht alle Knoten im Ring benötigt, wie zuvor.

Für den Fall, dass *l* = log N, oder *l* = $\sqrt{N}$, so kann auch die Join-Anfrage in O(log N), bzw. O($\sqrt{N}$) beantwortet werden.

5.2.2 Komplexe Anfragetypen

Mit Hilfe der einfachen Anfragen können nun auch komplexe Anfragen aufgelöst werden. Dies geschieht dadurch, dass komplexe Anfragen durch mehrfache Ausführung einfacher Anfragen beantwortet werden können.

Anfragetyp [SR, MA, =]
Eine mögliche SQL-Anfrage dieses Typs sieht wie folgt aus:

```
SELECT *
FROM R
WHERE R.A = a
   OR R.B = b
  AND R.C = c
```

Diese Anfrage wird in drei Anfragen des Typs [SR, SA, =] aufgelöst, für jedes betrachtete Attribut eine. Die Anfragen können parallel ausgeführt werden, und werden jeweils an den Anfrageknoten zurückgeschickt. Dort werden die Ergebnismengen gemäß der Operatoren verknüpft. Durch die parallele Ausführung bleibet die Komplexität bei O(log N), allerdings entsteht eine höhere Netzauslastung, je mehr Attribute die Anfrage enthält.

Anfragetyp [SR, MA, <>]
Eine mögliche SQL-Anfrage dieses Typs sieht wie folgt aus:

```
SELECT *
FROM R
WHERE R.A IN (low, high)
  AND R.B IN (low, high)
```

Diese Anfrage wird über den mehrfachen Aufruf der Anfrage [SR, SA, <>] aufgelöst. Auch hier können die Teil-Anfragen parallel ausgeführt werden, so dass bei Einsatz von Range Guards die Komplexität bei O(l) bleibt.

Anfragetyp [MR, MA, =]
Diese Anfrage wird aufgelöst über den mehrfachen Aufruf von [SR, MA, =], welche wiederum aufgelöst wird über Aufrufe von [SR, SA, =]. Ein Sonderfall dieses Typs ist eine Join-Anfrage, wenn 2 oder mehr Attribute denselben Wert haben. Diese Anfrage kann dann über [MR, MA, join] gelöst werden.

Anfragetyp [MR, MA, <>]
Die Anfrage wird aufgelöst in mehrere Aufrufe von [SR, MA, <>], was wiederum aufgelöst wird in mehrere Aufrufe von [SR, SA, <>].

Anfragetyp [MR, MA, =, sf]
Je nach spezieller Funktion wird die Anfrage auf unterschiedliche Weise gelöst. Bei *group by* oder *order by* kann die Anfrage in die einfachen Anfragen [SR, SA, =] und [SR, SA, <>] aufgelöst werden. Die Funktionen *group* und *order* werden danach über die Ergebnistupel ausgeführt. Dabei kann man ausnutzen, dass die Hash-Funktion Reihenfolge-erhaltend ist und die Teilmengen schon vorsortiert sind.

Auch für die Aggregatfunktionen *min* und *max* kann man die Reihenfolge-Erhaltung der Hash-Funktion ausnutzen. Für *min* oder *max* Anfragen müssen nicht mehrere Knoten durchsucht werden. Bei min reicht es, den Ring bis zum ersten Treffer zu durchsuchen. Alle anderen Tupel, die weiter hinten im Ring liegen, haben zwangläufig einen höheren Wert. Lediglich bei Funktionen wie *sum*, *average* müssen alle Knoten durchsucht werden. Auch hier können Range Guards die Anzahl der Suchschritte reduzieren.

5.2.3 Erweiterungen des Protokolls

Es existieren einige Erweiterungen, um die Effizienz des Protokolls zu verbessern. Die erste Erweiterung, die Einführung einer besonderen Klasse von Knoten, den Range Guards, wurden schon erwähnt. Allerdings sind die Anforderungen in Bezug auf Rechenleistung und Netzwerkanbindung der Knoten bei Range Guards höher, als bei den restlichen Knoten.

Um die hohe Speicherbelastung durch die vielen Kopien der Tupel entgegenzuwirken ist es möglich, jedes Tupel nur noch einmal im Ring zu speichern, z.B. auf dem Knoten succ(t), wobei t der Primärschlüssel des Tupels ist. Alle anderen Knoten speichern nun statt dem kompletten Tupel nur eine Referenz auf succ(t). Als Nachteil müssen mehr Suchschritte pro Anfrage in Kauf genommen werden (O(log N) Schritte werden zusätzlich benötigt, um das entsprechende Tupel anzufragen).

Eine weitere Möglichkeit die hohe Redundanz der Tupel zu umgehen besteht bei Join-Anfragen. Da jedes Tupel mit k Attributen k+1 mal im Ring auftaucht, und bei einer Join-Anfrage jeder Knoten durchsucht werden muss, wird auch jedes Tupel k+1 mal betrachtet. Hier reicht es aus, wenn jeder Knoten bei der lokalen Suche nur die Tupel betrachtet, deren Hash-Wert für das entsprechende Attribut auf diesem Knoten liegt. Hier wird die Anzahl der Suchschritte nicht verringert, jedoch wird die Zeit, die jeder Knoten für die Suche benötigt um den Faktor k verkürzt.

Wenn es mehrere Attribute gibt, bei denen angenommen wird, dass öfter Anfragen über alle diese anliegen, so kann eine Hash-Funktion benutzt werden, die auf alle Attribute anwendbar ist. Danach reicht für eine Anfrage über diese Attribute eine Anfrage im Chord-System. Sie muss nicht mehr in mehrere Teilanfragen aufgelöst werden, wie zuvor.

6. Zusammenfassung

P2P Systeme können mittlerweile weitaus mehr, als nur Dateien auszutauschen. Hierbei sind strukturierte Systeme den unstrukturierten in Bezug auf Routing und Suchanfragen überlegen. So zeigt das verbesserte CAN-System eine Anfragedauer, die kaum über der Verzögerung der zugrunde liegenden Netz-Struktur liegt. Probleme bereitet es jedoch immer noch, auf DHT basierenden Systemen komplexe Anfragen zu verarbeiten. Es gibt viele Ansätze, die einen bestimmten Anfragetyp zulassen, wie es in [HHH+02] oder [GuAA03] der Fall ist. Triantafillou und Pitoura sagen von ihrem Projekt, dass es zurzeit das einzige ist, dass eine breite Menge an Anfragen zulässt. Allerdings befindet sich das Projekt noch in der Entwicklung. So gibt es noch starke Einschränkungen, z.B. dass nur Integer als Wertebereich möglich ist. Die komplexen Anfragen über mehrere Attribute und mehrere Relationen selbst werden nur kurz behandelt. Auch wird die Thematik der anfallenden Kosten, wenn ein Knoten das Netz verlässt, oder neu in das Netz hinzukommt noch nicht behandelt. Die Forschung auf dem Gebiet Query-Verarbeitung ist

noch lange nicht abgeschlossen. Ob sich die Technik der strukturierten P2P-Systeme hier durchsetzen kann wird sich erst in Zukunft herausstellen.

7. Literaturverzeichnis

[AbHa02] Karl Aberer, Manfred Hauswirth. Peer-to-peer information systems: concepts and models, state-of-the-art, and future systems. *ICDE2002.*
(Elektronisch verfügbar unter http://lsirpeople.epfl.ch/hauswirth/papers/ICDE2002-Tutorial.pdf)

[GuAA03] Abhishek Gupta, Divyakant Agrawal, Amr Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In *Proceedings of the 2003 CIDR Conference* .
(Elektronisch verfügbar unter http://www-db.cs.wsc.edu/cidr/program/p13.pdf)

[HHH+02] Matthew Harren, Joseph . Hellstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, Ion Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. *Electronic Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS)* .
(Elektronisch verfügbar unter http://www.cs.rice.edu/Conferences/IPTPS02/191.pdf)

[Ratn02] Sylvia Paul Ratnasamy. A Scaleable Content-Addressable Network. Ph.D. Thesis, October 2002
(Elektronsch verfügbar unter http://www.icir.org/sylvia/thesis.ps)

[RFH+01] Sylvia Ratnasami, Paul Francis, Mark Handley, Richard Karp, Scott Shenker. A Scaleable Content-Addressable Network. *SIGCOMM 2001*.
(Elektronisch verfügbar unter http://www.acm.org/sigs/sigcomm/sigcomm2001/p13-ratnasamy.pdf)

[Scho01] Rüdiger Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications.
(Elektronisch verfügbar unter http://csdl.computer.org/comp/proceedings/p2p/2001/1503/00/15030101.pdf)

[SMD+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. Chord: A Scaleable Peer-to-peer Lookup Service for Internet Applications. *SIGCOMM 2001.*
(Elektronisch verfügbar unter http://www.pdos.lcs.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf)

[TrPi03] Peter Triantafillou, Theoni Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-peer Data Networks.
(Elektronisch verfügbar unter http://www.ceid.upatras.gr/faculty/peter/papers/dbisp2p.pdf)