Seminararbeit

Consistent Hashing und Balancing in DHTs

Informationsverwaltung in Netzen
Sommersemester 2003

Simon Rieche

Institut für Informatik
Freie Universität Berlin

Berlin, Mai 2003

Betreuer:
Dr. Artur Andrzejak,
Konrad-Zuse-Zentrum für Informationstechnik Berlin

Inhaltsverzeichnis

1 Einführung

In einem Peer-to-Peer (P2P) System kommunizieren Rechner direkt miteinander, ohne einen zentralen Rechner. Da in einem Peer-to-Peer System Knoten nur Informationen über ihre Nachbarschaft besitzen, müssen sie ihre Arbeit mit unvollständigen Informationen verrichten.
Um trotzdem globale Informationen persistent zu speichern, werden sogenannte Distributed Hash Tables verwendet. Distributed-Hash-Tables (DHT) sind über mehrere Rechner verteilte Hashtables, das heißt es wird jeder im P2P-System zu veröffentlichende Eintrag als Schlüssel eindeutig auf ein Wert abgebildet. Der Wertebereich wird in Intervalle aufgeteilt, die einzelnen Knoten zugeteilt werden. Diese Knoten sind dann dafür zuständig, die Anfragen zu bearbeiten, die in seinen Wertebereich fallen [1].
Zum Suchen von Einträgen im Peer-to-Peer System existieren verschiedene Strategien. Eine ineffiziente Lösung ist das Broadcasten des Netzes, d.h. das Senden der Abfrage an alle Rechner im Netz (Flooding), wodurch aber eine hohe Netzlast entsteht. Es gibt effizientere Lösungen mit Aufteilung des Namensbereichs, wie es z.B. die Systeme CAN (Content-Addressable Network) [7] oder Chord [9] betreiben. So kann mit weniger Netzlast der richtige Knoten gefunden werden.
Es wird gezeigt, dass in den original Distributed Hash Tables die die mögliche Unterstützung von effizienten Bereichsabfragen oder regulären Ausdrücken (find [abc]*grid*.pdf) fehlt.
In dieser Arbeit wird das Verfahren des Consistent Hashing beschrieben. Anschließend wird das Systems Chord, als ein Vertreter von Distributed-Hash-Tables, untersucht und kurz mit einem weiteren System, CAN, verglichen. Zum Schluss sollen die Nachteile von DHT-Systemen gezeigt werden.

2 Consistent Hashing

2.1 Motivation

„Consistent Hashing and Random Tress“ ist eine Entwicklung am MIT in Cambridge [3]. Entwickelt wurde es, um sogenannte Hot Spots im Internet zu verhindern. Hot Spots sind Rechner im Netz, die so viele Anfragen von Clientrechner bekommen, dass sie unter der Last zusammenbrechen oder stark verzögert antworten. Dies ist für Benutzer gleichermaßen wie für Betreiber unbefriedigent. Das Konzept ist aber auch für andere Client-Server-Situationen anwendbar.

2.2 Verschiedene Caching-Strategien

Es gab vorher schon Ansätze durch Caches die Last auf verschiedene Server zu verteilen. Es sollen ein paar Strategien vorgestellt werden, aus dem das Consistent Hashing entstand.

2.2.1 Proxy als Cache

Ein erster Ansatz war, Webseiten in so genannten Proxies zu cachen. Dabei teilen sich mehrere Clients ein Proxy, der die am meisten frequentierten Webseiten lokal als Kopie speichert, und die Daten an seine Clients zurückgibt, ohne dass diese sich direkt an den eigentlichen Urpsprungsserver der Webseite wenden müssen. Aber so können die Caches selber zu Hot Spots werden, die durch die Last beeinträchtigt werden.

2.2.2 Caches mit Multicast

Ein weiterer Vorschlag in [5], mehrere Caches zu nehmen schlägt am Netzverkehr fehl, der dabei erzeugt wird. In diesem Modell fragen die Clients eine Cache nach einer Datei o.ä., und erhalten von diesem das Ergebnis, wenn er die Seite lokal gespeichert hat. Sind die Daten aber nicht im Cache, wird die Anfrage an alle anderen Caches per Multicast weitergeleitet, was zu der angesprochen hohen Netzlast führt. Je mehr Caches sich beteiligen, um so größer wird der zu managende Datenstrom, das Konzept skaliert nicht.

2.2.3 Caches in Baumstruktur

Ein effizienter Ansatz in [2] ist, die Caches in einem virtuellen Baum anzuordnen. Die Anfragen zu einem Datum werden an ein Blatt gesendet. Diese prüft, ob sie oder seine Geschwisterknoten die gesuchten Daten im

Cache haben. Sollten diese Daten nicht vorhanden sein, wird die Anfrage an die Elternknoten gesendet. Dies wird solange fortgeführt bis ein Knoten die gewünschten Daten hat, oder die Wurzel des Baumes erreicht wird. Dann wird die Anfrage direkt an den eigentlichen Server für die Daten der Anfrage geschickt. Alle Knoten merken sich dabei immer die Daten, die sie vom höher gelegenden Knoten erhalten haben für eine gewisse Zeit. Der Vorteil ist, dass die Knoten nur Anfragen von ihren Kindern oder Geschwistern bekommen, so dass nicht viele Anfragen simultan beantwortet werden müssen. Der Schwachpunkt ist aber vor allem die Wurzel. Dadurch, dass sie alle Anfragen eine zeitlang speichert und die Daten von allen Servern holen muss, kann sie bei vielen Anfragen innerhalb kürzester Zeit, stark beansprucht werden. Außerdem müssen bis zum Erhalt der Daten möglicherweise so viele Proxies gefragt werden, wie die Höhe des Baumes ist, zuzüglich deren Geschwister.

2.3 Modell

Im folgenden werden folgenden Konventionen für die Beschreibung des Consistent Hashing und der Random Trees getroffen:

- Alle Anfragen werden von Browsern gestellt.
- Die Webseiten sind auf Servern gespeichert.
- Cashes sind extra Computer, die benutzt werden, um die Server vor dem Bombardement von Anfragen von Browsern zu schützen.

2.3.1 Hashfunktion

Die Hashfunktion bildet Objekte in ein Intervall ab. Dabei wird davon ausgegangen, dass die Objekte zufällig im Intervall verteilt werden, das heißt dass die Objekte im Intervall gleichverteilt abgebildet werden.

2.4 Random Trees

Als erstes werden die Random Trees (Zufallsbäume) beschrieben. Dabei wird davon ausgegangen, dass alle Knoten alle Caches kennen, die Latenz zwischen je 2 Maschinen immer 1 ist und alle Anfragen zur gleichen Zeit gestellt werden. Mit diesen Einschränkungen erreicht man mit Random Trees eine Verzögerung von $\Theta(logC)$, wobei C die Anzahl der Caches ist. Das Protokoll ist dabei eine Erweiterung der Baumidee aus 2.2.3 und [6]. Dadurch erhalten die Knoten nur Anfragen von Kindern. Um die obersten Knoten(z.B. die Wurzel) zu entlasten, wird für jede Seite ein zufällig generierter Baum

erzeugt. Dadurch ist jeder Cache nur Wurzel für wenige Seiten, die Last wird somit verteilt.

2.4.1 Protokoll

Jeder Seite wird einem Baum, dem so genannten Abstrakten Baum (abstract tree) zugeordnet. Dabei hat jeder Knoten d Kinder. Die Anzahl der Knoten im Baum ist gleich der Anzahl der Caches, und der Baum ist so gut es geht balanciert. Die Knoten werden durch ihre Reihenfolge bei der Breitensuche gekennzeichnet. Alle Anfragen werden als 4-Tupel gesendet. Dort ist die Identität des Absenders, der Name des gesuchten Seite, die Sequenz von Knoten durch die die Anfrage gehen soll und eine Sequenz von Caches die als diese Knoten fungieren sollen angegeben. Um die letzte Sequenz zu entscheiden, welcher Cache zur Zeit die Arbeit für ein gegeben Knoten macht, werden die Knoten auf Maschinen abgebildet. Die Wurzel des Baumes ist immer der Server der Seite. Alle anderen Knoten werden durch eine Hashfunktion h auf die Caches zugeteilt. Diese Hashfunktion muss allen Browsern und Caches bekannt sein. Die Hashfunktion h lautet $h : P \times [1...C] \longrightarrow C'$, wobei P die Menge aller Seiten ist, C die Anzahl der Caches der Menge C'. Um nicht zu viele Seiten zu speichern, wird durch einen Parameter q festgelegt, ab wievielen Anfragen ein Cache die Seite als Kopie speichert.
Mit der Hashfunktion h, und den Parametern d und q ergibt sich folgendes Protokoll.
Browser Wenn ein Browser eine Seite wünscht, es wählt ein zufälligen Pfad vom Blatt zur Wurzel, bildet die Knoten mit h auf Maschinen ab, und fragt das Blatt nach der Seite. Die Anfrage enthält die eigene Identität, den Namen der gesuchten Seite, den Pfad, und das Resultat der Abbildung.
Cache Wenn ein Cache eine Anfrage erhält, überprüft er ob er lokal eine Kopie hat, oder er gerade eine Kopie erhält. Ist dies vorhanden sendet die Antwort an den Anfrager zurück. Andernfalls inkrementiert er den Zähler für die Seite und schickt die Anfrage an den nächsten Knoten im Pfad. Wenn der Zähler q erreicht, speichert er eine Kopie vom Dokument.
Server Wenn denn Server eine Anfrage erreicht, sendet er eine Kopie an den Absender der Anfrage.

2.4.2 Analyse und Vorteile

Die Latenz beträgt maximal $2\,log_d\,C$ wenn die Anfrage bis zum Server geht, wenn ein Maschine auf dem Pfad eine Kopie hat entsprechend weniger. Die Latenz zwischen je 2 Maschinen beträgt nach obiger Definition immer 1. Dies ist also gleichbedeutend mit der Anzahl der befragten Caches. $2\,log_d\,C$ entspricht also genau der doppelten Höhe des Baumes, der in jeder Stufe d Kinder und insgesamt C Knoten hat.
In [3] wird gezeigt, dass die Anfragen auf die einzelnen Knoten in etwa

gleichverteilt sind und so kein Knoten zu sehr durch Anfragen beansprucht wird. Durch den Parameter q wird die Anzahl der im System gespeicherten Kopien klein gehalten, und nur von populären Seiten gibt es entsprechend viele Kopien. So skaliert die Methode besser als die anderen vorgestellten Cachingstrategien.

2.5 Consistent Hashing

Die Idee war, die Hashing-Methoden aus [6] zu erweitern. Vorher existierende hashbasierte Systeme funktionierten mit einer bekannten, festen Anzahl von Servern. In einem P2P-Ansatz oder im Internet allgemein, in dem die Rechner ständig ausfallen können, keine globale Sicht existiert und jeder Knoten nur seine Nachbarn kennt, versagen diese Methoden jedoch. Motiviert wird Consistent Hashing durch eine einfaches Schema der Datenreplikation. Der Server verteilt mittels Hashfunktion Kopien von Daten an verschiede Caches, die Clients fragen mit derselben Hashfunktion die Caches nach den Daten ab. Dies versagt aber, wenn die Maschinen, die die Caches darstellen, ausfallen oder anderweitig nicht vorhanden sind, oder jeder Client eine andere Menge an Caches kennt. Consistent Hashing löst das Problem der verschiedenen Sichten (view). Eine Sicht ist eine Menge von Caches, die ein Client kennt. Ein Client nutzt eine Consistent Hash Funktion, um ein Objekt auf einen Cache in seiner Sicht abzubilden.

2.5.1 Definitionen

In diesem Abschnitt werden einige Definitionen fürs Consistent Hashing eingeführt. I ist die Menge der Einträge, I' die Anzahl der Elemente darin und B die Menge der Buckets (Behälter) oder Caches.
Eine **ranged Hashfunktion** ist eine Funktion, die die Einträge(Seiten) für jeden View, auf die Knoten verteilt. $f_V(i)$ ist der Knoten zu dem der Eintrag i im View V abgebildet wird.
Eine ranged Hashfamilie **(ranged hash family)** ist eine Familie von ranged Hashfunktion.
Eine zufällige ranged Hashfunktion **(random ranged hash function)** ist eine Funktion, die eine Funktion aus obengewählter Familie auswählt.
F ist im folgenden eine ranged hash family, f ist eine ranged hash function, V ist ein View, i ist ein Eintrag, und b ist ein bucket oder Cache.
Eine ranged Hashfunktion ist **balanciert**, wenn für ein gegebenes View eine Menge von Einträgen, mit einer zufälligen ranged Hashfunktion mit hoher Wahrscheinlichkeit jeder Bucket gleich viele Einträge hat.
Eine ranged Hashfunktion ist **monoton**, wenn für alle views S, T mit $S \subseteq T$ gilt: $f_T(i) \in S \Rightarrow f_S(i) = f_T(i)$. Falls Einträge (Seiten) durch f ursprünglich zu einem Bucket vom View S zugeordnet sind, und dann S zu einem View T

(mit mehr Buckets) erweitert wird, dann kann f_T die Einträge zwar zu einem neuen Bucket (in T, nicht in S) zuordnen, aber nicht zu einem anderen alten Bucket in S.

2.5.2 Konstruktion

In diesem Abschnitt wird die Konstruktion einer solchen ranged hash family mit „guten“ Eigenschaften. Eine Funktion bildet buckets, eine andere Einträge zufällig auf Punkte in einem Einheitsintervall ab. Der Eintrag i wird dann dem Bucket zugeordnet, dessen Bild am nächsten liegt. Zu Beweiszwecken muss jedem Bucket mehr als ein Punkt im Einheitsintervall zugeordnet werden. Für eine Anzahl von Buckets von weniger als C (Anzahl der Caches), muss jeder Bucket $k \log(C)$ mal repliziert werden, für ein konstantes k. Dazu wird jeder Bucket $k \log(C)$-mal repliziert, und die Hashfunktion verteilt diese Kopien zufällig. Diese Funktionen bilden die ranged Hashfamilie, welche unter anderem monoton und balanciert ist. Wenn ein neuer Bucket hinzugefügt wird, müssen nur die Elemente die nun am nächsten beim neuen Punkt liegen verschoben werden, zwischen vorhandenen Buckets werden keine Einträge verschoben.

2.5.3 Implementation

Consistent Hashing ist eine neue Hashfunktion. Diese bildet Einträge in Behälter in einer Sicht (View) ab. Eine View ist eine Menge von Caches, die ein bestimmter Client berücksichtigt. Ein Client verwendet eine consistent Hashfunktion, um ein Eintrag in einen der Caches in seiner Ansicht abzubilden.
Die genaue Funktionsweise wird anhand von [4] erklärt. Eine einfache Implementation der consistent Hashfunktion genügt vollkommen. Eine Standard Hash Funktion bildet Strings (Zeichenketten) in ein Zahlenintervall [0,...,M] ab. Geteilt durch M ergibt das eine Hashfunktion im Bereich [0,1]. Dies wird der Reihe nach auf dem Einheitskreis angeordnet. Dadurch wird jede URL auf einen Punkt auf dem Kreis abgebildet. Zur gleichen Zeit wird jeder Cache ebenfalls zu einem Punkt auf dem Kreis abgebildet. Nun ist für jede URL derjenige Cache zuständig, der im Uhrzeigersinn der nächstgelegende ist. Aus den oben erwähnten Gründen wird jeder Cache mehrmals im Intervall abgebildet.
Die folgende Abbildung aus [4] zeigt in (i) wie die Dokumente 1-4 auf die jeweils nachfolgenden Server A und B verteilt sind. Server A ist für die Dokumente 1-3, Server B für die Dokumente 4-5 zuständig. In (ii) werden die Dokumente 1 und 2 dem neu hinzugekommen Server C zugeordnet.

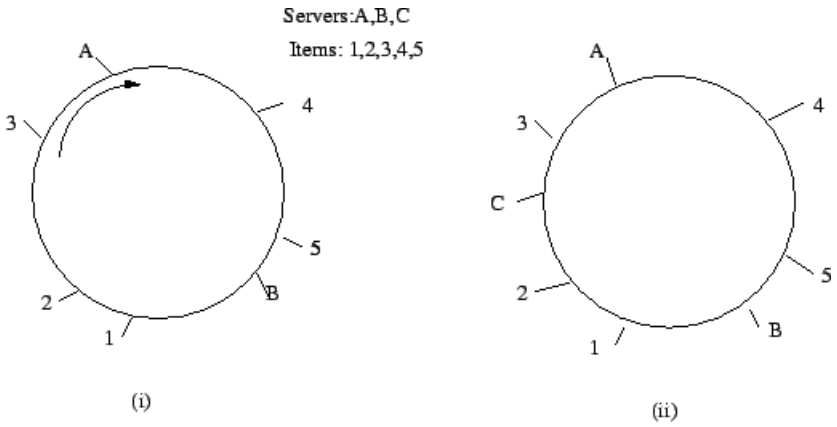

Abbildung 1: Kreis im Consistent Hashing

Alle Caches werden in einem virtuellen Binären Baum angeordnet. Der im Uhrzeigersinn nachfolgende Knoten zu einer gehashten URL kann durch Suchen im Baum gefunden werden. Dies liefert Consistent Hashing für n Caches in $O(log\, n)$.
Ein Theorem beschreibt die Eigenschaften für ein System mit m Caches und c Clients, jeder mit einer willkürlichen Sicht (view) über die Hälfte der Caches. Wenn $\Theta(log\, m)$ Kopien von jedem Cache existieren und die Kopien und URLs auf einen Einheitskreis abgebildet werden mit einer „guten“ Hashfunktion, dann werden folgende Eigenschaften erreicht:

- **Gleichverteilung**
  In jedem View sind die URLs gleichverteilt über die Caches
- **Last**
  über alle Views hat kein Cache mehr als $O(log\, c)$ mal die Durschnittliche Anzahl von URLs.
- **Ausdehnung**
  keine URL ist in mehr als $O(log\, c)$ Caches gespeichert.

2.5.4 Bewertung und Nachteile

Consistent Hashing wurde entwickelt um Server mit Caches zu entlasten. Es löst die Aufgabe indem die Dokumente durch Hashfunktionen den im Intervall nächstgelegenden Caches zugeordnet werden. Diese Zuordnung entspricht aber nicht der realen räumlichen Zuordnung der Maschinen. Zum Navigieren im Intervall gibt es keine Möglichkeit die Suche abzukürzen, es muss immer vom Eintrittspunkt im Intervall der Kreis mit dem Nachfolgeknoten abgelaufen werden, um den zuständigen Knoten zu finden.

3 Distributed-Hash-Tables

Distributed-Hash-Tables (DHT) sind über mehrere Rechner verteilte Hashtables. In Distributed-Hash-Table-Systemen wird jedem zu speichernden Datum ein Hash-Schlüssel (key) mit einer Hashfunktion zugewiesen. Dies kann z.B. aus dem Dateinamen oder andere Metadaten berechnet werden. Unter diesem Schlüssel wird die Datei im System verteilt, um die Daten im System zu erreichen benötigt man diesen Schlüssel. Die Schlüssel sind in einen vorgegeben Wertebereich. Das gesamte Intervall aller möglichen Werte wird im Peer-to-Peer-System auf die einzelnen Knoten verteilt. Um eine Datei zu finden wird der Hashwert berechnet und nach dem entsprechenden zuständigen Knoten gesucht. Der Unterschied zwischen den verschiedenen DHT-Systemen ist die Vorgehensweise, wie das Netzwerk aufgebaut ist und wie der Suchprozess abläuft. Im folgenden wird das System Chord untersucht und anschließend kurz mit dem System CAN verglichen.

3.1 Chord

„Chord A Scalable Peer-to-peer Lookup Service for Internet Applications“ wurde vom Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek und Hari Balakrishnan am MIT Laboratory for Computer Science entwickelt [9]. Das Ziel war es ein skalierbares Protokoll zum Suchen von Daten in einem dynamischen Peer-To-Peer Netzwerk zu entwickeln. Das Chord Protokoll kennt dabei nur einen Befehl, lookup(key). Damit wird zu einen gegebenen Key die IP des zugehörigen Knoten geliefert. Chord sorgt dabei zusätzlich für die Aufrechterhaltung des Netzwerkes. Andere Funktionen des Peer-to-Peer-Systems müssen von den Applikationen realisiert werden, wie z.B. Authentisierung und Caching.

3.1.1 Funktionsweise

Das Chord-Protokoll spezifiziert wie der Aufenthaltsort von keys gefunden werden kann, wie neue Knoten zum System hinzugefügt werden können und wie sich beim Verlassen von Knoten zu verhalten ist.
Chord benutzt dabei eine Variante des consistent hashing um Knoten und Datenobjekten Schlüssel zuzuweisen. Dadurch, dass die einzelnen Knoten aber in Chord nicht alle anderen kennen müssen, skaliert Chord besser als consistent hashing.
Die Hashfunktion erzeugt m-bit identifier indem sie die IP-Adressen der Knoten abbildet. Beim Hashen eines keys, wie beim suchen etc., erzeugt sie ebenfalls einen m-bit Key-identifier.
Die identifier in Chord bilden einen virtuellen geordneten Kreis $modulo\, 2^m$. Mit dem successor (Nachfolger) und predecessor (Vorgänger) wird darin

navigiert. Der successor(k) eines Schlüssels k ist der im Kreislauf nächstfolgende Knoten n mit $n \geq k$. Die Schlüssel werden im Uhrzeigersinn mit 0 bis $2^m - 1$ bezeichnet. Der predecessor ist für Knoten n und Schlüssel k der nächste vorhergehende Knoten m mit $m < n$ oder $m < k$.
Als Beispiel zeigt folgende Abbildung 2 einen Kreis mit $m = 4$.

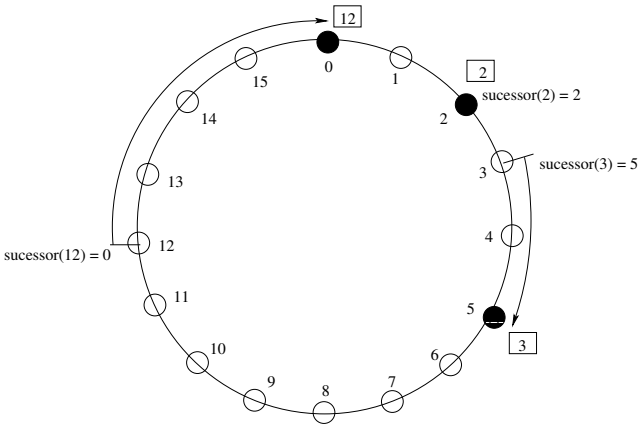

Abbildung 2: Beispiel eines Chord-Ringes

In diesem Beispiel existieren sechzehn (von 0 bis 15) Elemente, die im Kreis angeordnet sind. Im Kreis sind die Punkte 0, 2 und 5 durch Knoten besetzt. Da der successor vom Schlüssel 2 ebenfalls die 2 ist, kann key 2 in Knoten 2 lokalisiert werden. Ebenso wird key 3 in Knoten 5 gespeichert, in Knoten 12 der Schlüssel 9.Im oberen Beispiel ist also successor(2) = 2 für den Schlüssel 2 und successor(2) = 5 für den Knoten 2.

3.1.2 Finger-Tabelle

Da das Suchen von Daten in einem Ring, der nur Vorgänger und Nachfolger kennt, ineffizient ist, da unter Umständen der halbe Ring abgelaufen werden muss, speichert jeder Knoten n eine so genannte Finger-Tabelle. Der Nachfolger eines Knotens wird in dieser Finger-Tabelle ebenso wie der Vorgänger gespeichert. Außerdem zeigen die m Einträge(finger) in dieser Tabelle jedes Knotens auf Knoten innerhalb des Chord-Kreises. Diese Tabelle wird zum Suchen der zuständigen Knoten benutzt. Die Anzahl der Einträge und welche Knoten vorhanden sind, hängt also direkt von der Größe des Ringes m ab. In jeder Zeile der Finger-Tabelle wird ein Attribut start, interval und node gespeichert. Der i-te Eintrag in der Tabelle von Knoten n speichert dabei den ersten Knoten s, der mindestens 2^{i-1} keys von n entfernt ist. Also sind die Einträge $s = sucessor(n + 2^i)$, wobei $1 \leq i \leq m$ ist (alle Rechnungen modulo 2^m, um nicht aus dem Ring zu laufen). Wir nennen s den i-ten finger von Knoten n, bezeichnet wird er mit n.finger[i].node.

Ein Fingereintrag speichert die IP-Adresse und den Chord-identifier des entsprechenden Knotens. Der erste finger-Eintrag entspricht also genau dem Nachfolger. Alle Einträge, die in der Finger-Tabelle zusätzlich gespeichert sind, erklärt folgende Tabelle 1.

Bezeichnung	**Definition**	**Bedeutung**
finger[k].start	$(n + 2^{k-1}) \, mod \, 2^m$, $1 \leq k \leq m$	Intervallbeginn
finger[k].interval	$[finger[k].start, finger[k+1].start)$	Intervallgröße
finger[k].node	$erster \; Knoten \geq n.finger[k].start$	Zuständiger Knoten
successor	finger[1].node	Der Nachfolger des Knotens n
predecessor	Der Vorgänger des Knotens n	

Tabelle 1: Definition der Variablen für Knoten n

Die Intervalle der Einträge teilen den Kreis damit in Abschnitte ein. Wie dies zum Suchen von Daten genutzt wird, erklärt der nächste Abschnitt.
Die folgende Abbildung 3 zeigt für m=3 (6 Identifiers) die Finger-Intervalle für Knoten 1.

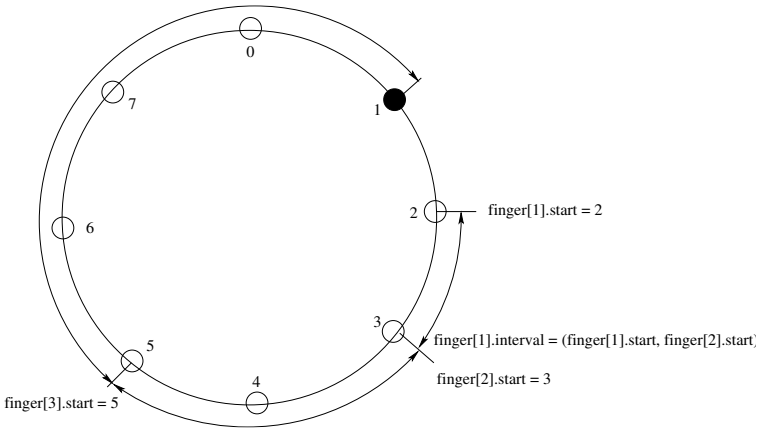

Abbildung 3: Intervalle im Chord-Ring

Die Intervalle beginnen im diesen Beispiel vom Knoten 1 also bei den Identifiern $(1 + 2^0) \, mod \, 2^3 = 2$, $(1 + 2^1) \, mod \, 2^3 = 3$ und $(1 + 2^2) \, mod \, 2^3 = 5$.

Eine komplette Fingertabelle aller Knoten in einem Chordring für m=4 und 3 besetzen Knoten zeigt die nachfolgende Abbildung 4. Es sind die Knoten 0, 2 und 7 besetzt.

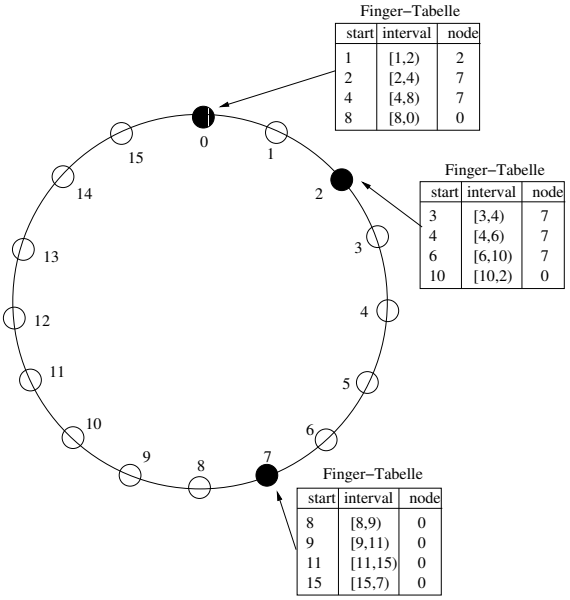

Abbildung 4: Fingertabellen der Knoten

3.1.3 Suche von Daten

Um zu einem Eintrag den verantwortlichen Knoten, also den Knoten der für den Schlüssel zuständig ist, zu finden, sucht man den predecessor des Schlüssels. Die Suchfunktion des Knoten an den die Anfrage gestellt wurde folgt dem Finger, der auf den am weitest entferntesten Knoten zeigt, der aber noch vor dem gesuchten Schlüssel liegen muss. Alle Knoten wiederholen dies so lange, bis der Schlüssel zwischen dem gefundenen Knoten und seinem successor liegt, denn dieser successor ist dann gleichzeitig der successor des gesuchten Schlüssels. Dies ist der gesuchte Knoten, der für die Daten zuständig ist.

3.1.4 Hinzufügen von Knoten

In einem dynamischen Netz können Knoten ständig dem System beitreten oder es verlassen. In diesem Abschnitt wird gezeigt wie Knoten dem System beitreten können, später wird das Verlassen des Rings behandelt.
Solange jeder Knoten seinen successor kennt, soll gewährleistet werden das Chord immer voll funktionsfähig ist. Dies ist eine Invariante des Systems.

Eine zweite ist, dass für jeden key k, der Knoten successor(k) zuständig ist für k.
Dazu werden drei Schritte für einen Knoten k, der neu in das System eintritt, ausgeführt.

- Initialisierung der finger und des predecessors von Knoten n
- Update der finger und predecessor der existierenden Knoten
- Die höher gelegene Applikation benachrichtigen, dass die Daten vom alten Knoten auf den neuen verschoben werden können.

Im Detail funktioniert ein Hinzufügen eines neuen Knotens wie folgt:
Wenn ein Knoten k in das bestehende Chord-System neu eintreten soll, muss ein Knoten j bekannt sein, der sich seinerseits schon im Chord-System befindet, an dem sich der neue Knoten anmeldet. Durch die Hashfunktion wird für den neuen Knoten von dem System einen Schlüssel im Chord-Kreis berechnet und zugewiesen. Der neue Knoten berechnet nun mit seinem erhaltenen Schlüssel und der Hashfunktion alle Intervalle und deren Startpunkte für seine finger-Tabelle. Vom alten Knoten bekommt er den Knoten geliefert der k im Kreis nachfolgt. Dann lässt er sich von seinem Nachfolgeknoten dessen predecessor geben, macht ihn zu seinem eigenen predecessor, und teilt seinem Nachfolger mit, dass er sein neuer predecessor ist. Der Knoten j sucht anschließend alle Knoten, die zu den Fingern von k gehören. Anschließend informiert er alle anderen Knoten, deren finger-Knoten er sein könnte, von seiner Existenz, also auch den von predecessor. Diese potentiellen Knoten findet er, in dem er alle verschiedenen Intervallgrößen (bis auf das längste) von sich aus rückwärts geht und von diesen Positionen den predecessor sucht. Zum Schluss wird die höher liegende Applikation veranlasst sämtliche Daten, für die nun der neue Knoten zuständig ist, vom successor zu kopieren, und dann die Daten anschließend zu löschen. Nun ist der neue Knoten einsatzfähig.
Die folgende Abbildung 5 zeigt die Fingertabellen, wenn zu den Knoten in Abbildung 4 noch Position 12 durch einen Knoten besetzt wird.
In der Abbildung sind alle Fingereinträge, die sich verändert haben in schwarz, Einträge die unverändert geblieben sind in grau.

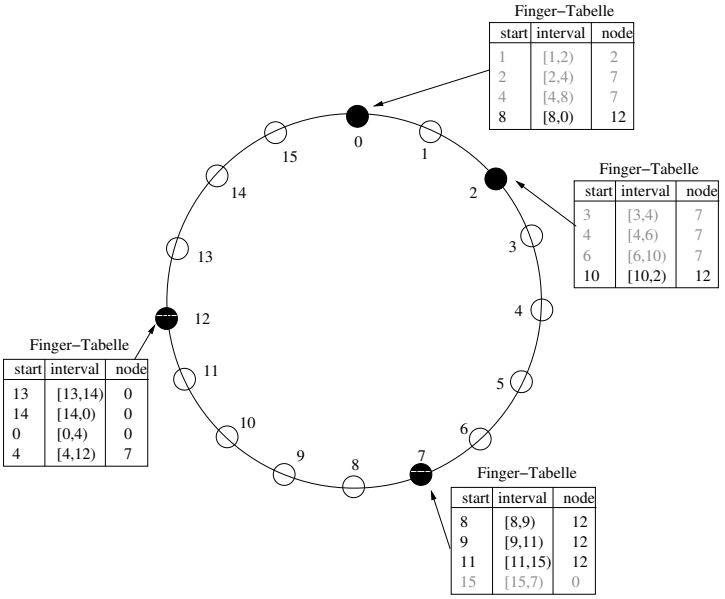

Abbildung 5: Nach dem Hinzufügen eines Knotens

3.1.5 Stabilisierung

Damit es zu keinem Fehlverhalten kommt, wenn ein Knoten ausfällt oder mehrere Knoten gleichzeitig dem Ring beitreten gibt es zusätzlich zu dem eigentliche Chord-Protokoll noch Stabiliesierungsroutinen.
In gewissen Abständen, die davon Abhängen wie oft Knoten dem System beitreten, überprüft jeder Chordknoten, ob sein successor wirklich noch aktuell ist. Dazu überprüft er, ob der predecessor des successor zwischen den beiden Knoten liegt. Dadurch ist dieser der wirkliche successor. Anschließend schickt er eine Nachricht an den neuen oder alten successor und teilt ihm mit, dass er sein predecessor ist. Dieser überprüft dies und trägt bei Korrektheit den neuen predecessor ein. Außerdem überprüft der Knoten in regelmäßigen Abständen zufällig nach obigen Schema seine finger auf Korrektheit.

3.1.5.1 Entfernen von Knoten

Jeder Knoten im Chord-Ring überprüft in regelmäßigen Abständen zufällig seine finger auf Korrektheit. Damit es nicht zu einem Zusammenbruch des Systems zu verhindern, falls der successor ausfällt, hat jeder Knoten noch eine Liste der nächst folgenden Knoten. Diese kann er durch eine Abfrage der successor-Knoten entlang des Rings erstellen. Falls nun durch die regelmäßigen Abfragen ein Knoten feststellt das sein successor Knoten ausgefallen ist, kann er ihn durch den ersten Eintrag aus dieser Liste ersetzen und abwar-

ten, bis das System sich durch seine oben genannten Stabilisierungsroutinen wieder korrigiert.
Die folgende Abbildung 6 zeigt die Fingertabellen, wenn von den Knoten in Abbildung 5 der Knoten an Position 2 verloren geht.

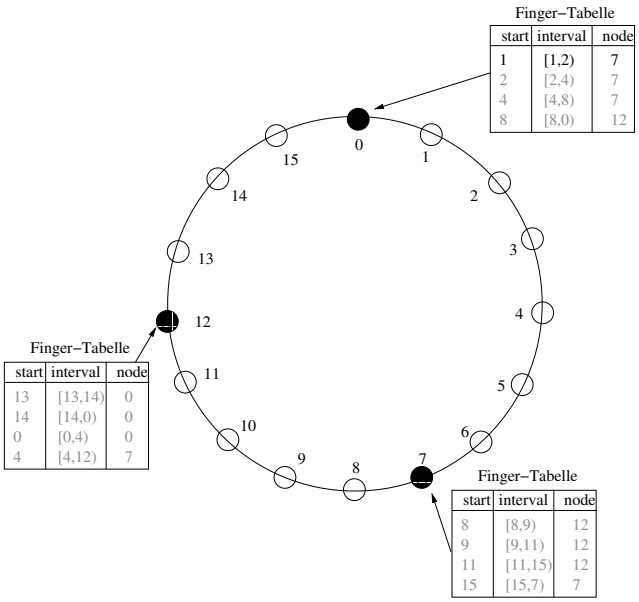

Abbildung 6: Nach dem Entfernen eines Knotens

In dieser Abbildung sind wieder alle unveränderten Fingereinträge in grau.

3.1.6 Lastverteilung

Die Last in dm System wird durch die Hash-Funktionen verteilt. Diese verteilt die Werte so über die Knoten, dass jeder Knoten in etwa die gleiche Anzahl von Elementen zu bearbeiten hat. Diese Methode der Lastverteilung erzeugt aber einige Nachteile, wie sie später beschrieben werden.

3.1.7 Laufzeiten

Durch das beschrieben Protokoll von Chord, muss jeder Knoten $O(logN)$ andere Knoten von insgesamt N kennen. Eine Suchoperation braucht $O(log\,N)$ und zum Verlassen oder Hinzufügen von Knoten sind $O(log^2\,N)$ Nachrichten nötig.

3.2 Vergleich

Folgende Tabelle gibt einen kurzen Vergleich von Chord zu CAN [7] einem Distributed-Hash-Table-System, das mit virtuellen multidimensionalen Feldern arbeitet.

Algorithmus	**Modell**	**Suchen**	**Nachbarn**
Chord	ein-dimensionaler Ring	$log(N)$	$O(logN)$
CAN	multi-dimensionales Feld	$dN^{\frac{1}{d}}$	$O(d)$

N= Anzahl der Knoten, d = Anzahl der Dimensionen

Tabelle 2: Vergleich der Systeme

3.3 Nachteile von DHT

Distributed Hash Tables haben vor allem den Nachteil, das keine effizienten Bereichsabfragen möglich sind. Um alle Einträge die zum Beispiel mit a anfangen zu finden, müssen alle Knoten abgefragt werden. Durch die Hashfunktion werden diese Einträge aber auf die Knoten verteilt, ohne das es möglich ist vorherzusagen, wieviele Elemente in diesem Intervall enthalten sind, und wo diese sich befinden. Um mit der Hashfunktion die Elemente zu finden müssten alle mögliche Einträge genau mit der Hashfunktion berechnet werden. Da dies ein hohen Aufwand bedeutet, müssen alle Knoten abgefragt werden. Da ein Knoten aber nicht alle Knoten kennt, sondern nur die Existenz seiner Nachbarn kennt, muss z.B. in Chord der Ring einmal komplett abgelaufen werden.

4 Fazit

Chord als Distributed-Hash-Tables-System bietet eine gute Möglichkeit Daten in einem heterogenen System zu speichern und mit geringem Aufwand zu finden. Durch intelligente Einteilung und Fingertabellen wird die Suche in $log(N)$ Schritten möglich. Lastbalancierung wird aber nur dadurch erreicht, das die Hashfunktion die Keys gleichmäßig im Intervall verteilt, ist dies durch die Hashfunktion nicht gewährleistet, kommt es zu Knoten die mehr Elemente haben als andere. Ein Wechsel der Hashfunktion ist im laufenden Betrieb nicht möglich, da alle Knoten und Daten neu geordnet werden müssen.
Ein weiterer Nachteil von Distributed-Hash-Tables wurde im vorherigem Abschnitt vorgestellt.
Es gibt aber noch einige Fragestellungen in Bezug auf Distributed-Hash-Tables, die noch nicht gelöst sind. Ein Beispiel sind die 15, in [8] gestellten, offenen Fragen. Eine ist z.B. ob die Nachbarn in $O(1)$ und die Pfadlänge in $O(log\, n)$ in einem DHT-Systemen liegen können (siehe Tabelle 2). Dies wäre das ideale System, denn die Pfadlänge bei der Suche, und der Aufwand beim Hinzufügen von Knoten wäre minimal.

A Literatur

[1] Balakrishnan, Hari; Kaashoek, M. Frans; Karger, David; Morris, Robert und Stoica, Ion.
Looking up data in P2P systems.
Communications of the ACM, 46(2):43–48, 2003.
ISSN 0001-0782.
http://doi.acm.org/10.1145/606272.606299.

[2] Chankhunthod, Anawat; Danzig, Peter B.; Neerdaels, Chuck; Schwartz, Michael F. und Worrell, Kurt J.
A Hierarchical Internet Object Cache.
In *USENIX Annual Technical Conference*, Seiten 153–164, 1996.
http://citeseer.nj.nec.com/chankhunthod95hierarchical.html.

[3] Karger, David; Lehman, Eric; Leighton, Tom; Levine, Mathhew; Lewin, Daniel und Panigrahy, Rina.
Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.
In *ACM Symposium on Theory of Computing*, Seiten 654–663, 1997.
http://citeseer.nj.nec.com/karger97consistent.html.

[4] Karger, David; Sherman, Alex; Berkheimer, Andy; Bogstad, Bill; Dhanidina, Rizwan; Iwamoto, Ken; Kim, Brian; Matkins, Luke und Yerushalmi, Yoav.
Web Caching with Consistent Hashing.
In *Proceedings of the 8th International WWW Conference*, Toronto, Canada, 1999.
http://www8.org/w8-papers/2a-webserver/caching/paper2.html.

[5] Malpani, Radhika; Lorch, Jacob und Berger, David.
Making World Wide Web Caching Servers Cooperate.
In *Proceedings of World Wide Web Conference*, 1996.
http://www.w3.org/Conferences/WWW4/Papers/59/.

[6] Plaxton, C. Greg und Rajaraman, Rajmohan.
Fast Fault-Tolerant Concurrent Access to Shared Objects.
In *IEEE Symposium on Foundations of Computer Science*, Seiten 570–579, 1996.
http://citeseer.nj.nec.com/plaxton96fast.html.

[7] Ratnasamy, Sylvia; Francis, Paul; Handley, Mark; Karp, Richard und Shenker, Scott.
A Scalable Content-Addressable Network.
In *Proceedings of ACM SIGCOMM, San Diego, CA*, 2001.
http://citeseer.nj.nec.com/ratnasamy01scalable.html.

[8] Ratnasamy, Sylvia; Shenker, Scott und Stoica, Ion.
Routing Algorithms for DHTs: Some Open Questions.
In *Proceedings of First International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.
`http://citeseer.nj.nec.com/ratnasamy02routing.html`.

[9] Stoica, Ion; Morris, Robert; Karger, David; Kaashoek, M. Frans und Balakrishnan, Hari.
Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.
In *Proceedings of ACM SIGCOMM, San Diego, CA*, 2001.
`http://citeseer.nj.nec.com/stoica01chord.html`.