Informationsverwaltung in Netzen
Sommersemester 2003

Consistent Hashing und Balancing in DHTs

Freie Universität Berlin
Institut für Informatik

Simon Rieche
rieche@inf.fu.berlin.de

Inhalt

- Random Trees
- Consistent Hashing
- Chord
  - Aufbau
  - Suchen von Daten
  - Einfügen und Entfernen von Knoten
  - Lastbalancierung
  - Nachteile
  - Vergleich

Consistent Hashing / Random Trees

- Entwickelt wurde es, um 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
- Dies ist für Benutzer und Betreiber unbefriedigend.
- Das Konzept ist aber auch für andere Client-Server-Situationen anwendbar.

Random Trees (1)

- Random Trees (Zufallsbäume)
- Dabei wird davon ausgegangen,
  - alle Knoten kennen alle Caches
  - die Latenz zwischen je 2 Maschinen ist immer 1
- Die Knoten erhalten nur Anfragen von Kindern.
- Um die obersten Knoten (z.B. die Wurzel) zu entlasten, wird für jede Seite ein zufällig generierter Baum erstellt.

Random Trees (2)

- 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, der Baum ist so gut es geht balanciert.
- Die Knoten werden durch ihre Reihenfolge bei der Breitensuche gekennzeichnet.

Random Trees (3)

- Alle Anfragen werden als 4-Tupel gesendet.
  - die Identität des Absenders
  - der Name des gesuchten Seite
  - die Sequenz von Knoten, durch die die Anfrage gehen soll
  - eine Sequenz von Caches, die als diese Knoten fungieren sollen

Random Trees (4)

- Die Wurzel des Baumes ist Server der Seite
- Alle anderen Knoten werden durch eine Hashfunktion h auf die Caches zugeteilt.
- Hashfunktion muss allen Browsern und Caches bekannt sein
- Die Hashfunktion h lautet h : P x [1...C] → C‘,
  - P die Menge aller Seiten
  - C die Anzahl der Caches der Menge C’.
- Um nicht zu viele Seiten zu speichern, wird durch einen Parameter q festgelegt, ab wie vielen Anfragen ein Cache die Seite als Kopie speichert.

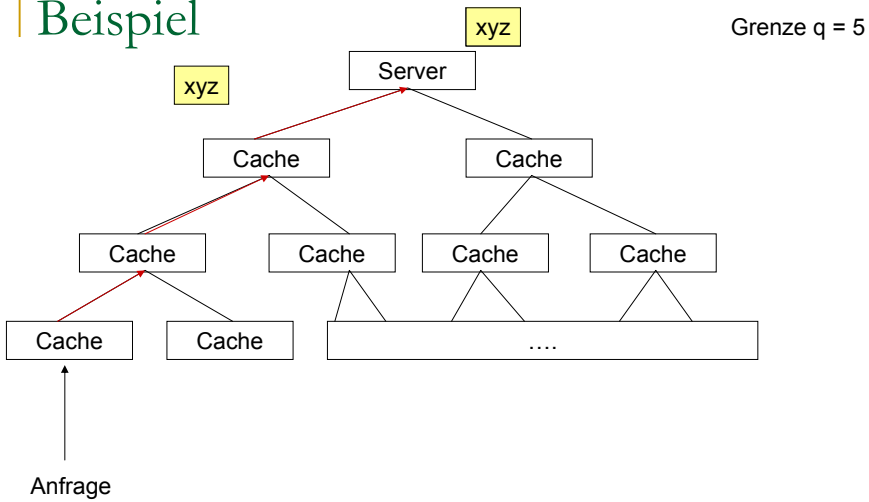
Protokoll (1)

- Browser
  - wählt ein zufälligen Pfad vom Blatt zur Wurzel
  - bildet die Knoten mit h auf Maschinen ab
  - Fragt das Blatt nach der Seite
  - Die Anfrage enthält
    - die eigene Identität
    - den Namen der gesuchten Seite
    - den Pfad
    - und das Resultat der Abbildung.

Protokoll (2)

- Cache
  - Überprüft wenn er eine Anfrage erhält, ob er lokal eine Kopie hat oder ob er gerade eine Kopie erhält.
  - Ist dies vorhanden sendet die Antwort an den Anfrager zurück
  - Andernfalls inkrementiert er den Zähler für die Seite
  - 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.

Beispiel



Vorteile

- Die Latenz beträgt maximal 2 $\log_d C$
  - Die Latenz zwischen je 2 Maschinen beträgt nach Definition immer 1.
  - Entspricht also der Anzahl der befragten Caches.
  - Entspricht der doppelten Höhe des Baumes, der in jeder Stufe d Kinder und insgesamt C Knoten hat.
- Die Anfragen werden auf die Knoten gleichverteilt
- Kein Knoten wird zu sehr beansprucht
- Durch q wird die Anzahl der im System gespeicherten Kopien klein gehalten
  - Kopien nur von populären Seiten
- Die Methode skaliert besser als andere Strategien.

Consistent Hashing

- Server verteilt mittels Hashfunktion Kopien von Daten an verschiede Caches
- 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
- jeder Client kennt eine andere Menge an Caches
- Consistent Hashing löst das Problem der verschiedenen Sichten (view).
  - Eine Sicht ist eine Menge von Caches, die ein Client kennt.

Konstruktion

- 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.
- Die Hashfunktion verteilt diese Kopien zufällig.
- Wenn ein neuer Bucket hinzugefügt wird, müssen nur die Elemente die nun am nächsten beim neuen Punkt liegen zum neuen Cache zugeordnet werden, zwischen vorhandenen Buckets werden keine Einträge verschoben.

Einfache Implementation (1)

- Eine Standard Hashfunktion 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 nächstgelegene ist.
- Aus Beweisgründen wird jeder Cache mehrmals im Intervall abgebildet.

Einfache Implementation (2)

- Alle Caches werden in einem virtuellen Binären Baum angeordnet.
- Der im Uhrzeigersinn nachfolgende Knoten zu einer gehaschten URL kann durch Suchen im Baum gefunden werden.
- Dies liefert Consistent Hashing für n Caches in O(log n).
- Eigenschaften für ein System mit m Caches und c Clients, jeder mit einer willkürlichen Sicht (view) über die Hälfte der Caches.
- Wenn (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 Durchschnittliche Anzahl von URLs.
  - Ausdehnung
    - keine URL ist in mehr als O(log c) Caches gespeichert.

Vorteile gegenüber Random Trees

- Jeder Rechner muss nur einige Caches kennen
- Anzahl der Caches muss nicht konstant sein
- Nicht für jede Seite neuer Kreis

Chord

- „Chord A Scalable Peer-to-peer Lookup Service for Internet Applications“
- Ziel ist ein skalierbares Protokoll zum Suchen von Daten in dynamischen Peer-To-Peer Netzwerk
- Chord Protokoll kennt nur einen Befehl, lookup(key)
  - Zu einen gegebenen Key die IP des zugehörigen Knoten geliefert
- Chord sorgt die Aufrechterhaltung des Netzwerkes.
- Andere Funktionen des Peer-to-Peer-Systems müssen von den Applikationen realisiert werden
  - z.B. Authentisierung und Caching.

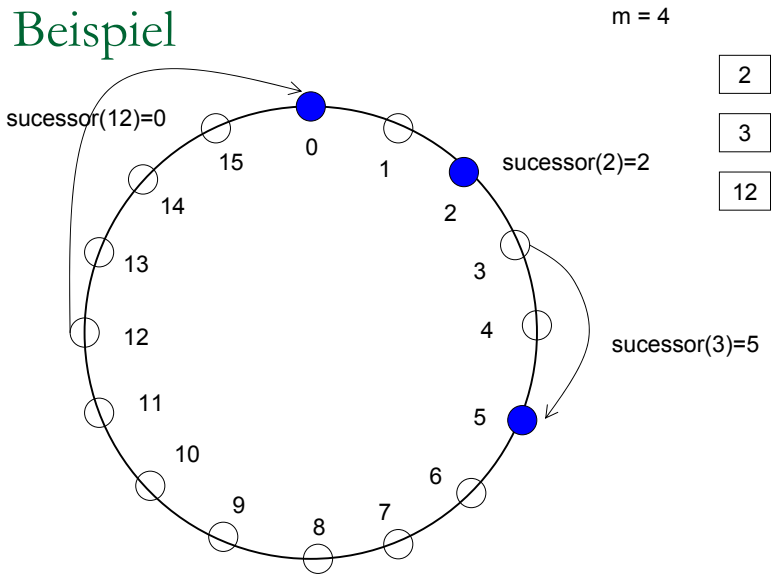
Chord

- Die Hashfunktion erzeugt m-bit Identifier
  - bildet die IP-Adressen der Knoten ab
  - 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.
- Die Schlüssel werden im Uhrzeigersinn mit 0 bis 2^m-1 bezeichnet

Successor

- Navigation mit
  - successor (Nachfolger)
  - predecessor (Vorgänger)
- Der successor(k) eines Schlüssels k ist der im Kreislauf nächstfolgende Knoten n mit n > k.
- Der predecessor ist für Knoten n und Schlüssel k der nächste vorhergehende Knoten m mit m < n oder m < k.

Beispiel



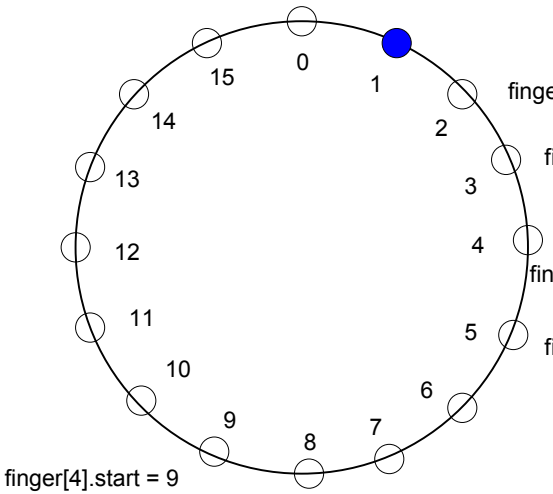
Intervalle

- Suchen von Daten in einem Ring mit nur Vorgänger und Nachfolger ineffizient
  - Im Worst Case halben Ring ablaufen
- ➔ Intervalle
  - Die Anzahl der Einträge hängt direkt von der Größe des Ringes m ab
  - 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.
  - Der erste finger-Eintrag entspricht also dem Nachfolger

Beispiel

m = 4

$(n + 2^{k-1}) \bmod 2^m$, $1 \le k \le m$



Finger-Tabelle

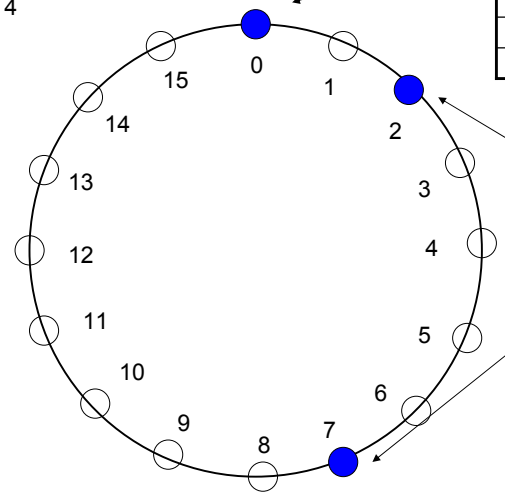
- Diese Tabelle wird zum Suchen der zuständigen Knoten benutzt.
- In jeder Zeile der Finger-Tabelle wird ein Attribut start, interval und node gespeichert.
- Ein Fingereintrag speichert die IP-Adresse und den Chord-identifier des entsprechenden Knotens.

Finger-Tabelle

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

Beispiel

m = 4



start	interval	node
1	[1,2)	2
2	[2,4)	2
4	[4,8)	7
8	[8,0)	0

start	interval	node
3	[3,4)	7
4	[4,6)	7
6	[6,10)	7
10	[10,2)	0

start	interval	node
8	[8,9)	0
9	[9,11)	0
11	[11,15)	0
15	[15,7)	0

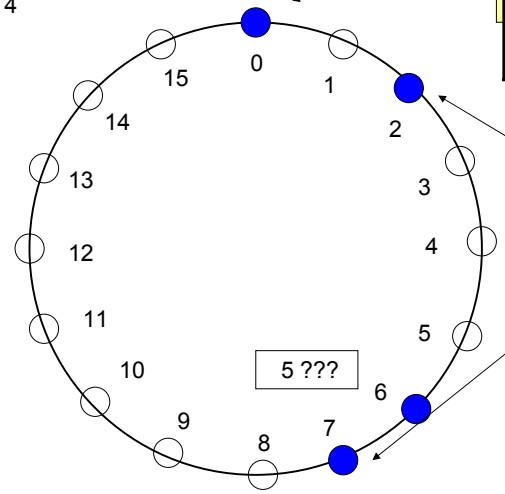
Suche von Daten

- Suche des zu einem Eintrag den verantwortlichen Knoten
  - den successor des Schlüssels finden .
  - Die Suchfunktion des Knoten, an den die Anfrage gestellt wurde, folgt dem Finger, der auf den am weit Entferntesten Knoten zeigt
  - Dieser muss aber vor dem gesuchten Schlüssel liegen
  - Alle Knoten wiederholen dies bis der Schlüssel zwischen dem gefundenen Knoten und seinem successor liegt
  - successor ist dann gleichzeitig der successor des gesuchten Schlüssels.

  ➔ Dies ist der für die Daten zuständige gesuchte Knoten

Beispiel

m = 4



start	interval	node
1	[1,2)	2
2	[2,4)	2
4	[4,8)	6
8	[8,0)	0

start	interval	node
3	[3,4)	6
4	[4,6)	6
6	[6,10)	6
10	[10,2)	0

start	interval	node
8	[8,9)	0
9	[9,11)	0
11	[11,15)	0
15	[15,7)	0

Hinzufügen von Knoten (1)

- Dazu werden drei Schritte für einen Knoten k, der neu in das System eintritt ausgeführt
  - Initialisierung der finger und des predecessors vom Knoten
  - 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.

Hinzufügen von Knoten (2)

Im Detail:

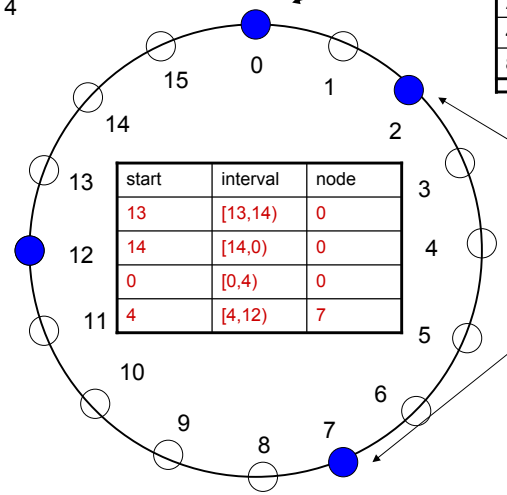
- Wenn ein Knoten k in Chord-System eintreten soll, muss ein Knoten j bekannt sein, der sich schon im Chord-System befindet,
- an dem meldet sich der neue Knoten an
- Durch die Hashfunktion wird für den neuen Knoten von dem System einen Schlüssel im Chord-Kreis berechnet und zugewiesen
- Der neue Knoten 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

Hinzufügen von Knoten (3)

- 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
- er informiert alle anderen Knoten, deren Finger-Knoten er sein könnte, von seiner Existenz, also auch den von predecessor.
- Diese Knoten findet er, in dem er alle verschiedenen Intervallgrößen von sich aus rückwärts geht und von diesen Positionen den predecessor sucht.
- die höher liegende Applikation wird veranlasst sämtliche Daten, für die nun der neue Knoten zuständig ist, vom successor zu verschieben
- Nun ist der neue Knoten einsatzfähig.

Beispiel

m = 4



start	interval	node
13	[13,14)	0
14	[14,0)	0
0	[0,4)	0
4	[4,12)	7

start	interval	node
1	[1,2)	2
2	[2,4)	2
4	[4,8)	7
8	[8,0)	12

start	interval	node
3	[3,4)	7
4	[4,6)	7
6	[6,10)	7
10	[10,2)	0

start	interval	node
8	[8,9)	0
9	[9,11)	0
11	[11,15)	0
15	[15,7)	0

Lastverteilung

- Die Last in dm System wird nur durch die Hashfunktionen verteilt.
- Diese verteilt die Werte so über die Knoten, dass jeder Knoten in etwa die gleiche Anzahl von Elementen zu bearbeiten hat.
- Verhindert nicht, das einzelne Knoten zu stark beansprucht werden

Stabilisierung (1)

- Zusatz zum Protokoll
- Gegen Fehlverhalten, wenn ein Knoten ausfällt oder mehrere Knoten gleichzeitig beitreten
- In gewissen Abständen, die davon Abhängen wie oft Knoten dem System beitreten, überprüft jeder Knoten, ob sein successor noch aktuell ist.
- Dazu überprüft er, ob der predecessor des successor zwischen den beiden Knoten liegt.
- Wenn ja ist dieser der wirkliche successor.

Stabilisierung (2)

- 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.

Entfernen von Knoten

- 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
- Dann abwarten bis das System sich durch seine oben genannten Stabilisierungsroutinen wieder korrigiert.

Laufzeiten

- Jeder Knoten muss O(logN) andere Knoten von insgesamt N kennen.
- Eine Suchoperation braucht O(logN)
- Zum Verlassen oder Hinzufügen von Knoten sind $O(\log^2 N)$ Nachrichten nötig.

Vergleich

Algorithmus	Modell	Suchen	# Nachbarn
Chord	ein-dimensionaler Ring	O(log(N))	O(log(N))
CAN	d-dimensionaler Torus	$dN^{1/d}$	O(d)

Vor / Nachteile von Chord

- Vorteile zu Consistent Hashing
  - Effizienteres Suchen
  - Fehlertoleranter
  - Bei Consistent Hashing muss jeder Knoten Baum speichern
- Nachteile
  - Range Queries nicht effizient möglich
  - Einstiegsknoten muss bekannt sein
  - Wechsel der Hashfunktion nicht möglich
  - Zum Suchen muss kompletter Name etc. bekannt sein

Ende

- Fragen ?