Seminar P2P Systeme
Fachgebiet Verteilte Systeme
Sommersemester 2007

Vortragsausarbeitung

Chord und Varianten

Werner Gaulke
(8. Semester)

Vortrag: 16.07.2007
Sommersemester 2007

Universität Duisburg-Essen · Abteilung für Informatik und angewandte Kognitionswissenschaft
Arbeitsgruppe Verteilte Systeme – Prof.Dr.-Ing. Torben Weis
betreut durch: Mirko Knoll Dipl. Inform.

Inhaltsverzeichnis

1 Einleitung

Mit der Datenstruktur der verteilten Hashtabellen wurde die dritte Generation von P2P Systemen eingeleitet. Knoten und Daten werden dabei auf einen Namensraum abgebildet. Es liegt an den Protokollsystemen, wie dieser Raum effizient angeordnet werden kann.
Chord hat es sich zum Ziel gemacht, diesen Namensraum auf eine einfache und verständliche Struktur abzubilden, auf der alle nötigen Operationen einfach und effizient durchzuführen sind. Eine weitere Besonderheit von Chord ist es, dass Suchergebnisse korrekt aufgelöst werden können. Das richtige Suchergebnis (Fund oder kein Fund) wird also garantiert.
Im Anschluss an den theoretischen Aufbau von Chord wird ein kurzer Blick auf die Praxisperformance geworfen. Im letzten Abschnitt werden einige Chord Varianten und Methoden vorgestellt, die versuchen Chord weiter zu verbessern. Das Basiskonzept der Varianten wird erläutert sowie die daraus resultierenden Vor- und Nachteile vorgestellt.
Ein optionaler Anhang stellt kurz die freie Chord Java Implementierung „Open Chord“ vor, mit der Experimentierfreudige die Theorie lokal ausprobieren können.

Diese Lektüre dient also dazu, dem Leser den Aufbau und die Funktionen des Chord Systems so zu vermitteln, dass er es mit anderen Systemen vergleichen kann.

2 Chord Aufbau

2.1 Verteilte Hash Tabellen

Grundlage für Chord ist eine verteilte Hash Tabelle. Verteilte Hash Tabellen [1] ordnen, wie auch normale Hash Tabellen, Schlüsseln Werte zu. Die Schlüssel einer Hashfunktion werden in einem Schlüsselraum angeordnet. Dieser Schlüsselraum ist groß genug zu wählen, damit keine Kollisionen entstehen [2].
Bei der verteilten Hash Tabelle werden Schlüssel (die auf Daten zeigen), wie auch Knoten (im Netzwerk) in den Schlüsselraum gehasht. Schlüssel können also auf Knoten wie auch auf Daten zeigen. Der Einfachheit halber soll der Begriff Knoten im folgenden den Schlüssel, wie auch den Knoten selbst meinen. Was davon zutrifft bestimmt der Kontext. Der Begriff Schlüssel wird den Daten, die dahinter stehen, zugeordnet.
Jedem Knoten können dabei mehrere Schlüssel zugeordnet werden. Ein Knoten kann also als Behälter für Schlüssel verstanden werden. Beim herkömmlichen Hashing wird, beim hinzukommen neuer Schlüssel, viel umsortiert. Dies ist hier zu vermeiden, da ein umsortieren mit Transfer im Netzwerk verbunden ist. Daher wird bei verteilten Hash-Tabellen konsistentes Hashing verwendet. Beim konsistenten Hashing werden beim Einfügen von neuen Schlüsseln Bewegungen der alten Schlüssel vermieden.
Es sind also zwei Funktionen für Verteilte Hash-Tabellen verfügbar: $put(data)$ welchen einen Key zurückgibt und $get(key)$ welche die Daten zurückgibt.

Zusammenfassend: Verteilte Hash Tabellen ordnen Daten und Knoten in ihrem Adressraum an. Knoten werden Datenschlüsseln zugeordnet. Auf Anfragen nach Schlüsseln werden die dafür zuständigen Knoten zurückgegeben. Aufbauend auf diesem Grundwissen über DHTs wird nun im folgenden Chord vorgestellt.

2.2 Das Chord Projekt

Chord ist keine P2P Implementation, sondern ein Routing Protokoll mit verteilten Hash-Tabellen als Grundlage. Es ist also dafür verantwortlich die Kommunikation der Knoten miteinander zu ermöglichen. Das Chord-Projekt[3] wurde 2001 von Ion Stoica , Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek,

[1] Im folgenden oft DHT (Distributed Hash Table) genannt
[2] Üblich für Chord ist z.B. die SHA-1 Hashfunktion mit 128bit Schlüsseln
[3] http://pdos.csail.mit.edu/chord/

Hari Balakrishnan in [1] vorstellt. Es ist ein Projekt der Abteilung „Parallel & Distributed Operating System Group“[4] des „Massachusetts institute of technology“ [5]. Folgende Punkte charakterisieren[6] das Chord System:

- Ausgeglichen: Dadurch, dass Chord auf einer DHT aufsetzt, werden Schlüssel gleichmäßig über die Knoten verteilt.
- Dezentral: Chord priorisiert keine Netzwerkknoten. Jeder Knoten kann mit jedem Nachrichten austauschen. Dies macht die Knoten gleichwertig und das Netzwerk gleichzeitig robust, da es keine Knoten gibt, die bei Wegfall das Netz zerstören.
- Skalierbar: Chord ordnet die Knoten so an, dass im Logarithmus zur Anzahl der Knoten gesucht werden kann. Dazu verwaltet jeder Knoten selbst log n Schlüssel.
- Verfügbarkeit: Chord verwaltet die internen Knotentabellen automatisch, um das Auffinden eines existierenden Schlüssels garantieren zu können. Egal wie stark das Netz Veränderungen unterworfen ist.
- Flexible Namensgebung: Schlüssel erhalten keine Abhängigkeiten, sodass Anwendungen die Namensgebung flexibel handhaben können.

Zusammen mit diesen Eigenschaften stellt Chord eine Protokollsystem dar, dass die Stärken der DHTs mit effizienten Operationen verknüpft. Im Folgenden wird auf die Struktur des Chord Netzwerkes eingegangen, sowie auf die Realisierung der Operationen.

2.3 Die Chord Struktur

Wie erwähnt verwendet Chord als Grundlage Distributed Hash Tables. Mittels zweier Hashfunktionen[7] werden Knoten wie auch Schlüsseln (welche auf Daten zeigen) einem $m - bit$ Schlüssel zugeordnet. Dieser Schlüssel befindet sich nun in einem $2^m - 1$ großen Schlüsselraum. m ist so zu wählen, dass Kollisionen unwahrscheinlich sind.
Die Besonderheit von Chord ist es nun, dass der Schlüsselraum als Ring angeordnet wird. Dabei gilt der Schlüsselraum also geschlossen, d.h. es wird $mod\ 2^m$ gerechnet. Die gehashten Knoten und Schlüssel werden im Uhrzeigersinn in diesem Ring angeordnet. Dabei wird ein Schlüssel dem Knoten zugeordnet, der sich als nächstes vor ihm befindet.
Damit diese Ringförmig angeordneten Knoten nun kommunizieren können, müssen sie miteinander verbunden sein. Jeder Knoten besitzt einen Zeiger [8], der auf seinen

[4] http://pdos.csail.mit.edu/
[5] http://www.mit.edu/
[6] Frei nach [1]
[7] Chord verwendet überlicherweise SHA-1 mit 128bit Schlüsseln
[8] Zeiger werden in Chord Finger genannt. Dieser Ausdruck wird im Folgenden verwendet

Vorgänger und einen, der auf seinen Nachfolger zeigt. Es bildet sich also ein geschlossener Kreis.
Zwei Zeiger bei einem n Knoten großen Netz sind allerdings zu wenig, um eine schnelle Suche zu garantieren. In diesem Fall hat die Suche ein Laufzeitverhalten von $O(n)$. Damit ein stabiles Netz mit einer schnellen Suche ermöglicht wird, sind weitere Routing Informationen nötig. Dazu speichert jeder Knoten weitere Nachfolger in einer Tabelle, die Fingertabelle.

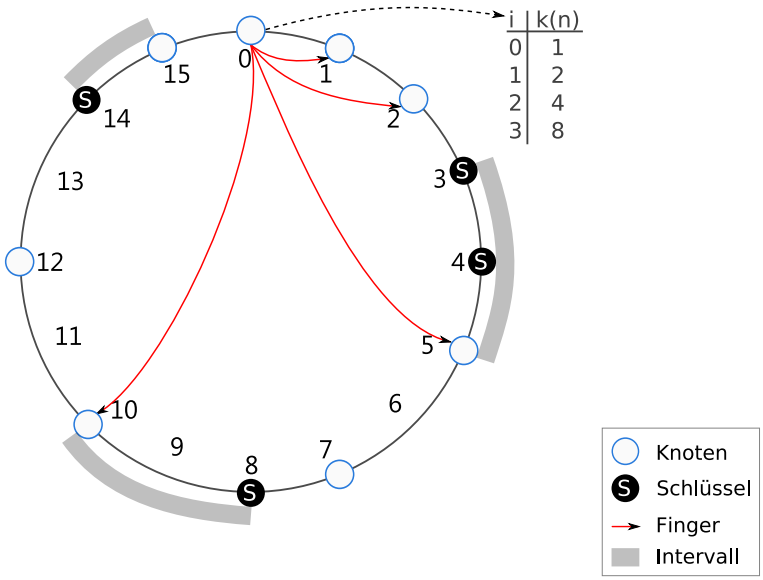

Abbildung 2.1: Chord Ring mit Fingertabelle für Knoten null.

2.3.1 Die Fingertabelle

Jeder Knoten im Chord Netz besitzt eine Fingertabelle mit Routing Informationen über $log\ n$ Nachfolger. Von jedem Knoten sind IP und Port bekannt, um ihn kontaktieren zu können.
Welche Nachfolger in die Tabelle aufgenommen werden, wird über die Berechnung $k(n) = (n + 2^i)\ mod\ 2^m$ ermittelt. Dabei bezeichnet i den i-ten Eintrag in der Tabelle und m die Größe des Hashbereichs und n der Knoten selbst. Das Ergebnis dieser Funktion bezeichnet die Stelle des Folgeknotens im Chord Ring.
Wurde ein Knoten berechnet, wird versucht diesen zu kontaktieren. Bei sehr großen Ringen wird es oft vorkommen, dass an der berechneten Stelle kein Knoten vorhanden ist. In diesem Fall wird der nächste verfügbare Nachfolgerknoten an dieser Stelle in die Fingertabelle aufgenommen. Abbildung 2.1 zeigt einen Chord Ring mit der für Knoten Null ermittelte Fingertabelle.

2.4 Operationen in Chord

Mit der nun vorhandenen Struktur des Chord Netzes können die nötigen Operationen realisiert werden. Da Chord ein reines Routing Protokoll ist, beschränken sich die Operationen auf Grundlegendes:

- Suche von Schlüsseln: Schlüssel müssen im Netz gefunden werden.
- Einfügen von Knoten: Neue Knoten können jederzeit hinzukommen.
- Entfernen von Knoten: Knoten können das Netz jederzeit verlassen.
- Stabilisieren des Netzes: Änderungen am Netz müssen von den Knoten erkannt werden.

2.4.1 Suche

Die Suche nach Schlüssel ist eine primäre Aufgabe in einem P2P System. Ausgehend davon, dass jeder Knoten mit einem Finger auf seinen Nachfolger zeigt, bedeutet dies für die Suche, dass maximal n Knoten für ein korrektes Ergebnis durchlaufen werden müssen. Der Aufwand liegt in diesem Fall bei $O(n)$, was angesichts großer Netzwerke und Signallaufzeiten allerdings immer noch zu langsam ist.
Dadurch, dass jeder Knoten in einem Chord Netz $log\ n$ Einträge in der Fingertabelle hat, kann mit diesem Wissen die Suche wesentlich effizienter durchgeführt werden. Dazu wird bei Suchbeginn genau der Knoten in der Tabelle angesprochen, der sich am dichtesten vor dem zu suchenden Key befindet. Dies ist möglich, da der Schlüsselraum ringförmig, aufsteigend angeordnet ist. Dieser Knoten wird beauftragt, die Anfrage weiter durchzuführen. Dies geschieht so lange, bis der gewünschte Schlüssel gefunden ist.

Daraus folgt: Die Laufzeit der Suche ist mit hoher Wahrscheinlichkeit $O(log\ n)$. Im schlechtesten Fall liegt die Laufzeit bei $O(n)$.

Beweisskizze: Wenn ein Schlüssel gesucht wird, der Nachfolger eines Knotens z ist (also diesem Knoten zugeordnet ist), dann wird vom Ausgangsknoten n zum dichtesten Vorgänger d des Knotens z gesprungen der in n's Fingertabelle vorhanden ist. Der Angesprungene Knoten d ist dabei der i-te Eintrag in der Fingertabelle von n. Sollte sich der Zielknoten z zwischen dem i-ten und $i+1$-ten Fingereintrag von n befinden, dann hat sich die Entfernung zum Zielknoten z halbiert. Mit jedem weiteren Schritt halbiert sich die Distanz, bis z gefunden ist.
Mit hoher Wahrscheinlichkeit sind daher $O(log\ n)$ Schritte bis zum Erreichen von z nötig. Im Schlimmsten Fall muss der ganze Ring durchwandert werden, was $O(n)$ Schritte sind.

2.4.2 Einfügen von Knoten

In P2P Netzen können jederzeit neue Knoten hinzukommen. Es wird ein Mechanismus benötigt, der das Hinzufügen von Knoten performant realisiert. Während sich der neue Knoten im Netz einordnen, müssen die alten Knoten Fingertabellen aktualisieren. Auch die Schlüssel, für die der neue Knoten verantwortlich ist, müssen diesem Übertragen werden. Es muss beachtet werden, dass der Ring dabei nicht gespalten wird oder Schlüssel verloren gehen.

Vorraussetzung für das Einfügen eines Knotens ist es, dass n einen bereits im Netz befindlichen Knoten e findet. Chord selbst bietet keinen Mechanismus um einen Knoten e zu finden. Diese Aufgabe muss die auf Chord aufbauende Applikation realisieren [9].
Wenn ein Knoten bekannt ist, werden beim einfügen folgende drei Schritte ausgeführt:

- *Initialisieren*:
  Von unmittelbaren Nachfolgerknoten p übernimmt n den Vorgänger. Der Nachfolgerknoten übernimmt den neuen Knoten als seinen Vorgänger. Der Nachfolgerknoten wird nun benutzt um die Fingertabelle von n zu füllen. Dazu kann nun für jeden Tabelleneintrag beim Nachfolger jedes Mal $find\_successor$ aufgerufen werden, eine Funktion, die den unmittelbaren Nachfolger für einen Knoten ausgibt. Um den Vorgang zu beschleunigen testet der neue Knoten aber jedesmal selbst, ob der $i-te$ Finger gleichzeitig auch der Richtige $(i+1)-te$ Finger ist. Bei großen Wertebereichen trifft dies mit hoher Wahrscheinlichkeit zu, wodurch sich eine Suchzeit von $O(log\ n)$ ergibt. Um also die Fingertabelle mit $log\ n$ Einträgen zu füllen, müssen diese gesucht werden, jeweils mit einem Aufwand von $O(log\ n)$, was insgesamt $O(log^2\ n)$ Nachrichten ergibt.

- *Updaten:*
  Nach dem Einordnen des neuen Knotens, muss dieser von anderen Knoten in die Finger Tabelle aufgenommen werden. Ein Knoten p wird n in seine Fingertabelle genau dann aufnehmen, wenn p mindestens 2^i-1 vor n ist und der $i-te$ Finger von p größer ist als n. Das Aktualisieren der Fingertabellen geschieht automatisch durch das Aufrufen der Stabilisieren Funktion.

- *Schlüssel Transferieren:*
  Dem neuen Knoten n werden alle Schlüssel zugewiesen, für die er zuständig ist. Dazu zählen alle Schlüssel, für die n der Nachfolger ist. Dazu muss n seinen direkten Nachfolger verständigen, der vorher für diese Schlüssel zuständig war, damit diese zu ihm übertragen[10] werden. Die Schlüssel sind solange auf beiden Knoten vorhanden, bis der Vorgänger von n ihn als seinen Nachfolger übernommen hat. Dies geschieht, sobald bei diesem die Stabilisieren Funktion aufgerufen wird.

[9] Z.B. über eine mitgelieferter Liste wahrscheinlich aktiver Knoten oder andere Mechanismen
[10] Das übertragen der Schlüssel selbst, ist Teil der Software, die Chord einsetzt.

Wichtig für die Einfüge-Operation ist, dass laufende Suchanfragen nicht unterbrochen werden.

Beweisskizze: Das Einfügen eines neuen Knotens n unterbricht den Ring nicht, da er im Uhrzeigersinn läuft. Das einfügen des neuen Knotens verursacht nur, dass dessen Nachfolger ihn als Vorgänger aufnimmt. Der Vorgänger des schon vorher vorhandenen Knotens zeigt weiterhin auf diesen und übernimmt den neuen Knoten erst, wenn die Stabilisierungs- Operation aufgerufen wird. So werden Suchanfragen auch während des Einfügens an das richtige Ziel geleitet.

2.4.3 Entfernen von Knoten

Beim entfernen von Knoten muss darauf geachtet werden, dass keine Keys verloren gehen und das Netzwerk stabil bleibt. Fällt ein Knoten n weg, müssen alle Knoten, die n in ihrer Fingertabelle haben die Fingertabelle auf den Nachfolger von n aktualisieren.
Um diesen Prozess zu vereinfachen, besitzt jeder Knoten zusätzlich eine Liste mit den r nächsten Nachfolgern. Sobald ein Knoten bemerkt, dass sein Nachfolger ausgefallen ist, wird er anhand dieser Liste den nächsten aktiven Knoten als seinen neuen Nachfolger auswählen.
Die zuvor beschriebenen Stabilisierungsfunktionen sorgen automatisch dafür, dass nicht gültige Netzzustände behoben werden. Suchanfragen, die während der Stabilisierungsphase einen Knoten erreichen, werden dabei nicht verworfen. Sobald ein Knoten wieder den einen aktiven Nachfolger kennt, wird die Anfrage weiter aufgelöst.
Der Aufwand bei einer $log\ n$ großen Nachfolgerliste um einen Nachfolger zu finden liegt bei $O(log\ n)$ wenn jeder Knoten der Liste mit einer Wahrscheinlichkeit von $1/2$ noch aktiv ist.

Beweisskizze: Die $log\ n$ große Nachfolgerliste wird mit hoher Wahrscheinlichkeit mindestens einen aktiven Nachfolger beinhalten.

2.4.4 Stabilisierung des Netzes

Als aktives P2P Netz ist der Chord Ring ständigen Änderungen unterlegen. Jederzeit können mehrere Knoten das Netzwerk betreten oder verlassen. Durch diese Zustandswechsel wird es vorkommen, dass Knoten Einträge in Fingertabellen haben, die auf nicht mehr vorhandene Knoten oder nicht gültige Nachfolger/Vorgänger zeigen.
Es ist ein Mechanismus notwendig, der das Netz stabilisiert und aktualisiert. Die Stabilisierung sorgt für die Erreichbarkeit existierender Knoten, sie kann allerdings nicht das Spalten des Netzes verhindern, welches z.B. böswillig herbeigeführt werden kann.
In [2] sind zur Stabilisierung drei Funktionen angegeben. Die Funktionen $stabilize$ und $check\_predessor$ sorgen für aktuelle Vorgänger und Nachfolger und $fix\_fingers$ für gültige Fingertabellen Einträge. Die Funktionen werden von den Knoten immer

wieder selbstständig ausgeführt.
Bei $check\_predessor$ wird geprüft, ob der Vorgänger noch vorhanden ist. Falls der Vorgänger fehlt, wird der Zeiger auf $null$ gesetzt. Der Zeiger wird wieder belegt, wenn ein Vorgängerknoten die $stabilize$ Funktion ausführt.
Die Funktion $fix\_fingers$ prüft die Knoten der Fingertabelle und findet bei nicht aktiven Fingerknoten geeignete Nachfolger.
Die $stabilize$ Funktion führt bei Aufruf folgende zwei Schritte durch:

1. Der Knoten n fragt seinen Nachfolger s nach seinem Vorgänger p.

2. Wenn nun p gleich n ist, ist der korrekte Nachfolger von n verifiziert, andernfalls handelt es sich bei p um einen neuen Knoten, der fortan Nachfolger von n wird.

Mit diesen Funktionen wird das Chord-Netz aktuell gehalten. Selbst wenn mehrere Knoten auf einmal hinzukommen oder ausfallen sorgen diese Funktionen dafür, dass sich das Netz selbst aufrecht hält. Der Aufwand für die Stabilisierung ist ebenfalls $O(log\ n)$, da für das aktualisieren eines Tabelleneintrags im Wesentlichen ein Knoten gesucht werden muss.

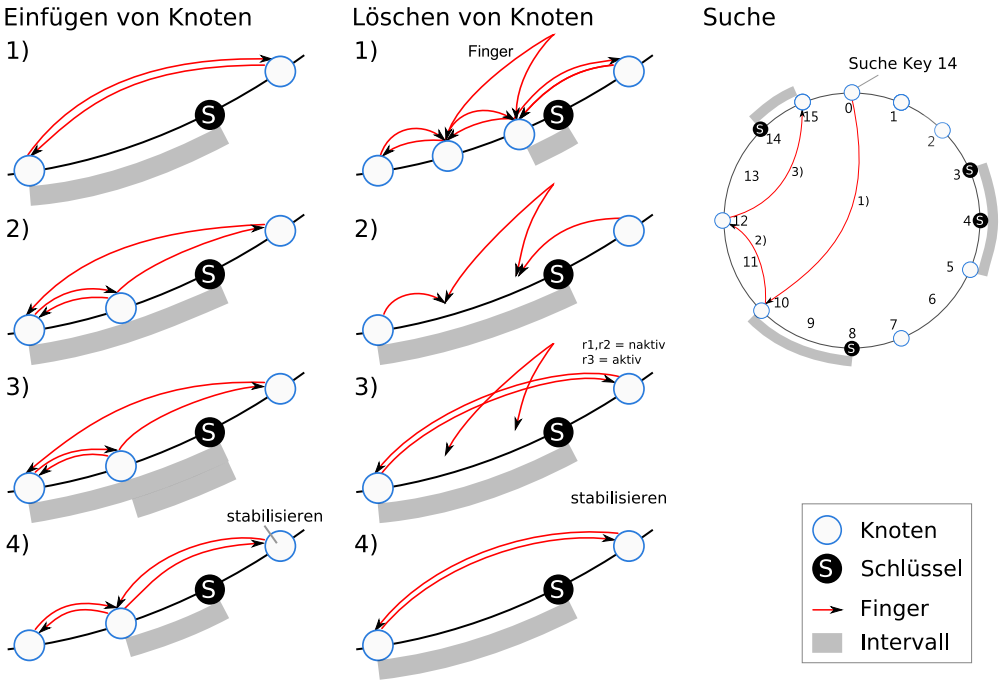

Abbildung 2.2: Einfügen, Löschen und Suchen.

3 Chord in der Praxis

3.1 Performance

Um über die Güte von Chord urteilen zu können, sowie um Stärken und Schwächen aufzudecken, ist ein Test unter realistischen Bedingungen nötig. In [1] finden zwei Analysen statt: Zum einen ein Test in einer simulierten Umgebung und zum anderen ein Test in einer realen Netzwerkumgebung. An dieser Stelle soll auf den Test in realer Umgebung eingegangen werden. Als Parameter für den Test wurden 160Bit Keys zusammen mit dem SHA-1 Verfahren verwendet. Die Kommunikation der Knoten findet über TCP statt. Chord wird hierbei in einem verteilten Dateisystem verwendet. Die Knoten sind über die USA verteilt und haben im Mittel eine Latenzzeit von 60ms.
Es werden die Latenzzeiten bei der Suche über mehrere Knoten ermittelt. Das Experiment zeigt, dass selbst bei steigender Knotenzahl die Latenzzahlen gleichmäßig mitwachsen. Für einen Lookup über 180 Knoten ergibt sich eine durchschnittliche Latenzzeit von 300ms. Abbildung 3.1 gibt Aufschluss über den Mittelwert sowie Minimal- und Maximalwerte. Dieser kleine Praxistest zeigt, dass Chord in realen Umgebung Le-

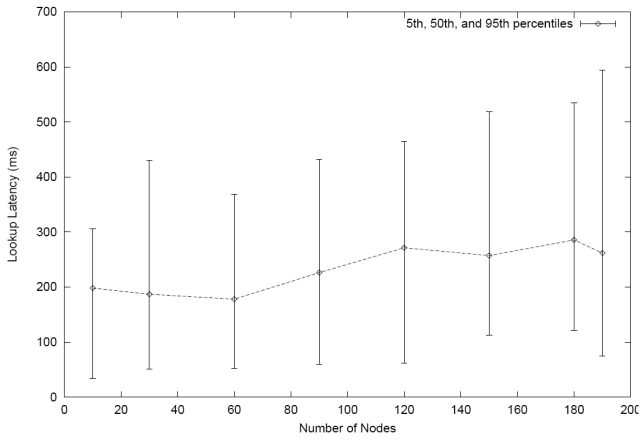

Abbildung 3.1: Entwicklung der Latenzen aus [1]

bensfähig ist und bei großen Netzen skaliert. Doch auch wenn Chord gut skaliert kann für die Praxis eine Suchanfrage mit der Komplexität $O(log\ n)$ zu lang sein. Doch es gibt Varianten und Verbesserungen für Chord, die sich u.a. diesem Problem annehmen.

3.2 Verbesserungen für Chord

Obwohl Chord in seiner Basisversion durchaus performant arbeitet, kann dies durch Modifikationen noch verbessert werden. Doch werden Vorteile meist mit Einbußen in anderen Bereichen erkauft. So bestimmt der Anwendungskontext, welche Modifikation Sinn ergibt. Im Folgenden werden einige Modifikationen und Varianten vorgestellt.

3.2.1 Kleine Modifikationen

- Priorisierung nach Latenz
  Um hohe Anfragelatenzen zu vermeiden, können Knoten zusätzlich nach ihrer Latenz geordnet werden. Knoten mit niedriger Latenz werden bevorzugt.
  Vorteil: Geringere Gesamtlatenzen bei Anfragen.
  Nachteil: Aktualisierungsaufwand, Priorisierung von Knoten.

- Zerteilte Ringe
  Durch Ausfall von Knoten oder Angreifer kann es passieren, dass der Ring zerteilt wird. Dies kann verhindert werden, wenn sich jeder Knoten eine Anzahl zufälliger Knoten merkt und diese hin und wieder kontaktiert. Abgesplitterte Knoten können so wieder Kontakt mit dem Ring aufnehmen.
  Vorteil: Schutz gegen Zersplitterung.
  Nachteil: Weitere Knoten, die kontaktiert werden müssen.

- Absicherung gegen Angreifer
  Angreifer können versuchen bestimmte Schlüssel aus dem Ring auszuschließen, indem sie sich in deren Nähe hashen und Anfragen blockieren. Durch festes Binden der IP an den Schlüssel wird das gezielte Einordnen verhindert.
  Vorteil: Störungen durch Angreifer vermeiden.
  Nachteil: IP mit Schlüssel zu verbinden kann umgangen werden.

3.2.2 EPI Chord - Paralelle Suche

Die EPI Chord [3] genannte Variante optimiert das Chord System auf mehrere Arten. Die Fingertabellen fallen weg und werden durch Caches ersetzt, in denen mehrere Vorgänger und Nachfolger eingetragen werden. Die Caches werden dabei durch Anfragen, die einen Knoten treffen, aktuell gehalten. Bei zu wenig Anfragen muss der Cache manuell aktualisiert werden. Neue Knoten erhalten die Caches benachbarter Knoten.
Suchanfragen werden parallel durchgeführt indem p Anfragen an Knoten gesendet werden, die sich in der Nähe des Zielschlüssels befinden. Diese Anfrage liefert die nächsten besten[1] Knoten für die weitere Suche zurück. Ein Suchergebnis ist dann nach n Sprüngen erreicht. Der Suchaufwand beläuft sich in EPI Chord im durchschnittlichen Fall auf $2(p+n)$ Nachrichten. Im schlechtesten Fall beläuft sich der Aufwand auf $log\ n$

[1] Die Knoten könnten z.B. nach Latenz geordnet sein

Nachrichten. Dies stellt eine wesentliche Verbesserung gegenüber den Aufwand vom normalen Chord Suchverfahren dar.
Vorteil: Starke Verbesserung der Suche.
Nachteil: Bei starker Parallelisierung Performance Verluste (zu viele Nachrichten).

3.2.3 Verwenden von Erasure Codes

In [4] wird eine Chord Variante vorgestellt, dessen Einsatzzweck ein Backup-System darstellt. Chord wird auf mehrere Arten modifiziert um die Verfügbarkeit und Erreichbarkeit von Daten zu erhöhen.
Anstelle von verteilten Hashen für das Speichern von Daten werden hier „erasure Codes“ verwendet. Ein Datenblock wird hierbei in k Teile zerteilt, von denen eine Untermenge m ausreicht um den Block zu rekonstruieren. Dies erhöht die Ausfallsicherheit wenn Knoten oder Daten aus den Netz wegfallen.
Um die Erreichbarkeit zu erhöhen, werden Knoten dreidimensional Angeordnet um Distanzen eintragen zu können und so lokale Erreichbarkeit zu beschleunigen.
Vorteil: Hohe Datensicherheit bei Knotenwegfällen.
Nachteil: Größerer Overhead und komplexes Messen der Distanz.

3.2.4 B-Chord

B-Chord [5] modifiziert die Fingertabellen und die Suche. Die Fingertabelle nimmt in dieser Variante Vorgänger- als auch Nachfolgerknoten auf. Die Suche profitiert davon, indem sie zu einem Zielknoten den günstigsten Vorgänger und Nachfolger in der Fingertabelle auswählt und von diesen Alternativen den näheren Knoten von beiden wählt. Günstig kann z.B. näher am Zielknoten oder geringere Latenz bedeuten.
Der Suchaufwand wird dadurch zwar nicht verbessert (er bleibt bei $O(log\ n)$), aber durch die Wahlmöglichkeit werden im realen Umfeld oft geringere Latenzen erreicht.
Vorteil: Geringere Suchlatenzen.
Nachteil: Höherer Aufwand die Fingertabelle zu aktualisieren.

3.2.5 S-Chord

S-Chord [6] erweitert Chord um Symmetrie, was das beidseitige (bidirektionale) Wandern durch den Chord Ring ermöglicht. Die Fingertabelle enthält dazu für jede Zeile zwei Einträge. Ein Eintrag auf einen Knoten im Uhrzeigersinn, ein Eintrag auf einen Knoten gegen den Uhrzeigersinn. Beide Knoten werden durch eine Berechnungsvorschrift ermittelt. Bei Suchoperationen wird aus der Tabelle der Knoten ausgewählt, der näher an dem Zielknoten liegt. Getestet mit äquivalent gleich vielen Fingern benötigt diese Variante 25% weniger Sprünge um einen Zielknoten zu erreichen.
Vorteil: Beschleunigte Suche.
Nachteil: Bei äquivalent gleich vielen Fingern wird weniger Tiefe im Ring erreicht.

4 Fazit

Im Rahmen dieser Ausarbeitung wurde mit Chord ein einfach strukturiertes, effizientes und skalierbares Routingsystem für verteilte Hash-Tabellen vorgestellt. Chords einfacher ringförmiger Aufbau erlaubt ein schnelles Nachvollziehen, wie die Operationen darauf ausgeführt werden. Ebenso zeigt Chord, dass ein einfacher Aufbau komplexeren Lösungen nicht zwangsläufig unterlegen ist. Die Betrachtung der Operationen, insbesondere die der Suche, hat gezeigt, dass auf dieser einfachen Struktur effiziente Verfahren möglich sind. Neben der Laufzeit von $O(log\ n)$ liefer die Suche korrekte Suchergebnisse. Schlüssel, die vorhanden sind, werden also in jedem Fall gefunden.
Da sich Chord auf das Routing beschränkt, ist eine Anwendung für die Realisation der weiteren Funktionen verantwortlich. Diese Kapselung der Basisfunktionen ermöglicht aber gleichzeitig den anwendungsunabhängigen Einsatz von Chord. Es kann also flexibel in einer Vielzahl von Szenarien verwendet werden.
Die Praxistests haben gezeigt, dass Chord auch unmodifiziert in realen Umgebungen eingesetzt werden kann. Dass Verbesserungspotenzial vorhanden ist, zeigen die vorgestellten Modifikationen und Varianten. Nicht jede Änderung bringt automatisch nur Vorteilte mit sich. Es gilt vielmehr abzuwägen, in welchem Einsatzumfeld eine Variante mit ihren Änderungen sinnvoll verwendet werden kann.

Mit dem in dieser Ausarbeitung vermittelten Wissen über Chord, soll es nun möglich sein Chord in Bezug auf Aufbau und Leistung mit anderen P2P Systemen zu vergleichen um Vor- und Nachteile der einzelnen Systeme zu erkennen. Je nach Anwendung soll so das passende System gefunden werden können.

5 Anhang: Open Chord

Open Chord [5] ist eine Open Source Implementation von Chord in der Sprache Java. Mit dieser Implementation ist es möglich Chord in einer eigenen Applikation als eigene Schicht einzubinden, wie es von den Chord Autoren selbst in [8] vorgeschlagen wird. Open Chord stellt in seinem *service* Paket ein Interface bereit, mit dessen Funktionen Chord verwendet werden kann. Kommunikation der Knoten kann lokal, über Java Sockets oder eigene Erweiterungen geschehen.
Lokal bietet open Chord eine Konsole an, mit der ein Chord Ring erzeugt und verändert werden kann. Um die Open Chord Konsole lokal auszuführen sind folgende Schritte notwendig:

1. Dowload von Open Chord
2. Den Open Chord *src* Ordner in ein Eclipse Java Projekt importieren. [1]
3. log4j.jar den Klassenbibliotheken hinzufügen
4. Die Main.java im Package *console* ausführen um die Anwendung zu starten

Alternativ kann die Konsole auch über die Kommandozeile gestartet werden. Siehe dazu das Open Chord Manual.
Folgendes Listing zeigt wie ein kleines lokales Chord Netz erzeugt wird.

Listing 5.1: Erzeugen eines kleinen Chord Netzes

```
1 create -names node0 \\ erzeugt neues chord netz
2 create -names node1_node2_node3_node4_node5 -bootstraps node0 \\fügt
    neue Knoten ein
3 show \\zeigt nun die Knoten im Ring an
4 insert -node node1 -key aNewKey -value valueOfKey \\einfügen eines
   Schlüssels in node1
5 entries \\zeigt die Knoten und ihre Schlüssel an
```

Mit der Eingabe von *help* werden alle Kommandos aufgelistet. Mit diesen kann nun experimentiert und getestet werden, wie sich Open Chord bei Änderungen verhält.

[1] Bzw. die Entwicklungsumgebung der Wahl

6 Literaturverzeichnis

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
„Chord: A Scalable Peertopeer Lookup Service for Internet Applications 2005“ (ACM SIGCOMM 2001, San Deigo, CA, August 2001)
http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

[2] Ion Stoica , Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan
„Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications“ (IEEE/ACM Transactions on Networking)
http://pdos.csail.mit.edu/chord/papers/paper-ton.pdf

[3] Ben Leong, Barbara Liskov, and Eric D. Demaine
„EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management“ (MIT Technical Report MIT-LCS-TR-963, August 2004)
http://project-iris.net/irisbib/papers/epichord:techreport/paper.pdf

[4] Emil Sit, Josh Cates, and Russ Cox
„A DHT-based Backup System“ (MIT Laboratory for Computer Science, 10 August 2003)
http://project-iris.net/isw-2003/papers/sit.pdf

[5] Jie Wang, Zhijun Yu
„A New Variation of Chord with Novel Improvement on Lookup Locality“ (Department of Computer Science University of Massachusetts Lowell)
http://www.cs.uml.edu/ wang/WangYu.pdf

[6] Valentin A. Mesaros, Bruno Carton, Peter Van Roy
S-Chord: Using Symmetry to Improve Lookup Efficiency in Chord (2002)
http://citeseer.ist.psu.edu/mesaros02schord.html

[7] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger Robert Morris, Ion Stoica, Hari Balakrishnan
„Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service“ (MIT Laboratory for Computer Science TODO JAHR)
http://pdos.csail.mit.edu/papers/chord:hotos01/hotos8.pdf

[8] Open Chord
Open Source Chord Implementation des Lehrstuhls für Praktische Informatik der Universität
http://www.lspi.wiai.uni-bamberg.de/dmsg/software/open_chord/