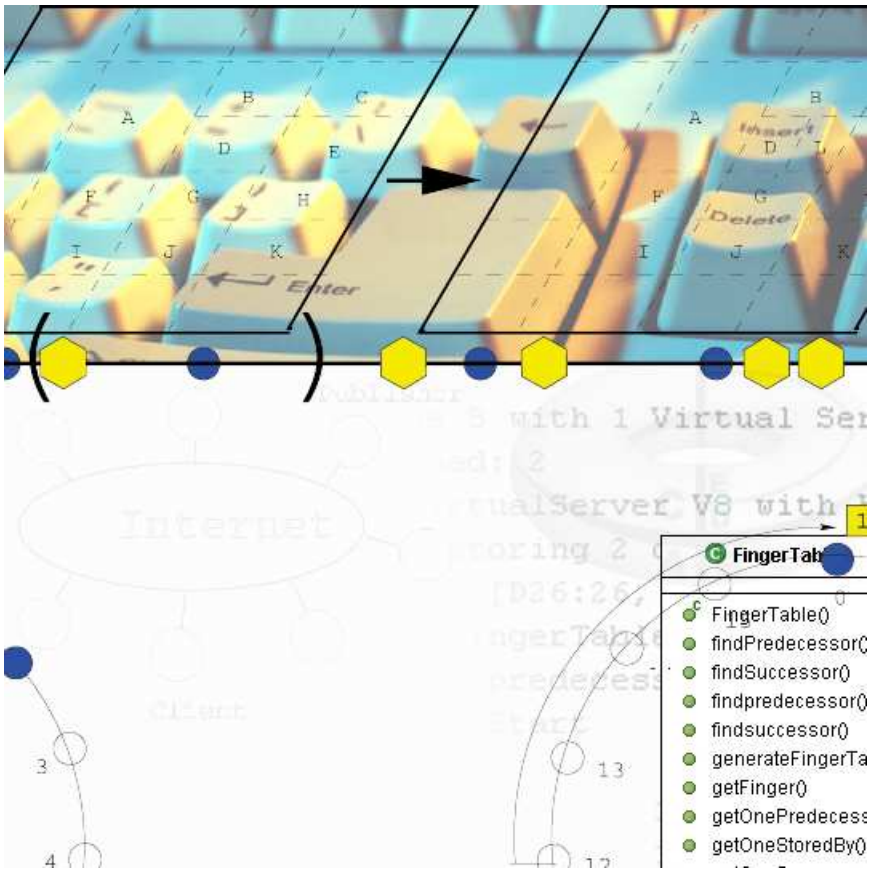
Simon Rieche

Lastbalancierung in Peer-to-Peer Systemen

Diplomarbeit



30. November 2003

Das Titelbild ist größtenteils aus Grafiken,
die für diese Arbeit entstanden sind,
zusammengesetzt.

Freie Universität Berlin
Fachbereich Mathematik und Informatik
Institut für Informatik

Lastbalancierung in Peer-to-Peer Systemen

Diplomarbeit

30. November 2003

Simon Rieche
rieche@inf.fu-berlin.de
Matrikelnummer: 348 09 47

Betreuer

Prof. Dr. Alexander Reinefeld,
Humboldt Universität zu Berlin,
Konrad-Zuse-Zentrum für
Informationstechnik Berlin

Prof. Dr. Jochen H. Schiller,
Freie Universität Berlin

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt zu haben.

Berlin, den 30. November 2003

Simon Rieche

Begegnet uns jemand, der uns Dank schuldig ist, gleich fällt es uns ein. Wie oft können wir jemand begegnen, dem wir Dank schuldig sind, ohne daran zu denken.

– Johann Wolfgang Goethe in „Wahlverwandtschaften“ (1809)

Danksagung

**Als erstes gilt mein Dank meinen Eltern,
die mich während meiner Diplomarbeit und
meines gesamten Studiums
unterstützt haben.**

**Weiterhin danke ich meinem Betreuer
Herrn Prof. Dr. Alexander Reinefeld,
Leiter des Bereichs Computer Science am
Konrad-Zuse-Zentrum für Informationstechnik Berlin,
der es mir ermöglicht hat diese Diplomarbeit
am ZIB zu schreiben.**

**Außerdem danke ich meinem zweiten Bertreuer
Herrn Prof. Dr. Jochen H. Schiller
vom Institut für Informatik,
Arbeitsgruppe Computer Systems & Telematics an der
Freien Universität Berlin,
der diese Arbeit an der FU Berlin betreut hat.**

**Schließlich danke ich besonders meinem beiden
direkten Betreuern
Herrn Dipl.-Inform. Florian Schintke
und
Herrn Dipl.-Inform. Thorsten Schütt
vom Konrad-Zuse-Zentrum für Informationstechnik Berlin
für ihre Unterstützung bei der
Entstehung dieser Diplomarbeit.**

Simon Rieche

Kurzfassung

Peer-to-Peer (P2P) Systeme werden u.a. zur Verwaltung von großen Datenmengen benutzt, die auf verschiedene Rechner verteilt sind. Benutzern soll damit der Zugriff auf Daten innerhalb des Systems leicht ermöglicht werden. Damit in P2P Systemen die Daten effizient verteilt und gesucht werden können, existieren Distributed Hash Tables. Distributed Hash Tables (DHT) sind eine Methode, um globale Informationen persistent speichern zu können. Der Wertebereich der Hashfunktion, welche die zu veröffentlichenden Einträge auf Werte abbildet, wird in Abschnitte aufgeteilt, die einzelnen Knoten zugeteilt werden. Die meisten DHTs haben aber ein Problem bei der Verteilung der Last. Die verschiedenen DHT Systeme beruhen meist auf einem identischen Ansatz der Lastverteilung. Die Last wird nur mit Hilfe einer Hashfunktion verteilt. Es wird davon ausgegangen, dass diese Funktion die Last gleichmäßig verteilt. Im Rahmen dieser Arbeit wurde ein Verfahren zur Lastverteilung entwickelt, simuliert und implementiert. Bei dem Verfahren wird die Last wie bei der Verteilung von Wärme an die Umgebung abgegeben. Es wird mit existierenden Lastbalancierungsalgorithmen verglichen. Mit diesem neuen Verfahren ist es möglich, Last in DHTs besser zu verteilen ohne große Änderungen an den DHTs vorzunehmen. Es wird gezeigt, wie mit dem Verfahren zusätzlich die Fehlertoleranz des P2P Systems erhöht werden kann.

Abstract

One of the many uses of Peer-to-Peer (P2P) systems is the administration of large data sets that are distributed across different computers, with the goal of facilitating user access to files within the system. Distributed Hash Tables (DHT) are designed to enable the efficient distribution and search of files, by allowing global information to be persistently stored. The range of values of the hash function (the possible entries in the published hash table) are assigned to individual nodes. Most DHTs, however, have a problem with load distribution. The various DHT systems usually operate by distributing load equally among nodes. Thus the load is distributed using the help of the hash function. One assumes this function distributes the load evenly. In the context of this work a method of distributing load has been developed, simulated and implemented. With this method load is transferred in a fashion analogous to the dissipation of heat into the environment. Comparisons with existing algorithms for load balancing are drawn. With the new procedure it is possible to better distribute load in DHTs without requiring major changes to the DHTs themselves. It is shown that with the procedure the fault tolerance of P2P systems may also be increased.

The number of UNIX installations has grown
to 10, with more expected.

– Unix Programmer's Manual, 2nd Edition (1972)

Inhaltsverzeichnis

Ich habe keine besondere Begabung, sondern
bin nur leidenschaftlich neugierig.

– Albert Einstein (1952)

Kapitel 1

Einleitung

1.1 Thema dieser Arbeit

Peer-to-Peer (P2P) [31] Systeme werden unter anderem zur Verwaltung von großen Datenmengen benutzt, die auf verschiedene Rechner verteilt sind. Diese Diplomarbeit beschäftigt sich mit der Lastverteilung von Daten zwischen verschiedenen Rechnern in einem solchen Netz. Dabei soll die Last effizient zwischen den beteiligten Rechner im P2P Netz verteilt werden. Ein Peer-to-Peer System ist eine Art eines verteilten Systems, zu dem zum Beispiel auch die klassische Client-Server Architektur zählt.

Das Hauptmerkmal in einem Peer-to-Peer (P2P) System ist, dass alle Komponenten gleich sind. Im P2P Netz existieren keine ausgezeichneten Rechner, wie zum Beispiel in der Client-Server-Architektur. Dadurch soll die Ausfallsicherheit und Skalierbarkeit des Gesamtsystems gegenüber anderen verteilten Systemen, wie zum Beispiel der erwähnten Client-Server-Architektur, verbessert werden.

Ein weiteres Merkmal eines P2P Systems ist die eingeschränkte Nachbarschaft. Die Rechner kommunizieren nur mit einigen wenigen, ihnen bekannten, Rechnern. Diese Nachbarschaft enthält nicht notwendigerweise sämtliche Rechner im Netz. Üblicherweise hat ein Knoten in einem System mit N Knoten $O(1)$ oder $O(\log(N))$ Nachbarn. Da in P2P Systemen die Rechner also nur Informationen über ihre Nachbarschaft besitzen, müssen sie ihre Arbeit unter Umständen mit unvollständigen Informationen verrichten. In einem P2P Netz existiert keine globale Sicht.

Distributed Hash Tables (DHT) sind eine Methode, um globale Informationen persistent in einem P2P System speichern zu können. Eine DHT bildet eine über mehrere Rechner verteilte Hashtabelle. Der Wertebereich der Hashfunktion, welche die zu veröffentlichenden Einträge auf Werte abbildet, wird in Abschnitte aufgeteilt, die einzelnen Knoten zugeteilt werden. Jeder Knoten ist dann dafür zuständig, die Anfragen zu bearbeiten, die in seinen Wertebereich fallen [7].

Verschiedene DHTs werden in dieser Arbeit nach verschiedenen Kriterien [31, 3, 14, 15] bewertet, wie die Effizienz des Suchalgorithmus oder die Ausdrucksstärke bei der Suche.

Zur Suche von Einträgen in einem Peer-to-Peer System existieren verschiedene Strategien. Eine ineffiziente Lösung ist das Senden der Anfrage an alle Rechner im Netz (Flooding). Durch diese Vorgehensweise skalieren die Systeme nicht, da das Netz mit Anfragen überlastet wird. Effizientere Systeme sind zum Beispiel die DHTs CAN (Content-Addressable Network) [44] oder Chord [57], die die Schlüssel nach einem geeigneten Schema über die Knoten verteilen. Da die Verteilung der Schlüssel bekannt ist, kann die Suche effizienter implementiert und mit weniger Netzlast der zuständige Knoten gefunden werden.

Ein weiteres Unterscheidungsmerkmal von DHTs ist die Ausdrucksstärke der unterstützten Anfragen. So können die DHTs entweder nur einfache Abfragen auf Identität erlauben, oder sie haben eine Unterstützung von komplexen Bereichsabfragen (zum Beispiel alle Elemente, die mit a Anfangen) oder regulären Ausdrücken (zum Beispiel find [abc]*grid*.pdf).

Ein weiteres Problem in DHTs ist die Lastverteilung. Die verschiedenen DHT Systeme beruhen meist auf einem identischen Ansatz der Lastverteilung. So wird die Last nur mit Hilfe einer Hashfunktion verteilt. Es wird davon ausgegangen, dass diese Funktion die Last gleichmäßig verteilt. Dabei sind unter dem allgemeinen Begriff Last verschiedene Parameter zusammengefasst. Es kann sowohl die Prozessorlast, die durch Anfragen innerhalb eines Zeitraums entsteht, als auch der benötigte Speicherbedarf durch die Daten optimiert werden.

In dieser Arbeit wird deshalb die Lastbalancierung in DHT Systemen untersucht. Da im Peer-to-Peer Ansatz jedoch keine globale Sicht existiert, müssen lokale Strategien zum Verteilen der Last entwickelt werden.

Ein Ansatz ist die Replikation. Die Daten werden per Kopie an andere Knoten gesandt. Daraufhin können Anfragen von mehreren Knoten bearbeitet werden. Dem Gewinn, dass die Last besser auf die einzelnen Knoten verteilt und das System ausfallsicherer ist, steht aber der Aufwand der Replikation entgegen [27]. Beim Replizieren der Daten muss also darauf geachtet werden, dass Daten nicht zu oft kopiert oder verschoben werden, da dies zu einer hohen Last in den Knoten und im Netz führen kann. Ein geschicktes Einsetzen der Replikation in anderen Lastbalancierungsalgorithmen kann aber zu einer Erhöhung der Fehlertoleranz des gesamten P2P Systems führen.

Die Lastverteilung kann dabei also die Ausfallsicherheit unterstützen. Wenn ein DHT System zusätzlich zur Lastbalancierung noch ausfallsicher sein soll, sind beim Verteilen der Last mögliche Vorgaben wie das Garantieren von Verfügbarkeit zu beachten, d.h. es darf keine Situation enstehen, in der ein Bereich von keinem oder von zu wenigen Knoten bearbeitet wird, um immer gewährleisten zu können, dass die gesuchten Daten gefunden werden können. Also müssen bei der Verschiebung der einzelnen Bereiche beim Verteilen der Last immer noch mehrere Knoten für einen Bereich zuständig sein.

Eine mögliche Lösung wäre, die Last ähnlich wie eine Wärmeausbreitung in einem Material zu verteilen, d.h. jeder Knoten gibt seine Belastung an die umgebenden Knoten weiter, bis ein gleichverteilte Last im System entsteht.

1.2 Ziele dieser Arbeit

In dieser Arbeit sollen bereits existierende Systeme zur Lastbalancierung und zum Routing vorgestellt und verglichen werden. Es wird ein weiterer Algorithmus zur Lastverteilung selber entwickelt. Diese Systeme sollen dann mit Hilfe von Simulationen verglichen werden. Der neue Lastbalancierungsalgorithmus soll als Client implementiert und in der Praxis beobachtet werden. Daraufhin sind Messungen an dem implementierten System bezüglich der Lastverteilung, etc. durchzuführen.

1.3 Struktur dieser Arbeit

Die folgende Abbildung 1.1 zeigt den groben Aufbau dieser Arbeit.

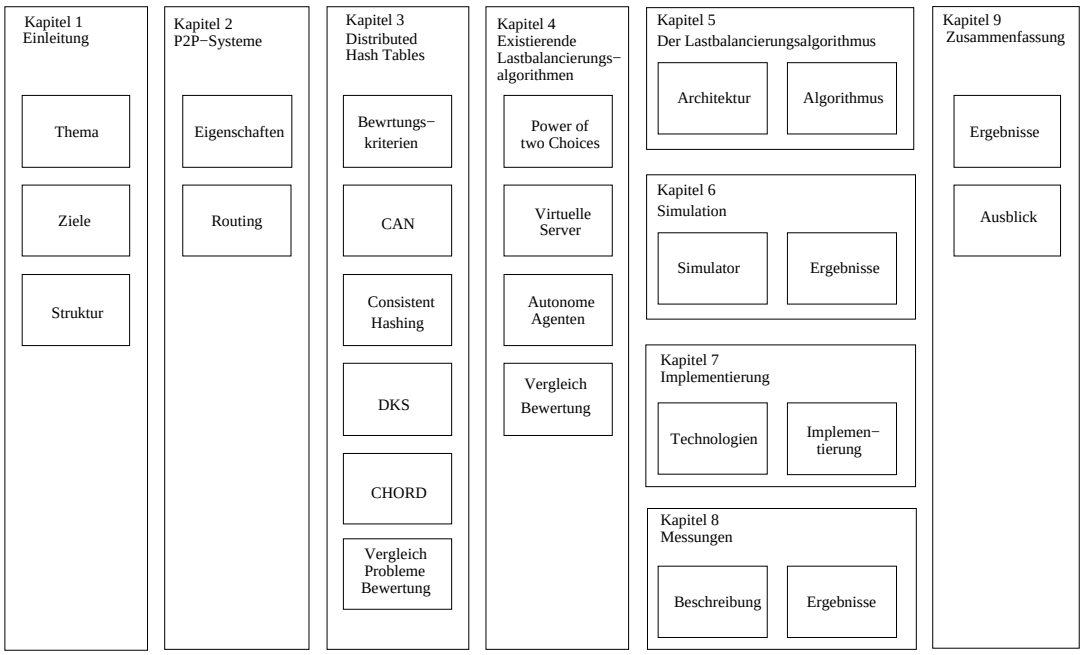


Abbildung 1.1: Struktur der Arbeit: Die einzelnen großen Kästen beschreiben die verschiedenen Kapitel (ohne Anhang). Innerhalb jedes Kastens werden die wichtigsten Teile des Kapitels in weiteren kleineren Kästen vor- und dargestellt. Mit der Anordnung der Kästen der Kapitel 5 bis 8 übereinander soll verdeutlicht werden, dass diese direkt vom Kapitel 5 abhängig sind und auf diesem aufbauen.

Zuerst werden im Kapitel 2 die Eigenschaften von Peer-to-Peer Systemen, sowie verschiedene Routingverfahren beschrieben. Es werden verschiedene Routingstrategien in Peer-to-Peer Systemen vorgestellt, und der Einsatz von Distributed Hash Tables zur Verwaltung von Daten im Peer-to-Peer Ansatz begründet.

Kapitel 3 beschreibt, bewertet und vergleicht die wesentlichen Distributed Hash Tables Systeme. Es werden die Nachteile und offenen Fragen bezüglich Distributed Hash Tables vorgestellt und gezeigt, warum die Lastbalancierung in diesen DHT Systemen alleine ungenügend ist.

Im Kapitel 4 werden existierende Lastbalancierungsalgorithmen vorgestellt. Diese einzelnen Systeme werden bewertet und deren Vor- und Nachteile aufgezeigt. Somit können diese Verfahren mit dem in dieser Arbeit vorgestellten Lastbalancierungsalgorithmus verglichen werden.

Das fünfte Kapitel erklärt den in dieser Arbeit entwickelten und verwendeten Algorithmus zur Lastbalancierung. Dabei werden die Voraussetzungen und Eigenschaften dargestellt und erläutert.

Im letzten Kapitel wurde der Algorithmus vorgestellt, im 6. Kapitel wird nun auf die Simulation der Algorithmen eingegangen. Dieses Kapitel beschreibt den Simulator, die Simulation und die Ergebnisse mit den vorgestellten Lastbalancierungsalgorithmen. Es werden sowohl einige der existierenden Algorithmen, als auch der neu entwickelte Algorithmus, simuliert, verglichen und bewertet.

Im folgenden 7. Kapitel wird auf die Implementierung eingegangen. Es wird unter anderem der Client vorgestellt, in den der Algorithmus eingebaut wurde.

Das 8. Kapitel zeigt die Messungen der Implementierung in verschiedenen Szenarios. Anschließend werden diese bewertet.

Abschließend enthält das 9. Kapitel die Zusammenfassung dieser Diplomarbeit und den Ausblick.

I think there's a world market
for about five computers.

– Thomas Watson, Gründer von IBM (1985)

Kapitel 2

Peer-to-Peer Systeme

Peer-to-Peer Systeme erfreuen sich großer Beliebtheit unter den Nutzern im Internet. Im Folgenden sollen drei Statistiken den heutigen Stand verdeutlichen.

- Im Juni 2003 haben allein in den USA nach einem Rückgang der Nutzerzahlen schätzungsweise immer noch 10,4 Millionen US-Amerikaner an Peer-to-Peer Musiktauschbörsen teilgenommen. Sie sollen dabei 655 Millionen Musikstücke allein in diesem Monat heruntergeladen haben [21].
- Das Peer-to-Peer Tauschsystem FastTrack mit seinem populärsten Klienten KaZaA hatte Mitte 2003 nach Rückgang der Nutzerzahlen immer noch bis zu 4 Millionen Nutzer gleichzeitig, die bis zu 800 Millionen Dateien anboten [40].
- Die Zahl der Nutzer von Tauschbörsen steigt dabei ständig. Das Peer-to-Peer Programm KaZaA Media Desktop wird laut [40] jede Woche über 3 Millionen Mal heruntergeladen, und der Vertreiber feierte nach eigenen Angaben im März 2003 bereits den 200 millionsten Download des KaZaA Media Desktop [55]. Der KaZaA Media Desktop ist ein Client zum Austausch von Dateien. Er wird hauptsächlich zum Anbieten und kostenlosen Herunterladen von urheberrechtsgeschützten Werken wie Musikstücken im MP3-Format benutzt.

Auch wenn Peer-to-Peer Systeme nicht nur aus Systemen für Tauschbörsen bestehen, zeigt dies doch deutlich das Ausmaß der Daten, die die Programme der Peer-to-Peer Tauschbörsen versenden, im heutigen Internetverkehr.

Im ersten Abschnitt dieses Kapitels sollen die Eigenschaften von Peer-to-Peer (P2P) Systemen [64] detailliert vorgestellt werden.

Um die Daten in einem großen P2P System finden zu können, bedarf es einer Routingstrategie. Eine Routingstrategie beschreibt, an welchen oder welche Knoten im P2P Netz eine Anfrage nach einem gewünschten Dokument o.Ä. weiterzugeben ist, wenn das Dokument nicht lokal beim Knoten gespeichert ist.

Es werden einige dieser verschiedenen Strategien, wie sie in verschiedenen Peer-to-Peer Systemen verwendet werden, vorgestellt und diskutiert. Abschließend wird der Einsatz von Distributed Hash Tables (DHTs) zur Verwaltung von Daten im P2P Ansatz begründet.

2.1 Eigenschaften von Peer-to-Peer Systemen

Anhand von [31, 52, 56] sollen im Folgenden die Eigenschaften von Peer-to-Peer Systemen genauer beschrieben werden.

P2P ist eine Möglichkeit in einem verteilten System Ressourcen zu teilen und Dienste anzubieten. Um diese Aufgabe erledigen zu können, basiert P2P auf demokratischen Prinzipien, d.h. alle Rechner im P2P System sind gleich. Dies bedeutet nicht, dass die Teilnehmer im P2P System, wie PCs oder PDAs nicht unterschiedliche Rechenleistung o.Ä. haben können, es gibt nur keinen hierarchischen Aufbau oder anderweitig ausgezeichnete Knoten. Gleichberechtigt bedeutet aber auch, dass jeder Rechner zugleich Client und Server ist. Ein P2P System ist selbstorganisierend und jeder Teilnehmer kann mit anderen Teilnehmern selbstständig in Kontakt treten und eine Kommunikation beginnen. Als Medium zur Kommunikation wird meist das Internet benutzt.

In einem P2P Netz wird jeder Teilnehmer, sei es ein PC, Handy oder PDA als Knoten bezeichnet. Als Nachbarschaft eines Knotens werden alle Knoten bezeichnet, die diesem Knoten bekannt sind. Üblicherweise hat ein Knoten in einem System mit N Knoten $O(1)$ oder $O(\log(N))$ Nachbarn, und muss deshalb nicht alle Knoten des Systems kennen. Damit wird die Skalierbarkeit des P2P Systems erhöht.

Die aktuellen P2P Systeme sind für spezielle einfache Aufgaben wie zum Beispiel das Anbieten und Tauschen von Daten ausgelegt. Durch die einfachen speziellen Aufgaben wird meistens der Programmcode eines jeden Knotens ebenfalls einfach und kurz gehalten. So benutzt Gnutella [11] nur fünf verschiedene Funktionen, das Distributed Hash Tables System Chord [57, 62, 13] nur wenige wichtige (je einen Befehl für das Einfügen eines Knotens und das Einfügen und Finden eines Dokuments). Dabei benutzen alle Knoten denselben Algorithmus zur Verrichtung ihrer Aufgaben.

Dadurch, dass viele Knoten eine Aufgabe erledigen, ist das System insgesamt weniger fehleranfällig. Andere Knoten können Aufgaben von ausgefallenen übernehmen. Eventuell gibt es einen Datenverlust, das System insgesamt läuft aber weiter.

Wenn alle Knoten gleichberechtigt sind und genügend Knoten eine Aufgabe erledigen, entstehen keine Flaschenhälse. Zentrale Server o.Ä. werden als Flaschenhälse bezeichnet, wenn zum Beispiel so viel Datenverkehr über diesen Server geleitet wird, dass er mit der Verarbeitung nicht nachkommt.

Da im P2P System Knoten aber nur Informationen über ihre Nachbarschaft besitzen, müssen sie ihre Arbeit mit unvollständigen Informationen verrichten.

Einige P2P Systeme wie das erste massenhaft eingesetzte P2P System Napster [38], das zum Dateiaustausch zwischen Benutzern benutzt werden konnte, weichen dabei aber von der Idee ab, dass alle Knoten direkt miteinander kommunizieren. In Napster wird zum Beispiel ein zentraler Server eingesetzt, der die Verwaltung des Katalogs übernimmt, der die Speicherorte der Daten verwaltet. Der Austausch der Daten erfolgt daraufhin direkt unter den Teilnehmern.

Software wie Napster und Gnutella [11] haben P2P populär gemacht. Das Filesharing (Austausch von Dateien), vor allem von Musikstücken o.Ä., mit diesen Produkten wird heute millionenfach benutzt. Durch die große Anzahl der Nutzer müssen jedoch intelligente Routingstrategien entwickelt werden, um den Netzverkehr so gering wie

möglich zu halten.

2.2 Routingstrategien in Peer-to-Peer Systemen

Um die Daten in einem großen P2P System finden zu können, bedarf es einer Routingstrategie für die beteiligten Knoten in diesem System.

Eine Routingstrategie beschreibt, wie sich ein Knoten im P2P Netz zu verhalten hat, wenn er eine Anfrage nach einem Dokument erhält, das nicht lokal bei ihm gespeichert ist.

Die Abbildung 2.1 zeigt in Anlehnung an [35] drei verschiedene Routingstrategien [19] in P2P Systemen.

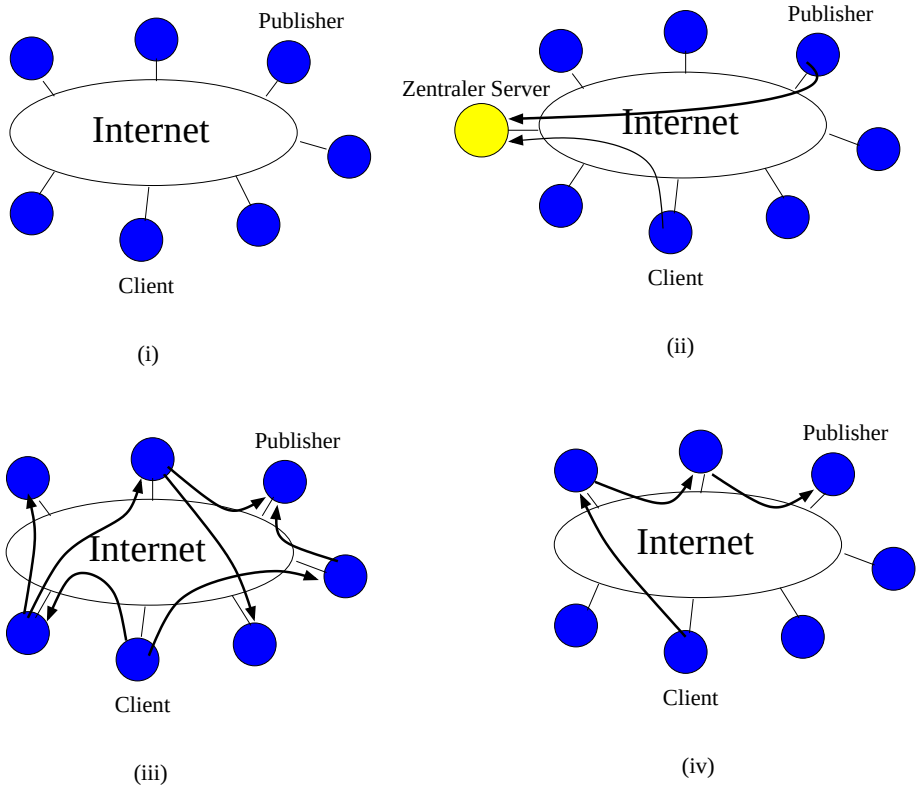


Abbildung 2.1: Routingstrategien: Die Abbildung zeigt in Teil (i) die Ausgangsstellung beim Suchen von Daten in einem P2P System. Der Client sucht ein Objekt, dass der Publisher gespeichert hat. Die Teile (ii) bis (iv) zeigen verschiedene Strategien, wie sie zum Beispiel von Napster (ii), Gnutella (iii) oder Chord als Distributed Hash Tables System (iv) benutzt werden.

Die Abbildung zeigt nur das Auffinden der Daten, nicht den eigentlichen Versand der Daten vom Anbieter (Publisher) an den suchenden Benutzer (Client). Der Versand erfolgt in allen Fällen direkt zwischen dem Clienten und dem Publisher.

Die Ausgangsstellung (i) und die einzelnen Strategien (ii) bis (iv) werden im Folgenden einzeln beschrieben.

- **Teil (i)** der Abbildung zeigt die Ausgangsstellung beim Suchen eines Objekts in einem P2P System.

  Ein Client sucht zum Beispiel ein Dokument, das auf einem beliebigen Knoten im Netz gespeichert sein kann. Alle Knoten sind über ein gemeinsames Netz, zum Beispiel das Internet, miteinander verbunden. Der Publisher hat das Dokument gespeichert, welches das Ziel der Suche des Clienten ist.

- **Teil (ii)** zeigt nun das Vorgehen von Systemen mit zentralen Komponenten, wie es das Peer-to-Peer System Napster [38] war.

  Es wird dem System aus Teil (i) ein zentraler Server hinzugefügt, der die Verwaltung der Daten und ihrer Speicherorte übernimmt. Der Publisher benachrichtigt den Server, über die bei ihm gespeicherten Dokumente, die er anderen Teilnehmern anbieten möchte. Der Client wendet sich bei der Suche nun an den Server und erfährt den Aufenthaltsort des Dokuments.

  Diese Art der Suche von Daten in einem Peer-to-Peer System ist zwar einfach, aber alle Anfragen gehen über diesen ausgezeichneten Server. Dadurch ist das System stark anfällig für Fehler. Fällt der zentrale Server aus, können keine Daten mehr im Peer-to-Peer Netz gefunden werden. Außerdem bildet dieser zentrale Punkt einen Flaschenhals für das System, da alle Anfragen von diesem Knoten nacheinander bearbeitet werden müssen.

- **Teil (iii)** zeigt die Flooding-Strategien (Fluten) von P2P Systemen wie Gnutella [11, 46, 50].

  Die Anfrage wird in das Netz an alle bekannten Knoten gesendet. Diese versenden wiederum die Nachricht an alle ihre bekannten Nachbarn. Der Knoten, der für die Suchanfrage zuständig ist, sendet schließlich, wenn ihn die Suchnachricht erreicht, das Ergebnis an den Sender der Anfrage zurück. Diese Strategie sorgt aber für starken Netzverkehr. So skalieren diese Systeme nicht, wie in [47] für Gnutella gezeigt wird.

  Bei diesem Verfahren wird eine so genannte Time-to-Live benutzt. Time-to-Live bedeutet wörtlich „Zeit zum Leben“ und wird mit TTL abgekürzt. Dies bedeutet, dass eine Nachricht nur eine begrenzte Anzahl oft mal weitergeleitet wird. Jeder Knoten, den eine Suchnachricht erreicht, verringert die TTL der Nachricht, bevor er sie an seine Nachbarschaft weiter sendet, um eins. Nur Nachrichten, deren TTL größer als null ist, werden weitergeleitet. So wird verhindert, dass eine Nachricht unendlich lange im System verbleibt.

  Durch eine Time-to-Live gibt es aber keine Garantie dafür, dass im System gespeicherte Daten tatsächlich bei einer Anfrage gefunden werden. Es kann sein, dass die Time-to-Live abgelaufen ist, bevor eine Nachricht den zuständigen Knoten erreicht hat.

- **Teil (iv)** zeigt die Strategie von Distributed Hash Tables.

  In DHTs wird die Anfrage zielgerichtet anhand bestimmter festgelegter Kriterien (Routing-Tabellen) an einen Knoten gesendet. Dieser fährt so fort, und so wird die Anfrage gezielt an den Knoten weitergeleitet, der die Daten speichert. Dadurch wird weniger Netzverkehr als beim Fluten erzeugt, die meisten Systeme erreichen den Zielknoten in $O(\log n)$ Schritten. Im folgenden Kapitel werden deshalb einige Distributed Hash Tables vorgestellt.

Keep it simple.
As simple as possible.
But no simpler.

– Albert Einstein (1879 - 1955)

Kapitel 3

Distributed Hash Tables

Eine Distributed Hash Table (DHT) ist eine über mehrere Rechner verteilte Hashtabelle. Dabei wird in einem DHT System ein Objekt, wie zum Beispiel ein Dokument oder eine Datei, in ein definiertes abgeschlossenes Intervall mittels einer Hashfunktion abgebildet. Dazu wird auf den Schlüssel dieses Objektes eine Hashfunktion angewendet. Dieser Schlüssel kann zum Beispiel aus dem Dateinamen oder andere Metadaten berechnet werden. Unter diesem Schlüssel wird die Datei dann im System verteilt. Dieser Schlüssel wird wieder benötigt, um die Daten im System bei einer Suche zu finden.

Weil die Schlüssel alle in einem vorgegebenen Wertebereich sind, kann das gesamte Intervall aller möglichen Werte auf die einzelnen beteiligten Knoten im System verteilt werden. Um eine Datei zu finden, wird der Hashwert berechnet und beim entsprechenden zuständigen Knoten angefragt.

Durch diese Vorgehensweise sind die DHTs deutlich netzschonender als klassische Peer-to-Peer Systeme wie zum Beispiel das inzwischen eingestellte Napster [38] oder Gnutella [11]. Manche Systeme wie Napster benutzen zentrale Server oder die Systeme wie Gnutella leiten Anfragen weiter, indem sie das Netz mit dieser Anfrage fluten. So skalieren diese Systeme nicht [47, 1].

Der Unterschied zwischen den verschiedenen DHT Systemen ist nun der Aufbau des Netzwerkes und der Ablauf des Suchprozesses.

Zuerst wird in diesem Kapitel Kriterien zur Bewertung erstellt, mit dem alle im Folgenden vorgestellten DHTs bewertet und verglichen werden.

Im Folgenden werden dann die wichtigsten DHT Systeme vorgestellt und untersucht.

In Abschnitt 3.2 ab Seite 12 wird das Content-Addressable Network (CAN) [44] vorgestellt. Die Idee von CAN ist es, die Daten in einem virtuellen mehrdimensionalen Raum anzuordnen. Durch diese Anordnung der Knoten kann eine Suchnachricht in einem CAN System deshalb gezielt an den Knoten weitergeleitet werden, der für die Datei zuständig ist.

Alle weiteren vorgestellten DHTs bauen aufeinander auf.
Im Abschnitt 3.3 wird Consistent Hashing [24] eingeführt. Chord [57] im Abschnitt 3.4 erweitert die Ideen von Consistent Hashing um eine intelligentere Suche und Maßnahmen zur Stabilisierung. DKS(N, k, f) [2] in Abschnitt 3.5 ab Seite 28 erweitert Chord wiederum um eine andere Fehlerbehandlung und eine Einteilung in

verschiedene Levels.

Abschließend werden weitere DHTs kurz vorgestellt, sowie Probleme und offene Fragen in Bezug auf DHTs aufgezeigt.

3.1 Kriterien

Alle DHT Systeme werden in diesem Kapitel nach einem festgelegten Schema bewertet. Dadurch können diese Systeme zum Abschluss besser untereinander verglichen werden.

Als Referenz wird dabei das klassische Peer-to-Peer System Gnutella [11, 46] im nächsten Abschnitt 3.1.1 bewertet.

Die einzelnen Kategorien [3, 14, 15], in denen die Systeme bewertet werden, sind dabei:

- Topologie (Topology)
- Platzieren der Daten (Data Placement)
- Routen von Nachrichten (Message Routing)
- Ausdrucksstärke der Suche (Expressiveness)
- Ergebnismenge der Suche (Comprehensiveness)
- Selbstständigkeit (Autonomy)
- Effizienz (Efficiency)
- Robustheit (Robustness)

Die ersten drei Kategorien sind eine kurze Beschreibung des DHT Systems.

Topologie beschreibt, wie die einzelnen Knoten im System verbunden sind.
Die Kategorie **Platzieren der Daten** beschreibt, wie die Daten auf die einzelnen Knoten verteilt werden, während **Routen von Nachrichten** angibt, wie die Nachrichten zwischen den Knoten weitergeleitet werden. Ein Beispiel wäre das Fluten des Netzes für eine Anfrage.

Die nächsten fünf Kategorien sind eine Bewertung des DHT Systems.

Die **Ausdrucksstärke der Suche** bewertet die Möglichkeiten der Abfrage, mit der die einzelnen Daten im System gesucht werden können. So ist ein wichtiges Kriterium der Ausdrucksstärke, ob in dem System zum Beispiel die Suche von Daten mit Wildcards (wie *) oder regulären Ausdrücken möglich ist.
Viele Systeme garantieren nicht, dass existierende Elemente gefunden werden. Die **Ergebnismenge der Suche** bewertet, ob garantiert wird, dass alle existierenden Elemente gefunden werden oder es vorkommen kann, dass kein Element gefunden wird, obwohl es im System vorhanden ist.
Die **Selbständigkeit** beschreibt, inwieweit die einzelnen Knoten autonom sind. Kann ein Knoten zum Beispiel bestimmen, wo seine Daten gespeichert werden oder mit welchen, vielleicht für ihn vertrauenswürdigen Knoten, er sich nur verbindet?

Die **Effizienz** bewertet die Benutzung von Ressourcen durch die Distributed Hash Table. Sie bewertet zum Beispiel den Ressourcenverbrauch des Systems zum Durchsatz von Anfragen oder gespeicherten Dokumenten.
Robustheit schließlich vergleicht die Fähigkeit des Systems, auf Fehler zu reagieren.

Die letzten fünf Kategorien werden aufsteigend von + (ungenügend) bis ++++ (sehr gut) bewertet.

3.1.1 Beispiel Gnutella

Die Tabelle 3.1 zeigt Gnutella [11, 46, 50], eingeordnet nach den Kategorien in 3.1.

Eigenschaften	Gnutella
Topologie	zufällig
Platzieren der Daten	beliebig
Routing	Fluten
Ausdrucksstärke	++++
Ergebnismenge	++
Selbständigkeit	++++
Effizienz	+
Robustheit	++

Tabelle 3.1: Bewertung: Einordnung des Peer-to-Peer Systems Gnutella nach den in Abschnitt 3.1 beschriebenen Kriterien [3, 14, 15].

Begründung der einzelnen Bewertungen:
Die Ausdrucksstärke von Gnutella ist hoch, da das System auch Bereichsabfragen unterstützt. Dabei wird aber nicht garantiert, dass vorhandene Elemente gefunden werden. Da aber mit zunehmender Ausbreitung der Suchnachricht das Finden der Daten wahrscheinlicher wird, erhält Gnutella in der Reichhaltigkeit der Antwort ein mittleres Ergebnis. Die Selbstständigkeit von Knoten in Gnutella ist dafür sehr hoch. Jeder Knoten in Gnutella kann beliebig ein- und austreten und selbständig dafür sorgen, dass nur Anfragen von vertrauenswürdigen Knoten angenommen werden, ohne dass das System gestört wird. Die Effizienz ist dagegen ungenügend. Nachrichten werden als Broadcast versendet, das Netz wird mit Anfragen überflutet. Das System ist robust, ausgefallene Knoten stören das Fortfahren des Systems nicht, die gespeicherten Daten des Knotens sind nur unter Umständen nicht mehr im System vorhanden.

3.2 Content-Addressable Network

Das erste Distributed Hash Tables System, das in dieser Arbeit vorgestellt werden soll, ist CAN. „CAN: A Scalable Content-Addressable Network“ [44] wurde von Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp und Scott Shenker an der University of California, Berkeley und dem ATT Center for Internet Research an der ICSI entwickelt und beruht auf einer Doktorarbeit von Sylvia Ratnasamy [43].

Die Idee von CAN ist es, die Daten in einem virtuellen mehrdimensionalen Raum anzuordnen. Jeder einzelne Teilnehmer, der Daten verwalten kann und am System teilnehmen möchte, wie ein Computer, PDA, etc., wird als Knoten aufgefasst und in das Peer-to-Peer System integriert. Jeder dieser Knoten bekommt eine eigene Region zugeteilt, für die er dann alleine zuständig ist. Zuständig bedeutet hierbei, dass er alle Anfragen, die seine Region betreffen, beantwortet.

Durch die Anordnung der Knoten in einem virtuellen mehrdimensionalen Raum können Nachrichten, wie zum Beispiel Suchnachrichten eines Benutzers, in einem CAN System gezielt an den Knoten weitergeleitet werden, der für diese Nachricht zuständig ist, weil er zum Beispiel dieses Dokument verwaltet.

Anfragen werden im Gegensatz zu Gnutella also nicht mehr per Multicast versandt. Dadurch wird das zu Grunde liegende Verbindungsnetz, wie zum Beispiel das Internet, stark entlastet.

3.2.1 Funktionsweise

Sämtliche Knoten in dem DHT System CAN verwalten einen Teil einer verteilten Hashtabelle. Alle Knoten werden dazu in einem virtuellen D-dimensionalen kartesischen Koordinatensystem in einem D-Torus (Abbildung 3.1(ii)) verteilt. Dazu wird jedem Knoten ein Teil des Koordinatensystem zugeteilt. Dieser Bereich kann durch seine Koordinaten im Raum eindeutig bestimmt werden. Für diesen Bereich ist der Knoten nun alleine zuständig.

Für neu einzufügende Daten, wie ein Dokument, wird durch eine Hashfunktion ebenfalls ein Punkt im gesamten Raum zufällig, aber eindeutig bestimmt. Der für diesen Bereich des Systems zuständige Knoten speichert dann die Daten.

Durch diese beschriebenen Eigenschaften des Systems CAN beträgt die mittlere Pfadlänge bei N Knoten im System $\frac{D}{4} \cdot N^{\frac{1}{D}}$, und jeder Knoten hat maximal $2 \cdot D$ Nachbarn. Dies bedeutet, dass im Mittel jeder Knoten nach $\frac{D}{4} \cdot N^{\frac{1}{D}}$ Schritten von jedem anderen Knoten erreicht werden kann.

Im Folgenden wird der Aufbau und die Organisation eines zweidimensionalen CAN Koordinatensystems genauer beschrieben. In höheren Dimensionen funktioniert CAN entsprechend, ist aber schwieriger dar- und vorzustellen. Dabei wird nach der folgenden Abbildung 3.1 zur besseren Veranschaulichung kein Torus mehr in Grafiken verwendet, sondern ein 2-dimensionales Koordinatensystem, dass eigentlich an den Enden verbunden ist.

Die Abbildung 3.1(i) zeigt ein 2-dimensionales $[0-1][0-1]$ Koordinatensystem. Alle Elemente im CAN System haben Koordinaten, die auf beiden Achsen im Intervall $[0-1]$ liegen. Im Beispiel haben sich fünf Knoten den gesamten Bereich untereinander aufgeteilt. Knoten B ist zum Beispiel für die Zone $(0.5-1, 0-0.5)$ zuständig.

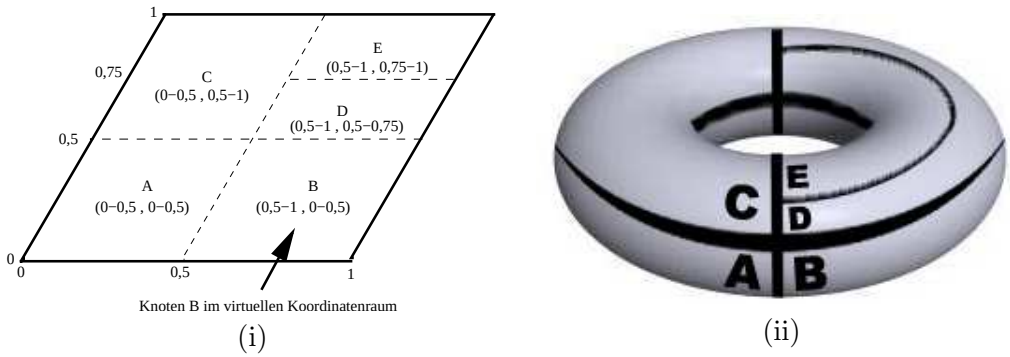


Abbildung 3.1: CAN: Im ersten Teil der Abbildung ist ein CAN Feld dargestellt, das sich fünf verschiedene Knoten untereinander aufgeteilt haben. Im zweiten Teil ist der eigentliche Aufbau als Torus und die gleiche Verteilung der Knoten darin dargestellt.

Dabei ist aber zur Veranschaulichung die Verbindung der gegenüberliegenden Ränder untereinander weggelassen. Das 2-dimensionales CAN System aus Teil (i) entspricht im Aufbau eigentlich einem Torus, der im Teil (ii) dargestellt ist.

3.2.1.1 Nachbarn

Jeder Knoten in einem CAN System speichert eine Liste seiner Nachbarn. Mögliche Nachbarn eines Knotens sind die Knoten, die direkt an seinen Bereich angrenzen.

Jeder Knoten speichert zu jeder Seite jedoch nur einen Knoten. Es müssen, wenn die Nachbarn eines Knotens ihre Bereiche aufteilen, trotzdem nur maximal $2 \cdot D$ Knoten gespeichert werden. Dabei speichert ein Knoten nur Kontaktinformationen, wie die IP-Adresse, und die Koordinaten der Zone seines Nachbarn.

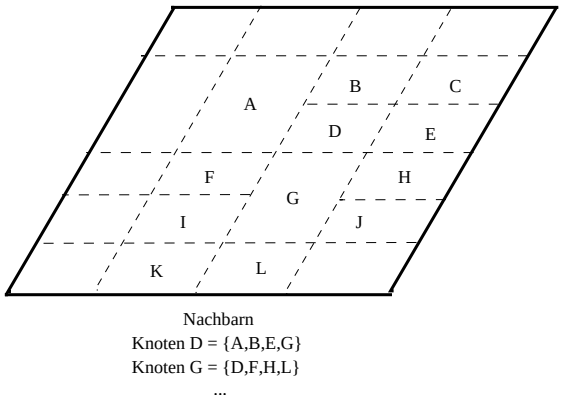


Abbildung 3.2: Nachbarschaft: In einem CAN Koordinatensystem sind die Nachbarn der Knoten D und G dargestellt. Jeder Knoten merkt sich zu jeder Seite nur einen Nachbarn. Deshalb speichert Knoten G für die linke Nachbarschaft nur Knoten F und nicht zusätzlich Knoten I.

Die Abbildung 3.2 zeigt die gespeicherten Nachbarn von Knoten D und G im 2-dimensionalen CAN System.

Alle Nachbarn eines Knotens müssen mit diesem also immer eine D-1 dimensionale Berührungsfläche haben. Im 2-dimensionalen CAN entspricht dies einer Geraden.

Damit die Menge der Nachbarn immer konsistent gehalten werden kann und die

Knoten ihre Nachbarschaft kennenlernen können, teilt jeder Knoten periodisch seinen Nachbarn die Knoten mit, die ihn aus seiner Sicht umgeben.

Die Aufrechterhaltung der beschriebenen Nachbarschaft ist wichtig für das korrekte Weiterleiten von Nachrichten, wie im folgenden Abschnitt gezeigt wird.

3.2.1.2 Routing

Alle Nachrichten im CAN System haben einen eindeutigen Zielpunkt im DHT. Zum Routen von diesen Nachrichten benutzen die Knoten ihre Tabellen über die Nachbarschaft. Dabei wird eine Nachricht immer von einem Knoten zu dem nächsten Knoten weitergeleitet, dessen euklidscher Abstand der kleinste zum Zielpunkt ist. Ist dieser Knoten nicht erreichbar, zum Beispiel weil er oder die Netzwerkverbindung ausgefallen ist, so wird versucht, über einen anderen Nachbarn mit dem nächst größeren euklidischen Abstand zu routen. Die durchschnittliche Pfadlänge von einem Knoten zum Zielpunkt in einem D-dimensionalen CAN beträgt dabei $\frac{D}{4} \cdot N^{\frac{1}{D}}$.

Die folgende Abbildung 3.3 zeigt wie eine Suchnachricht vom Knoten D zum Punkt (x, y) geroutet wird.

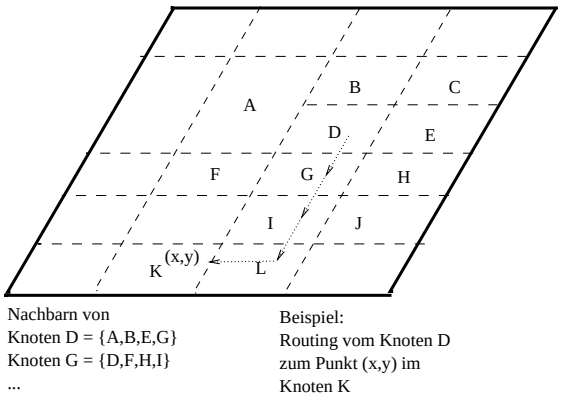


Abbildung 3.3: Routing: Eine Nachricht soll vom Knoten D zum Punkt (x, y) geschickt werden. Für diesen Punkt ist der Knoten K zuständig, da dieser Punkt in seinen Bereich fällt. Knoten D routet die Nachricht zum Knoten G, der weiter über die Knoten I und L zum Endpunkt Knoten K.

3.2.2 Einfügen und Suche von Daten

Im gesamten CAN System muss allen Knoten die Hashfunktion h bekannt sein. Diese bildet Daten zum Beispiel anhand ihres Namens zufällig aber eindeutig in das Koordinatensystem ab. Zum Einfügen oder Suchen von Daten wird als erstes vom Schlüssel der Daten der Punkt im Koordinatenraum mit Hilfe der Hashfunktion bestimmt. Als Schlüssel kann zum Beispiel der Dateiname dienen. Im zweidimensionalen Raum bestimmt die Funktionen $x = h_x(key)$ und $y = h_y(key)$ einen eindeutigen Punkt im Raum. Anschließend werden die Daten oder die Anfrage zum berechneten Punkt (x, y) mit Hilfe des beschriebenen Routings transferiert. Dort speichert der zuständige Knoten die Daten oder liefert bei einer Suche die gewünschten Daten an den Initiator der Suche zurück.

Jeder Knoten speichert aber auch alle Elemente (wie Dokumente), die seine Nachbarn gespeichert haben. Diese Eigenschaft wird im Fehlerfall benötigt und wird im Abschnitt über das Entfernen von Knoten vorgestellt.

3.2.3 Hinzufügen von Knoten

Damit ein neuer Knoten zum CAN System hinzugefügt werden kann, muss dieser einen bereits im Netzwerk eingebundenen Knoten kennen. Es wird anhand der IP-Adresse des neuen Knotens ein Platz im CAN System bestimmt. Daraufhin wird mit dem vorgestellten Routing der Knoten zu seinem Platz im System weitergeleitet.

Der für diesen Bereich bisher zuständige Knoten teilt seinen bisherigen Bereich des Koordinatensystems in der Mitte. Daraufhin ist der neue Knoten für eine Hälfte des Bereichs zuständig, und der alte Knoten sendet dem neuen Knoten die Daten, die dieser ab jetzt verwalten muss. Zum Abschluss werden die Nachbarn ausgetauscht und die Nachbarn des neuen Knotens über die neue Aufteilung informiert.

In der Abbildung 3.4 wird als Beispiel das Intervall vom Knoten D durch das Hinzufügen eines neuen Knotens mit der Bezeichnung L aufgeteilt.

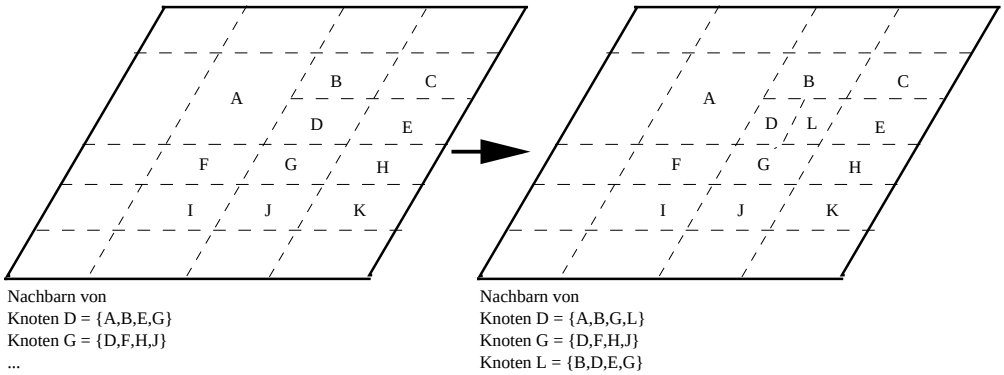


Abbildung 3.4: Hinzufügen eines Knotens: Dem linken CAN System mit den dargestellten Knoten A bis K tritt im zweiten Teil der Knoten L bei. Dabei wird der bisherige Bereich von D auf die Knoten D und L aufgeteilt. Nur die Nachbarn von Knoten L müssen dabei ihre Nachbarschaft aktualisieren, alle anderen Knoten betrifft diese Unterteilung nicht.

3.2.4 Entfernen von Knoten

Wenn ein Knoten das CAN System verlässt, muss gewährleistet sein, dass ein anderer Knoten dieses Intervall übernimmt.

Im Folgenden sind dazu zwei Fälle zu unterscheiden. Entweder der Knoten weiß, das er das System verlassen muss oder ein Knoten ist ausgefallen, auch zum Beispiel durch Störungen im Verbindungsnetz.

Wenn ein Knoten weiß, dass er nicht mehr am Netzwerk teilnehmen kann, teilt er dies seinen Nachbarn mit. Diese Knoten prüfen, ob sie den Bereich des Knotens übernehmen können. Sollte kein Knoten den Bereich übernehmen wollen, wird die Zone mit dem Nachbarn, der den kleinsten Bereich verwaltet, vereinigt. Sobald sich ein Knoten bereit erklärt hat den Bereich zu übernehmen, werden beide zu einem

neuen großen verschmolzen.

Wenn ein Knoten ausgefallen ist, wird dies von den Nachbarn durch die ausbleibenden periodischen Nachrichten bemerkt. Dann übernimmt der Knoten der Nachbarschaft, der bisher den kleinsten Bereich hat, die Zone des ausgefallenen Knotens.

Da jeder Knoten auch alle Elemente, die seine Nachbarn gespeichert haben, ebenfalls speichert, gehen durch den Ausfall eines einzelnen Knotens keine Daten verloren.

3.2.5 Einordnung

Die Tabelle 3.2 zeigt CAN eingeordnet nach den Kriterien aus Abschnitt 3.1.

Eigenschaften	**CAN**
Topologie	d-dimensionaler Torus
Platzieren der Daten	Hashing
Routing	Direkt
Ausdrucksstärke	+
Ergebnismenge	++++
Selbstständigkeit	++
Effizienz	+++
Robustheit	+++

Tabelle 3.2: Bewertung: Einordnung von CAN nach Abschnitt 3.1.

Begründung der einzelnen Bewertungen für die Kategorien Ausdrucksstärke bis Robustheit:
In CAN existiert nur die Suche nach einem bestimmten, genau angegebenen Schlüssel. Bereichsabfragen sind ohne Zusätze nicht effizient möglich. Deshalb wird die Ausdrucksstärke der Suche mit schlecht bewertet. CAN garantiert aber im Gegensatz zu Gnutella, dass wenn ein Element vorhanden ist, es auch gefunden wird. Die Knoten sind dagegen kaum selbstständig. Sie können bis auf den Wunsch dem System bei- oder auszutreten keine eigenen Entscheidungen treffen. Die Knoten können nicht beeinflussen, mit welchen Knoten sie sich verbinden, oder wo welche Daten gespeichert werden, da das System sonst nicht mehr funktionstüchtig ist. Die Effizienz dagegen ist gut. Die Nachrichten werden über Routing-Tabellen direkt gesendet, es entsteht kaum Netzverkehr. Dabei hängt die Geschwindigkeit der Suche aber von der gewählten Dimension ab. Je höher die Dimension gewählt wird, umso mehr Nachbarn hat ein Knoten. Dadurch werden dann aber auch mehr periodische Nachrichten an die Nachbarn verschickt. Wenn ein Knoten in CAN ausfällt, sind die Daten immer noch in den Nachbarzellen vorhanden, es können also einzelne Knoten austreten, ohne dass die Daten für das System verloren sind. Es dürfen aber nicht ganze Bereiche auf einmal ausfallen. Deshalb erhält CAN für die Robustheit ein gutes Ergebnis.

3.3 Consistent Hashing

Die Idee in Consistent Hashing [24] ist, verschiedene existierende Hashing-Methoden wie die aus [41] zu erweitern. Vorher existierende hash-basierte Systeme funktionieren 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 ein einfaches Schema der Datenreplikation. Der Server verteilt mittels Hashfunktion [12] Kopien von Daten an verschiedene 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 deshalb das Problem der verschiedenen Sichten (view). Eine Sicht ist eine Menge von Caches, die ein Client kennt. Ein Client nutzt eine consistent Hashfunktion, um ein Objekt auf einen Cache in seiner Sicht abzubilden.

3.3.1 Definitionen

In diesem Abschnitt werden zunächst einige Definitionen 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 Hashfunktionen.

Eine zufällige ranged Hashfunktion **(random ranged hash function)** ist eine Funktion, die eine Funktion zufällig aus oben beschriebener Familie auswählt.

3.3.2 Konstruktion

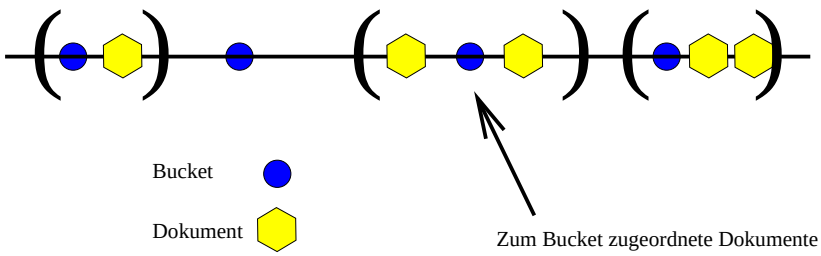


Abbildung 3.5: Server und Dokumente in Consistent Hashing: Die Dokumente werden den Buckets (Kreise) zugeteilt, denen sie am nächsten sind.

In diesem Abschnitt wird die Konstruktion einer solchen ranged hash family dargestellt. 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.

In der Abbildung 3.5 werden die Buckets und ihre zugeordneten Dokumente durch Klammern gezeigt.

Die Hashfunktion verteilt diese Kopien auf die Buckets. Diese Funktionen bilden die ranged Hashfamilie. 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.

3.3.3 Implementation

Durch die Hashfunktionen von Consistent Hashing werden Einträge in Behälter in einer Sicht (View) abgebildet. Eine View ist eine Menge von Caches, die ein bestimmter Client berücksichtigt. Ein Client verwendet eine Hashfunktion, um einen Eintrag in einen der Caches in seiner Ansicht abzubilden.

Die genaue Funktionsweise wird anhand von [26] erklärt. Eine Hashfunktion bildet Strings (Zeichenketten) in ein Zahlenintervall $[0, \ldots, M]$ ab. Geteilt durch M ergibt das eine Hashfunktion im Bereich [0, 1]. Diese wird der Reihe nach auf dem Einheitskreis angeordnet. Dadurch wird jedes Datum, zum Beispiel eine URL, auf einen Punkt auf dem Kreis abgebildet. Zur gleichen Zeit wird jeder Cache ebenfalls auf einen Punkt im Kreis abgebildet. Aus Effizienz-Gründen ist nun für jede URL derjenige Cache zuständig, der im Uhrzeigersinn der nächstgelegene ist, nicht mehr der Knoten, der dem Element insgesamt am nächsten liegt.

Abbildung 3.6 zeigt in Teil (i), wie die Dokumente 1 bis 6 auf die jeweils nachfolgenden Server A bis D verteilt sind. Server A ist für die Dokumente 2 und 3, Server B für die Dokumente 4 bis 6 usw. zuständig. In Teil (ii) wird das Dokument 2 dem neu hinzugekommenen Server E zugeordnet, Dokument 3 bleibt bei Server A.

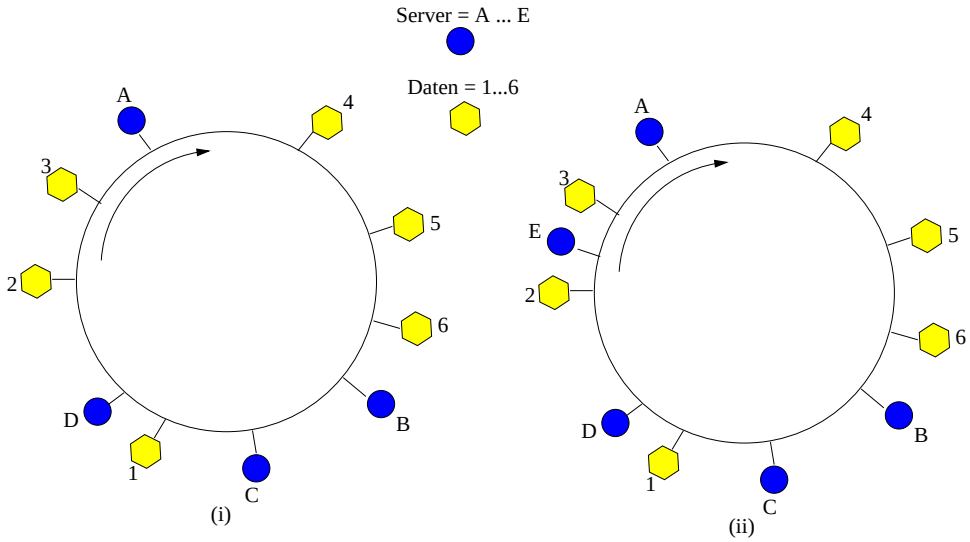


Abbildung 3.6: Server und Dokumente in Consistent Hashing: Im 1. Teil der Abbildung sind im Kreis die Dokumente 1 bis 6 auf die jeweils nachfolgenden Knoten A bis D verteilt. Nachdem im 2. Teil der Abbildung der Knoten E hinzugekommen ist, ist dieser nun für das Dokument 2 zuständig.

3.3.4 Einordnung

Die folgende Tabelle zeigt Consistent Hashing eingeordnet nach den Kategorien aus Abschnitt 3.1 von Seite 10.

Eigenschaften	Consistent Hashing
Topologie	Ring
Platzieren der Daten	Hashing
Routing	Ringsuche
Ausdrucksstärke	+
Ergebnismenge	++++
Selbstständigkeit	++
Effizienz	+
Robustheit	++

Tabelle 3.3: Bewertung: Einordnung von Consistent Hashing nach Kriterien, wie sie im Abschnitt 3.1 beschriebenen wurden.

Begründung der einzelnen Bewertungen:
Zum Suchen von Daten muss wieder der komplette Schlüssel angegeben werden. Auch hier sind wie bei CAN keine Bereichsabfragen oder Ähnliches möglich. Im System vorhandene Elemente werden aber garantiert gefunden. Die Knoten haben keine Selbstständigkeit, da sie zum Beispiel nicht bestimmen können, wo Elemente platziert werden. Das System ist nicht effizient. Da Consistent Hashing nur Vorgänger und Nachfolger kennt, muss unter Umständen der halbe Ring abgelaufen werden, um ein Dokument zu finden. Es gibt auch keine Methoden zur Stabilisierung des Systems, wenn ein Knoten ausfällt. Es wird auch nicht beschrieben, was die Knoten machen, wenn sie den Ausfall eines anderen bemerken. Da aber in den Caches von Consistent Hashing nur Kopien von Daten eines Servers gespeichert werden, sind bei Ausfall der Knoten immer noch alle Daten vorhanden, das System ist aber nicht mehr lauffähig.

3.3.5 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 nächstgelegenen 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 Nachfolgern abgelaufen werden, um den zuständigen Knoten zu finden.

3.4 Chord

Da das Navigieren in Consistent Hashing nur mit Hilfe des Nachfolgers und Vorgängers unter Umständen bedeutet, den halben Ring ablaufen zu müssen, wurde Chord als Erweiterung der Idee des Consistent Hashing entwickelt.

„Chord A Scalable Peer-to-peer Lookup Service for Internet Applications“ wurde vom Ion Stoica et al. am MIT Laboratory for Computer Science entwickelt [57, 62, 13]. Das Ziel war ein skalierbares Protokoll zum Suchen von Daten in einem dynamischen Peer-To-Peer Netzwerk zu entwickeln.

Das Chord Protokoll kennt dabei nur wenige Befehle. Der wichtigste ist lookup(key). Damit wird zu einen gegebenen Schlüssel die IP-Adresse des zugehörigen Knotens geliefert. Weiterhin existieren noch Befehle, um ein Knoten oder ein Dokument in das bestehende System einzufügen.

Chord sorgt zusätzlich für die Aufrechterhaltung des Netzwerkes. Andere Funktionen des Peer-to-Peer Systems dagegen müssen von den Applikationen realisiert werden, wie zum Beispiel Authentisierung und Caching.

Das System baut dabei auf Consistent Hashing aus dem vorherigen Abschnitt auf und erweitert dieses um so genannte Finger-Tabellen, die in Abschnitt 3.4.2 vorgestellt werden.

3.4.1 Funktionsweise

Das Chord-Protokoll spezifiziert, wie der Aufenthaltsort von Schlüsseln (keys, Identifier) gefunden werden kann, wie neue Knoten zum System hinzugefügt werden können und wie sich die Knoten beim Verlassen des Systems zu verhalten haben.

Chord benutzt dabei eine Variante des im letzten Abschnitt vorgestellten Consistent Hashing, um Knoten und allen Objekten eindeutige Positionen im Chord System zuzuweisen. Da die einzelnen Knoten in Chord nicht alle anderen Knoten des Systems kennen müssen skaliert Chord besser als Consistent Hashing.

Allen Knoten in dem Chord System muss eine festgelegte Hashfunktion bekannt sein. Diese Hashfunktion erzeugt aus den IP-Adressen der beteiligten Knoten und den Metadaten von Daten, die gespeichert werden sollen, m-bit große Identifier. Für die Berechnung des Identifier von Daten kann zum Beispiel der Dateiname verwendet werden.

Da die Identifier alle m-bit groß sind, haben alle Knoten und Dokumente mit der Hashfunktion eine eindeutige Zahl aus dem Intervall von 0 bis 2^m zugeordnet bekommen. Die Identifier in Chord bilden so einen virtuellen geordneten Kreis $modulo\, 2^m$. Die Schlüssel werden im Uhrzeigersinn mit 0 bis $2^m - 1$ bezeichnet.

Mit dem successor (Nachfolger) und predecessor (Vorgänger) kann im virtuellen Kreis von Chord navigiert werden. Der successor(k) eines Schlüssels k im virtuellen Chord-Ring ist der erste im Uhrzeigersinn nächstfolgende Knoten suc mit $suc \geq k$. Der predecessor ist für einen Knoten n oder Schlüssel k der Knoten pre mit dem höchsten Schlüssel im virtuellen Ring für den $pre < n$ bzw. $pre < k$ gilt.

Als Beispiel zeigt Abbildung 3.7 einen Kreis mit $m = 4$. 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

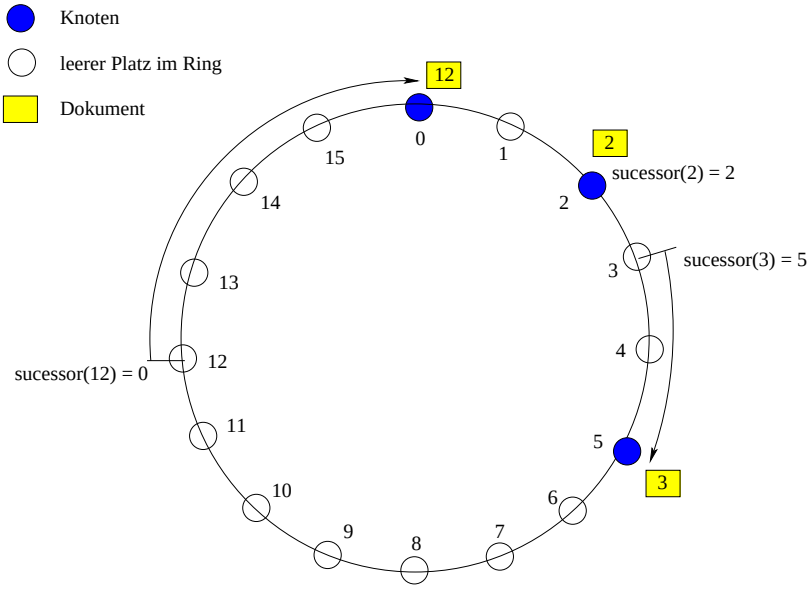


Abbildung 3.7: Chord-Ring: Beispiel eines Ringes mit $m = 4$, also 16 möglichen Plätzen für Elemente wie Dokumente oder Knoten. Die Pfeile zeigen die Nachfolger (successor) der Dokumente an. Die Dokumente sind an den Knoten eingezeichnet, bei denen sie wirklich gespeichert werden. Nur gefärbte Kreise sind Knoten, alle anderen Plätze im virtuellen Chord-Ring sind unbelegt.

die 2 ist, kann der Schlüssel 2 beim Knoten 2 lokalisiert werden.

Ebenso wird der Schlüssel 3 im Knoten 5 gespeichert, im 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.

3.4.2 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. Die Finger-Tabellen werden auch als Routing-Tabellen bezeichnet.

Die m Einträge (Finger) jedes Knotens zeigen in dieser Finger-Tabelle auf festgelegte Knoten innerhalb des Chord-Kreises. Die Anzahl der Einträge und welche Knoten vorhanden sind hängt also direkt von der Größe des Ringes m ab. Diese Finger-Tabelle wird zum Suchen des für einen Schlüssel zuständigen Knotens benutzt.

Zusätzlich zu den Fingern wird in dieser Finger-Tabelle der Nachfolger und Vorgänger eines Knotens gespeichert.

Die einzelnen Finger haben ein festgelegtes Format. In jeder Zeile der Finger-Tabelle wird ein Finger mit den 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} Schritte von n entfernt ist. Also gilt für den i-ten Eintrag $s = sucessor(n + 2^i)$, wobei $1 \leq i \leq m$ ist (alle Rechnungen modulo 2^m, um mit den Berechnungen nicht aus dem Ring zu laufen). Chord nennt s den i-ten Finger von Knoten n, bezeichnet wird er mit n.finger[i].node.

Die Finger teilen so den Ring ab der Position des Knotens in immer größere Intervalle. Das letzte Intervall entspricht der Hälfte des virtuellen Chordkreises. Das vorletzte Intervall hat die Größe von einem Viertel des Chordkreises.

Das erste Intervall hat deshalb eine Größe von eins, und der erste Finger-Eintrag entspricht genau dem Nachfolger.

Alle Einträge, die in der Finger-Tabelle gespeichert sind, erklärt folgende an [57] angelehnte Tabelle 3.4.

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	$erster \, Knoten \geq n.finger[k].start$	IP-Adresse des zuständigen Knotens
successor	finger[1].node	Der Nachfolger von Knoten n
predecessor	Der Vorgänger von Knoten n	

Tabelle 3.4: Chord: Definition der Variablen in Chord für Knoten n. Die ersten 3 Zeilen bilden zusammen einen Finger. Jeder Knoten speichert in seiner Finger-Tabelle genau m Finger. Zusätzlich werden Vorgänger und Nachfolger in der Tabelle gespeichert. Der erste Finger entspricht dabei genau dem Nachfolger (successor).

Die folgende Abbildung 3.8 zeigt für $m = 3$ (6 Identifier) die Finger-Intervalle für Knoten 1.

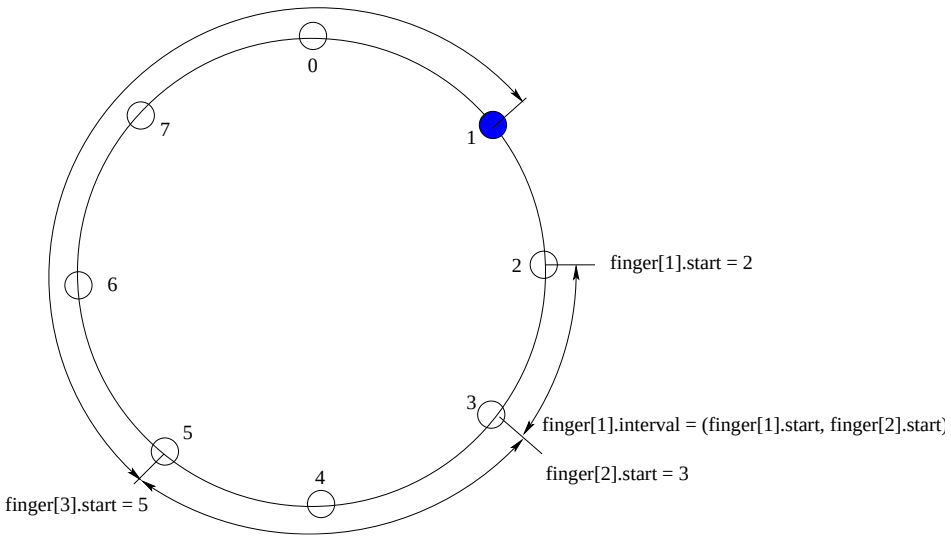


Abbildung 3.8: Intervalle: Der Knoten 1 im virtuellen Ring von Chord hat genau 3 Finger-Einträge, da $m = 3$ gilt.

Die Intervalle beginnen in diesem Beispiel vom Knoten 1 aus, also bei den Plätzen

$(1 + 2^0)\ mod\ 2^3 = 2$, $(1 + 2^1)\ mod\ 2^3 = 3$ und $(1 + 2^2)\ mod\ 2^3 = 5$ im Chord-Ring.

Eine komplette Finger-Tabelle aller Knoten in einem Chord-Ring mit $m = 4$ und drei besetzen Knoten zeigt die nachfolgende Abbildung 3.9. Es sind die Knoten 0, 2 und 7 besetzt.

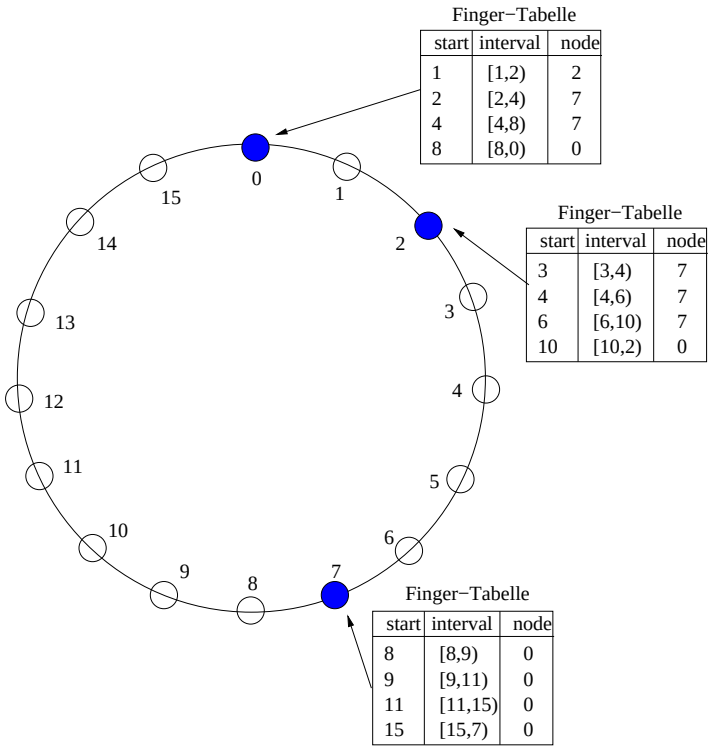


Abbildung 3.9: Finger-Tabellen der Knoten: Die Abbildung zeigt alle Finger aus den einzelnen Finger-Tabellen der drei Knoten an den Positionen 0, 2 und 7. Im Ring mit $m = 4$ muss jeder Knoten eine Finger-Tabelle der Größe 4 verwalten. Dabei können alle Finger einer Finger-Tabelle, wie bei Knoten 7 gezeigt, immer auf den gleichen Knoten zeigen, wenn dieser allen Anfängen der Intervalle nachfolgt.

3.4.3 Suche von Daten

Um zu einem Eintrag den verantwortlichen Knoten, also den Knoten der für den Schlüssel zuständig ist, zu finden, wird zuerst der predecessor des Schlüssels gesucht. Die Suchfunktion des Knotens, an den die Anfrage gestellt wurde, folgt dem Finger aus seiner Routing-Tabelle, 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.

Als Beispiel soll die Abbildung 3.10 dienen. Gesucht wird das Dokument mit dem Schlüssel 1. Die Anfrage wurde an den Knoten 7 geschickt, der die Anfrage an den weitest entferntesten Knoten seiner Finger-Tabelle sendet, der noch vor dem Schlüssel liegt, in dieser Abbildung an den Knoten 0. Dieser Knoten 0 gibt die Anfrage, da er Vorgänger des Schlüssels ist, an den Knoten 2 weiter, der schließlich

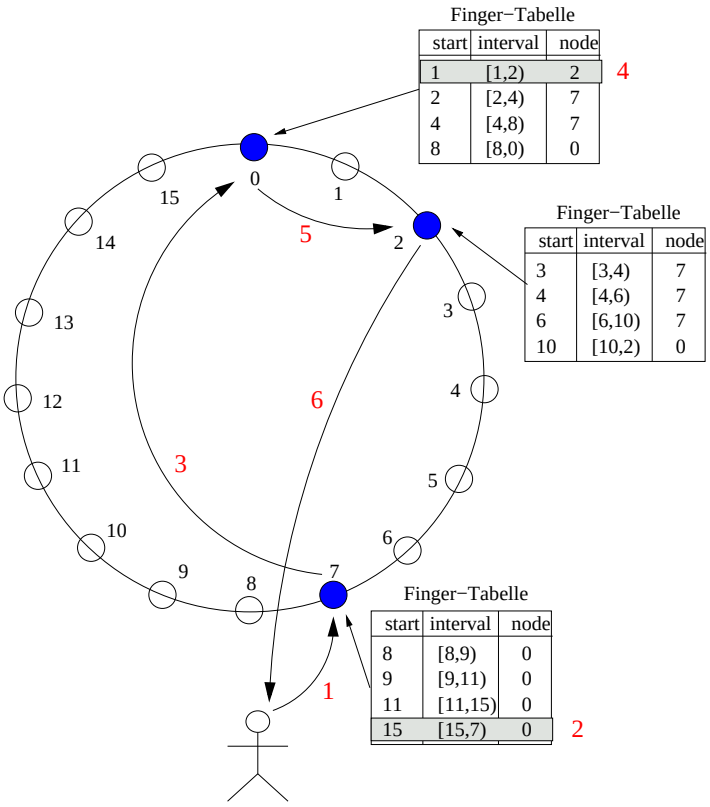


Abbildung 3.10: Suche von Schlüsseln: Der Benutzer sucht ein Dokument mit den Schlüssel 1 und wendet sich an den Knoten 7. Die gefärbten Zahlen 1 bis 6 geben die Reihenfolge bei der Suche an, grau markierte Finger werden zum Suchen verwendet.

das Dokument mit dem Schlüssel 1 an den Absender der Anfrage schicken kann.

3.4.4 Hinzufügen von Knoten

Da in einem dynamischen Peer-to-Peer Netz ständig Knoten dem System beitreten oder es verlassen, müssen effiziente Algorithmen existieren, die solche Fälle behandeln. In diesem Abschnitt wird gezeigt, wie Knoten dem System beitreten können, später wird das Verlassen des virtuellen Chord-Rings behandelt.

Chord benutzt zwei Invarianten, um das System stabil zu halten. Eine Invariante ist ein Bedingung, die vor und nach Ausführung einer Operation oder Aktion im System immer gelten muss.

Die erste Invariante ist, dass jeder Knoten immer seinen richtigen successor kennt. Das soll gewährleisten, dass Chord immer voll funktionsfähig ist. So können durch Kettenabfrage der Nachfolger immer alle fehlenden Routing-Einträge aktualisiert werden.

Die zweite Invariante besagt, dass für jeden Schlüssel k, der Knoten successor(k) immer für k zuständig ist.

Nachdem sich ein neuer Knoten bei einem Knoten des bestehenden Systems angemeldet hat, wird der neue Knoten an den zuständigen successor weitergeleitet, der seinem aus der IP-Adresse berechneten Hashwert nachfolgt.

Um diesen Knoten nun in ein System einzufügen, dass die beschriebenen Invarianten erfüllt, sind drei Schritte notwendig.

- Initialisierung der Finger und des Vorgängers (predecessor) vom neuen Knoten
- Update der Finger und des Vorgängers der existierenden Knoten
- Die höher gelegene Applikation benachrichtigen, dass die Daten vom alten Knoten auf den neuen verschoben werden können.

Nach diesen drei Schritten sind wieder sämtliche Invarianten erfüllt, und sowohl das Chord System, als auch der neue Knoten voll funktionsfähig.

Die folgende Abbildung 3.11 zeigt die Finger-Tabellen, wenn zu den Knoten in Abbildung 3.9 noch Position 12 durch einen Knoten besetzt wird.

In der Abbildung sind alle Finger-Einträge, die sich verändert haben schwarz dargestellt, Einträge die unverändert geblieben sind in grau.

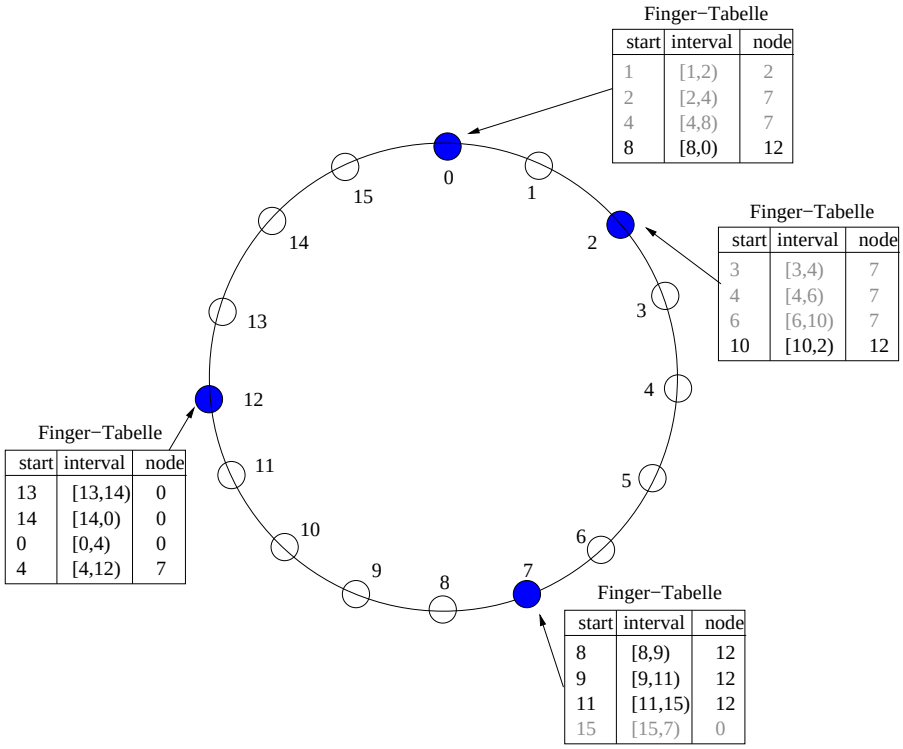


Abbildung 3.11: Hinzufügen eines Knotens: Nach dem Hinzufügen eines Knotens mit dem Hashwert 12, müssen bei den anderen Knoten insgesamt 5 von 12 Finger-Einträgen korrigiert werden.

3.4.5 Stabilisierung

Damit es zu keinem Fehlverhalten kommt, wenn ein Knoten im System ausfällt oder mehrere Knoten gleichzeitig dem Ring beitreten, gibt es zusätzlich zu dem eigentlichen Chord-Protokoll noch Stabilisierungsroutinen.

In gewissen Abständen, die davon abhängen wie oft Knoten dem System beitreten, überprüft jeder Knoten in Chord, ob sein successor wirklich noch aktuell ist. Dazu lässt er sich von seinem successor den predecessor nennen. Entspricht dies nicht dem eigenen Knoten, liegt ein weiterer Knoten zwischen dem Knoten und seinem eigentlich gedachten successor. Dadurch ist dieser Knoten in der Mitte eigentlich der richtige successor. Anschließend schickt er eine Nachricht an die beteiligten Knoten und korrigiert die Finger-Tabelle. In regelmäßigen Abständen überprüfen die Knoten nach dem gleichen Schema ihre Finger in der Routing-Tabelle zufällig auf Korrektheit.

3.4.5.1 Entfernen von Knoten

Damit es nicht zu einem Zusammenbruch des Systems kommt, falls der successor eines Knotens ausfällt, hat jeder Knoten noch eine Liste der nächst folgenden Knoten. Diese kann durch eine Abfrage der successor Knoten entlang des Rings erstellt werden. Falls nun durch die regelmäßigen Abfragen ein Knoten feststellt, dass sein successor ausgefallen ist, kann er ihn durch den ersten Eintrag aus dieser Liste ersetzen und abwarten bis das System sich durch seine oben genannten Stabilisierungsroutinen wieder korrigiert.

3.4.6 Lastverteilung

Die Last in Chord wird nur durch die Hash-Funktion 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 am Ende des kapitels näher beschrieben werden.

3.4.7 Laufzeiten

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

3.4.8 Vorteile gegenüber Consistent Hashing

Im Gegensatz zu Consistent Hashing ist in Chord ein effizientes Suchen nach Daten durch die Finger-Tabellen möglich. Während in Consistent Hashing nur mit Vorgänger und Nachfolger navigiert wird, wird bei Chord der Suchraum in jedem Schritt in etwa halbiert. Chord ist dabei auch fehlertoleranter als Consistent Hashing. Es hat für den Fall, dass Knoten ausfallen eingebaute Mechanismen zur Korrektur.

3.4.9 Einordnung

Die Tabelle zeigt Chord eingeordnet nach den Kategorien aus Abschnitt 3.1 von Seite 10.

Eigenschaften	Chord
Topologie	Ring
Platzieren der Daten	Hashing
Routing	Direkt
Ausdrucksstärke	+
Ergebnismenge	++++
Selbstständigkeit	++
Effizienz	+++
Robustheit	+++

Tabelle 3.5: Einordnung von Chord: Auch Chord wird nach den einzelnen Kategorien aus dem Abschnitt 3.1 bewertet.

Begründung der einzelnen Bewertungen:
Chord erlaubt, wie eigentlich fast alle DHTs, nur die Suche nach einem bestimmten genau angegebenen Schlüssel. Bereichsabfragen sind nicht möglich. Deshalb wird die Ausdrucksstärke mit schlecht bewertet. Chord garantiert dagegen, dass, wenn ein Element vorhanden ist, es auch gefunden wird. Die Knoten sind kaum selbstständig. Sie können bis auf den Wunsch dem System bei- oder auszutreten keine eigenen Entscheidungen treffen. Sie können nicht beeinflussen mit welchen Knoten sie sich verbinden müssen oder wo welche Daten gespeichert werden. Die Effizienz ist sehr gut. Die Nachrichten werden über Routing-Tabellen direkt gesendet. So entsteht wenig Netzverkehr für eine Suche. Chord hat zur Robustheit eingebaute Stabilisierungsroutinen. Wenn ein Knoten ausfällt, sind die Daten aber ohne zusätzliche Maßnahmen nicht mehr im System vorhanden.

3.4.10 Bewertung

Chord erreicht mit seiner Finger-Tabelle und dem geschickten Einteilen des Ringes in Intervalle, das Daten in logarithmischer Zeit gefunden werden. Es ist skalierbarer als Consistent Hashing, und hat trotzdem einen noch zu übersehenden Aufwand bei der Implementierung und ist leicht zu veranschaulichen. Aber Chord muss um Bereichsabfragen erweitert werden, um die Suche noch effizienter gestalten zu können, sowie um einen Lastbalancierungsalgorithmus erweitert werden.

3.5 DKS(N, k, f)

„DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications“ [2] ist eine Generalisation vom vorgestellten Chord. DKS(N, k, f) bildet ebenso wie Chord ein dezentralisiertes Overlay-Netzwerk über ein P2P Netz.

Dabei bezeichnet N im Namen DKS(N, k, f) die maximale Anzahl der Knoten die im Distributed Hash Tables Netzwerk vorhanden sein können. Der Parameter k gibt die Dimension der Suche an und f den Grad der Fehlertoleranz.

Dabei werden im Gegensatz zu Chord Fehlerfälle, wie ausgefallene Knoten, nicht ständig korrigiert. Bei Chord haben die Knoten periodisch ihre Finger in den Routing-Tabellen auf Korrektheit überprüft. Diesen Mechanismus benötigt DKS jedoch nicht. Nicht korrekte Routing-Einträge werden bei der Suche on-the-fly („im Fluge“, also nebenher und automatisch) korrigiert.

Für k = 2 entspricht DKS(N, k, f) dabei bis auf die Fehlerbehandlung dem bereits vorgestellten DHT Chord.

Im Folgendem wird DKS als Abkürzung für DKS(N, k, f) benutzt.

3.5.1 Funktionsweise

DKS erweitert Chord um verschiedene weitere Ansätze. Als erstes wird eine Form des Intervallroutens benutzt, das die Autoren „verteilte k-stufige Suche“ (distributed k-ary search) genannt haben.

Als zweites wird die so genannte „Korrektur bei Verwendung“ (correction-on-use) zum Korrigieren von Einträgen in den Routing-Tabellen benutzt.

Außerdem wird das System in verschiedenen Level und Sichten eingeteilt, und die Routing-Tabellen von Chord an diese angepasst.

3.5.1.1 Levels und Sichten

Jeder Knoten in einem DKS(N, k, f) System hat $\log_k(N)$ Level. Diese werden von 1 bis L durchnummeriert, wobei $L = \log_k(N)$ gilt. Im folgenden bezeichnet L' die Menge $1, 2, \ldots L$. In einem Level $l \in L'$ hat ein Knoten n den View (Sicht) V^l auf den Raum der Schlüssel. Die Sicht V^l enthält k gleiche Teile, die mit I_i^l, wobei $0 \leq i \leq k-1$ gilt, bezeichnet werden und Level für Level wie folgt definiert werden:

Level 1:

$$V_0^1 = I_0^1 \cup I_1^1 \cup \ldots \cup I_{k-1}^1,$$

mit

$$I_0^1 = [x_0^1, x_1^1[,\ I_1^1 = [x_1^1, x_2^1[,\ \ldots,\ I_{k-1}^1 = [x_{k-1}^1, x_0^1[$$

$$x_1^1 = (n + i\frac{N}{k})\ mod\ N,\ 0 \leq i \leq k-1.$$

Level $2 \leq l \leq L$:

$$V_0^l = I_0^l \cup I_1^l \cup \ldots \cup I_{k-1}^l,$$

mit

$$I_0^l = [x_0^l, x_1^l[,\ I_1^l = [x_1^l, x_2^l[,\ \ldots,\ I_{k-1}^l = [x_{k-1}^l, x_1^{l-1}[$$

$$x_1^l = (n + i\frac{N}{k^l})\ mod\ N,\ 0 \leq i \leq k-1.$$

Sei nun n ein Knoten aus dem DKS System mit einem l aus $1 \leq l \leq L$ und sei V^l seine Sicht auf das Level l. Jedes I_j^l hat einen zuständigen Knoten $R(I_j^l)$. Dies ist der Knoten den n fragen würde, wenn eine Key in das Intervall I_j^l fallen würde. Der zuständige Knoten für einen Identifier k ist der Knoten, der im Uhrzeigersinn dem Element folgt. S(x) bezeichnet den ersten Knoten der x nachfolgt, also der erste Knoten, der im Uhrzeigersinn im Intervall $[x, x[$ ist.

3.5.1.2 Routing-Tabelle

DKS speichert wie Chord Routingtabellen zur Suche von Daten und den dazugehörigen Knoten. In einem DKS(N, k, f) System muss k eine natürliche ganze Zahl sein und die Eigenschaft $k \geq 2$ erfüllen. Die maximale Anzahl von Knoten im System beträgt $N = k^L$, wobei L eine genügende große Zahl sein muss, um große Datenmengen speichern zu können. Jedem Knoten im System muss N, k und f bekannt sein, L kann jeder Knoten daraus selber berechnen. Angeordnet werden die Knoten wie bei Chord in einem virtuellen Ring, bezeichnet mit $I = 0, 1, \ldots, N-1$.

Der Parameter f gibt die Fehlertoleranz an, wobei f ein kleiner Faktor von k ist.

Wie bei Chord speichert jeder Knoten eine Routing-Tabelle. In dieser Tabelle speichert jeder Knoten in jedem Level sich selbst und k-1 andere Knoten, insgesamt also $(k-1)\log_k(N)$. Zusätzlich speichert jeder Knoten noch seinen Vorgänger im Ring.

Die Tabelle 3.6 auf der nächsten Seite enthält die Routing-Tabelle vom Knoten n.

3.5.1.3 Verteilte k-stufige Suche

Der Suchalgorithmus von DKS beruht auf der „verteilten k-stufigen Suche“ („distributed k-ary search“) [17]. Für einen gegebenen Schlüssel t können mit dieser Suche die Daten in $\log_k(N)$ Schritten gefunden werden. Am Anfang der Suche entspricht der Suchraum dabei dem ganzen Raum der Schlüssel. Nach jedem Schritt wird der aktuelle Suchraum in k gleiche Intervalle geteilt. Jeder Bereich ist in der Verantwortung eines ausgewählten Knotens. Diese Partitionierung wird so lange durchgeführt, bis k gleiche Teile entstehen, die nur aus einem Element bestehen. In einem Raum der Größe N braucht jede Suchoperation also $\log_k(N)$ Schritte, und jeder Knoten braucht eine Routing-Tabelle mit $(k-1)\log_k(N)$ Einträgen.

Das Protokoll zum Suchen von Schlüsseln in DKS(N, k, l) funktioniert also ähnlich wie bei Chord. Wenn ein Knoten n eine Suche für einen Schlüssel t erreicht, prüft er, ob t zwischen ihm und seinem predecessor liegt. Ist dies der Fall, ist der Knoten n selbst zuständig und gibt den Wert zum Schlüssel t zurück. Ansonsten stößt Knoten n den beschriebenen Suchprozess an, um den Knoten zu finden, der t im Kreis nachfolgt.

Das Einfügen von Objekten ist ähnlich dem des Suchens.

Level	Intervall	Zuständigkeit
l	I_0^1	n
	I_1^1	$S(x_1^1)$
	...	...
	I_{k-1}^1	$S(x_{k-1}^1)$
...	...	...
L-1	I_0^{L-1}	n
	I_1^{L-1}	$S(x_1^{L-1})$
	...	...
	I_{k-1}^{L-1}	$S(x_{k-1}^{L-1})$
L	I_0^L	n
	I_1^L	$S(n+1 \, mod \, N)$
	...	...
	I_{k-1}^L	$S((n+(k-1)) \, mod \, N)$

Tabelle 3.6: Routing-Tabelle: Die Tabelle aus [2] zeigt die Routingeinträge für den Knoten n im Distributed Hash Tables System DKS(N, k, f) in den verschiedenen Leveln .

3.5.1.4 Hinzufügen von Knoten

Beim Einfügen von Knoten sind zwei Fälle zu unterscheiden.

Wenn noch kein Knoten im System vorhanden ist, muss nur die Routing-Tabelle für den ersten Knoten erzeugt werden, wobei er für alle Intervalle selber zuständig ist und sein eigener predecessor ist.

Wenn bereits Knoten vorhanden sind, schickt der einzufügende Knoten q eine Nachricht an einen bekannten Knoten im System. Die Nachricht wird solange weitergeleitet, bis ein Knoten n gefunden wird, der im Kreis q nachfolgen kann. Der Knoten n berechnet nun die Routing-Einträge für den Knoten q und sendet ihm diese. Die Vorgänger des neuen und alten Knotens werden schließlich geändert. Die Routing-Tabelle wird dabei nur berechnet. Es wird keine Suche nach Knoten durchgeführt. So könne die Einträge unter Umständen ungültig sein.

3.5.1.5 Korrektur bei Verwendung

Die „Korrektur bei Verwendungs-Technik“ („correction-on-use“) beruht auf der Annahme, dass nicht aktuelle Routing-Einträge in P2P Systemen normal sind. Deshalb werden die Einträge zum Beispiel beim Suchen on-the-fly gefunden und repariert. Dazu werden in jede Nachricht Informationen eingefügt, die ein Knoten n' von einem Knoten n erhält. Knoten n' kann nun entscheiden ob die Routinginformationen, die Knoten n genutzt hat korrekt waren oder nicht. Wenn n' merkt, dass die Informationen falsch sind, informiert er n und teilt ihm mit, welcher Knoten seiner Meinung nach der mögliche Kandidat für den Routing-Eintrag von n ist. Wenn n eine solche Nachricht erhält, korrigiert er seine fehlerhaften Einträge und schickt die Nachricht nochmal los. So wird jede ungültige benutzte Routinginformation gefunden und eventuell korrigiert. In einem stabilen Zustand kommt das System also ohne zusätzliche Nachrichten aus.

Um Fehlertoleranter zu werden, speichert jeder Knoten n in DKS zusätzlich eine Liste fl_n. Diese enthält die nächsten $(f + 1)$ Knoten, die n nachfolgen. Durch die Applikation werden auch die gesamten Zustände der Knoten gespeichert. Damit können bis zu f Knoten ausfallen und der Zustand kann noch rekonstruiert werden.

Wenn Knoten n nun den Ausfall von m bemerkt und in seiner fl_n enthalten ist, veranlasst er den Nachfolger von m, der mit der gespeicherten Liste einfach gefunden werden kann, die Aufgabe von m zu übernehmen. Daraufhin wird für alle Nachfolger von f die fl_n-Liste überarbeitet. Wenn der Knoten m nicht in der Liste der Nachfolger vorhanden ist, veranlasst er einen Knoten, den er für ihn der Nähe hält, die Korrektur zu übernehmen.

3.5.1.6 Verlassen von Knoten

Wenn der Knoten n das System verlassen will, veranlasst er die Applikationsschicht, die Daten an den Knoten s, dem Nachfolger von Knoten n zu übergeben. Der Knoten n speichert alle eingehenden Suchanfragen etc. zwischen. Sobald die Daten von Knoten n auf s übertragen wurden, sendet s eine Nachricht an n. Dieser schickt nun alle gespeicherten Anfragen an s, und verlässt dann ohne eine weitere Nachricht das System. Das Verlassen von n wird von jedem anderen Knoten dann bemerkt, wenn er versucht mit n zu kommunizieren.

3.5.2 Laufzeiten

In einem Raum der Größe N braucht jede Suchoperation also $\log_k(N)$ Schritte, und jeder Knoten braucht eine Routing-Tabelle mit $(k-1)\log_k(N)$ Einträgen. Zusätzliche Nachrichten zum Aufrechterhalten des Netzwerkes benötigt DKS nicht.

3.5.3 Einordnung

Die Tabelle zeigt DKS(N, k, f) eingeordnet nach den Kategorien in 3.1 von Seite 10.

Eigenschaften	DKS(N, k, f)
Topologie	Ring in mehreren Leveln
Platzieren der Daten	Hashing
Routing	direkt
Ausdrucksstärke	+
Ergebnismenge	++++
Selbstständigkeit	++
Effizienz	+++
Robustheit	+++

Tabelle 3.7: Einordnung von DKS: Einordnung des Distributed Hash Tables System DKS nach dem in Abschnitt 3.1 beschriebenen Kriterien.

Zur Begründung der einzelnen Bewertungen.
DKS hat fast gleiche Bewertungen wie Chord. DKS erlaubt auch keine Bereichsabfragen und garantiert ebenfalls, dass, wenn ein Element vorhanden ist, es auch gefunden wird. Die Knoten sind kaum selbstständig. Sie können nicht beeinflussen mit welchen Knoten sie sich verbinden, oder wo welche Daten gespeichert werden, da sonst die Funktionsweise des Systems gestört ist. Die Effizienz ist gut. Je mehr Level eingeführt werden, um so aufwendiger wird jedoch das Berechnen sämtlicher Einträge. Die Nachrichten werden über Routing-Tabellen direkt gesendet, es entsteht kaum Netzverkehr. DKS hat zur Robustheit die correction-on-use Technik. Zur Wiederherstellung der Daten bei Ausfall eines Knotens kann bestimmt werden, wie viele Kopien eines Elements angelegt werden.

3.5.4 Vergleich und Bewertung

Gegenüber Chord [57] hat DKS(N, k, f) eine correction-on-use Technik. So werden Fehler on-the-fly korrigiert, wenn sie bemerkt werden. Mit seinem distributed k-ary search ist DKS(N, k, f) aber deutlich schwerer zu verstehen und zu implementieren als Chord. Meist wird daher Chord als Distributed Hash Table in einem Peer-to-Peer Netz ausreichen.

3.6 Weitere DHTs

Es entstehen zur Zeit ständig neue Distributed Hash Tables. Im Folgenden sollen deshalb nur kurz einige weitere DHTs mit ihrer Grundidee erwähnt werden, die beim Vergleich der DHTs nicht berücksichtigt wurden.

Kademlia [29, 30] ist eine DHT, die auf der so genannten XOR-Metric beruht. sie soll im Overnet Netz eingesetzt werden [65]. Overnet unterstützt das Filesharing zwischen verschiedenen Benutzern in einem P2P Netzwerk. Filesharing ist der Austausch und das Anbieten von Dateien, zum Beispiel mit anderen Benutzer eines P2P Systems.

Jeder Knoten wählt sich dabei eine zufällige 160-bit-Zahl als Identifizierer im Netz. Zu speichernde Dokumente werden ebenfalls mit einer solchen Zahl versehen. Kademlia benutzt zur Suche von Daten die XOR-Metric. Dabei lässt sich der Abstand $d(x, y)$ zwischen zwei Identifizieren x und y durch $d(x, y) = xXORy$ berechnen. Jeder Knoten speichert für jedes $0 \leq i < 160$ eine Liste von k Knoten, die mit ihm Kontakt getreten sind und einen Abstand zwischen 2^i und 2^{i+1} haben. Jede dieser einzelnen Listen wird ein k-bucket genannt. Diese k-buckets werden nach dem Zeitpunkt, wann das letzte Mal Kontakt mit den Knoten bestand, gespeichert. Anfragen an Knoten, Daten zu speichern oder zu suchen, werden dann an verschiedene bekannten Knoten in der Nähe parallel gesendet.

„Eine einfache DHT“ ist ein weiterer neuer Vertreter der Distributed Hash Tables. Die von den Autoren in [37] als „A Simple Fault Tolerant Distributed Hash Table“ („Eine einfache fehlertolerante DHT“) bezeichnete DHT, benutzt überlappende DHTs.

Weiterhin existieren DHTs die mit de Brujin Graphen [16] funktionieren.

Koorde [23] verbindet Chord [57] und die de Brujin Graphen. In einem de Brujin Graph hat jeder Knoten ein binäre Nummer mit b Bits. Jeder Knoten m speichert in Koorde seinen Nachfolger und den ersten de Brujin Knoten. Dieser ist der erste Knoten der $2m \, mod \, 2^b$ nachfolgt. Durch Shiften der Adressen beim Suchen kann der Knoten dann im de Brujin-Graph in $O(\log n)$ gefunden werden, und jeder Knoten hat nur 2 Nachbarn.

Aber die de Brujin Graphen werden nicht nur für Erweiterungen von Chord benutzt. So werden in **D2B** [18] diese de Brujin Graphen benutzt um CAN [44] zu erweitern.

Viceroy [28] hingegen möchte einen Schmetterling emulieren. Die Knoten werden in verschiedenen Level angeordnet. Alle Knoten sind über Vorgänger und Nachfolger mit 2 Knoten aller Level und 2 Knoten ihres Levels verbunden. Weiterhin ist jeder Knoten mit 2 Knoten des nächst höheren Levels und einem Knoten des niedrigeren Levels verbunden. Durch geschicktes ablaufen der Verbindungen, können Elemente ebenfalls in $O(\log n)$ gefunden werden, wobei jeder Knoten im Schnitt 7 Nachbarn hat.

3.6.1 Bemerkung

Es werden immer mehr verschiedene Distributed Hash Tables entwickelt. Die meisten versuchen Chord oder CAN weiter zu verfeinern, um sie zum Beispiel effizienter oder Fehlertoleranter zu machen. Es bleibt abzuwarten, welche Distributed Hash Tables

Systeme sich durchsetzen werden. In den vorangegangenen Abschnitten wurden deshalb nur die wichtigsten und grundlegenden DHTs vorgestellt.

3.7 Vergleich der DHTs

Folgende Tabelle 3.8 gibt nun einen kurzen Gesamtvergleich aller vorgestellten DHTs. Es wird das Modell, die Laufzeit und die Anzahl der Nachbarn für die DHTs gegenübergestellt.

Algorithmus	**Modell**	**Aufwand beim Suchen**	**# Nachbarn**
Chord	ein-dimensionaler Ring	$O(\log(N))$	$\log(N)$
CAN	D-dimensionaler Torus	$O(D \cdot N^{\frac{1}{D}})$	$2D$
DKS(N, k, f)	ein-dimensionaler Ring in mehreren Leveln	$O(\log_k(N))$	$(k-1)\log_k(N)$
Consistent Hashing	ein-dimensionaler Ring	$O(\frac{N}{2})$	2

Tabelle 3.8: Vergleich: Vergleich der DHT Systeme Chord, CAN, DKS(N, k, f) und Consistent Hashing. N ist die Anzahl der Knoten im System, D die Dimension im CAN System.

Die Tabelle 3.9 zeigt abschließend alle Systeme eingeordnet nach den Kategorien aus Abschnitt 3.1 von Seite 10.

Es ist zu sehen das die DHTs CAN, Chord und DKS etwa gleiche Eigenschaften haben. Gnutella und Consistent Hashing liegen vor allem in der Effizienz deutlich zurück.

3.8 Offene Fragen über DHTs

Es gibt aber noch einige weitere Fragestellungen in Bezug auf DHTs, die noch nicht gelöst sind.

Ein Beispiel sind die in [45] gestellten, offenen Fragen. Eine ist zum Beispiel, ob es ein DHT System gibt, indem jeder Knoten nur $O(1)$ Nachbarn hat, und die Pfadlänge trotzdem bei der Suche in diesem DHT System in $O(\log(N))$ liegt (siehe auch Tabelle 3.8).

Dies wäre ein optimales DHT System ohne zentralen Server, denn die Pfadlänge bei der Suche und der Aufwand beim Hinzufügen von Knoten wäre minimal. Wenn ein solches System gefunden wird, ist die nächste Frage, ob dann andere Faktoren existieren, die diese Lösung wieder schlechter als andere machen.

Eine weitere Fragestellung befasst sich mit der Lokalität [25]. Wie können in einem 1-dimensionalen Schlüsselraum die Schlüssel auf die Knoten nach geographischen Gesichtspunkten angeordnet werden, und wird dadurch die Robustheit, der Versuch der Vermeidung von Hotspots oder ähnliches, die durch DHTs erreicht werden, wieder aufgehoben?

Eigenschaften	Gnutella	CAN	Consistent Hashing	Chord	DKS (N,k,f)
Topologie	zufällig	d-dimensionaler Torus	Ring	Ring	Ring in mehreren Leveln
Platzierung der Daten	beliebig	Hashing	Hashing	Hashing	Hashing
Routing	Fluten	Direkt	Ringsuche	Direkt	Direkt
Ausdrucksstärke der Suche	++++	+	+	+	+
Ergebnismenge der Suche	++	++++	++++	++++	++++
Selbstständigkeit	++++	++	++	++	++
Effizienz	+	+++	+	+++	+++
Robustheit	++	+++	++	+++	+++

Tabelle 3.9: Vergleich aller vorgestellten DHTs: Die Tabelle zeigt abschließend alle Distributed Hash Tables Systeme eingeordnet nach den Kategorien aus Abschnitt 3.1 von Seite 10. Dazu wurden sämtliche Tabellen über die Einordnung der einzelnen DHTs aus diesem Kapitel in einer Tabelle zusammenfasst.

3.9 Nachteile von DHTs

In diesem Abschnitt sollen abschließend einige Schwachstelle der DHTs aufgezeigt werden.

3.9.1 Lastbalancierung mit DHTs

Die Lastbalancierung nur mit den Standardmechanismen von DHTs reicht nicht aus.

Die Last wird nur durch die Hashfunktion verteilt. Damit sollen die einzelnen Elemente gleichmäßig im gesamten Intervall verteilt werden. Dabei kann aber nicht berücksichtigt werden, dass zum Beispiel einzelne Knoten populäre Elemente haben, die viel nachgefragt werden und so eine große Last haben.

Außerdem wird nicht berücksichtigt, dass die Knoten unterschiedliche Rechner sind, die sich vielleicht stark im Speicherplatz etc. unterscheiden und nicht jeder Knoten gleich viele Elemente verwalten kann.

Auch wird nicht verhindert, dass die Intervalle untereinander stark in der Länge variieren, die die Knoten verwalten müssen.

Wie auch später bei der Vorstellung existierenden Lastbalancierungsalgorithmen oder in der Simulation gezeigt wird, haben die Knoten deshalb stark unterschiedlich viele Dokumente gespeichert.

3.9.2 Suche mit DHTs

Neben dem erwähnten Nachteil der Lastbalancierung haben DHTs vor allem den Nachteil, dass keine effizienten Bereichsabfragen [20] möglich sind.

Um alle Einträge, die zum Beispiel mit a anfangen, zu finden, müssen alle Knoten abgefragt werden. Durch die Hashfunktion werden diese Einträge auf verschiedene Knoten verteilt, ohne das es möglich ist vorherzusagen, wieviele Elemente in jedem 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 einen hohen Aufwand bedeutet, müssen alle Knoten abgefragt werden. Dazu muss zum Beispiel in Chord der Ring einmal komplett abgelaufen werden.

Ein Nachteil ist weiterhin, dass zum Suchen der Name unter dem Eingefügt wurde komplett bekannt sein muss, da sonst das Element nicht gefunden werden kann.

3.10 Fazit

Distributed Hash Tables (DHT) sind eine Methode, um globale Informationen persistent in einem Peer-to-Peer System speichern zu können. Eine DHT bildet dabei eine über mehrere Rechner verteilte Hashtabelle. Der Wertebereich der Hashfunktion, welche die zu veröffentlichenden Einträge auf Werte abbildet, wird in Abschnitte aufgeteilt, die einzelnen Knoten zugeteilt werden. Jeder Knoten ist dann dafür zuständig die Anfragen zu bearbeiten, die in seinen Wertebereich fallen.

Üblicherweise hat ein Knoten in einem DHT System mit N Knoten $O(1)$ oder $O(\log(N))$ Nachbarn.

Es werden zusätzlich bessere Methoden (zum Beispiel Routing-Tabellen) als in klassischen Peer-to-Peer Systemen angewendet, um Daten in einem Peer-to-Peer System zu finden. Klassische Peer-to-Peer Systeme haben ausgezeichnete Knoten (zum Beispiel Server) oder fluten die Netze mit Anfragen komplett.

Die Lastbalancierung nur mit einer Hashfunktion und die fehlenden Bereichsabfragen dieser Systeme reicht aber nicht aus. Deshalb werden im nächsten Kapitel einige existierenden Lastbalancierungsalgorithmen für Peer-to-Peer Systeme vorgestellt.

Das Nicht-Wahrnehmen von etwas beweist nicht dessen Nicht-Existenz.

– Tenzin Gyatso, XIV. Dalai Lama

Kapitel 4

Existierende Lastbalancierungsalgorithmen

In diesem Kapitel sollen verschiedene vorhandene Algorithmen zur Lastbalancierung in P2P Systemen untersucht und bewertet werden.

Wie in letztem Kapitel über Distributed Hash Tables gezeigt, reicht die Lastbalancierung dieser Distributed Hash Table Systeme mit Hilfe der Hashfunktion alleine nicht aus. Es müssen also Lastbalancierungsalgorithmen entworfen werden, die Last zwischen den Knoten besser verteilen. Da im Peer-to-Peer Ansatz keine globale Sicht existiert müssen diese Algorithmen lokale Strategien zum Verteilen der Last in einem DHT anwenden.

Dabei ist aber darauf zu achten, dass die Daten nicht einfach zwischen den Knoten in einem DHT verschoben werden können, da sonst die Suche in diesem Distributed Hash Table System unter Umständen nicht mehr richtig funktioniert.

Dabei können unter dem Begriff Last verschiedene Parameter verstanden werden. Es kann sowohl die Prozessorlast, die durch Anfragen innerhalb eines Zeitraums entsteht, als auch der benötigte Speicherbedarf durch die Daten sein. Diese Last soll dabei möglichst auf alle Rechner gleich verteilt werden. Die meisten Algorithmen beschreiben ein allgemeines Verfahren, dass zur Lastbalancierung eingesetzt werden kann und spezifizieren nicht wie die Last gemessen oder bestimmt wird. Meist wird daher die Last anhand der gespeicherten Daten pro Knoten bestimmt.

Abschnitt 4.2 zeigt einen Algorithmus, der virtuelle Server[42] benutzt, um die Last auf die Knoten zu verteilen. Die Idee dabei ist, dass ein Knoten mehrere Intervalle speichert und für diese zuständig ist.

Der Algorithmus Power of two Choices [9] in Abschnitt 4.3 beruht auf einem einfachen Prinzip. Für jeden zu speichernden Datum werden zwei oder mehr verschiedene Knoten durch mehrere Hashfunktionen ermittelt. Der Eintrag wird nun in dem Knoten gespeichert, der die geringere Last hat.

Anschließend wird im Abschnitt 4.4 ab Seite 46 ein System mit autonomen Agenten [33] vorgestellt. Die autonomen Agenten, die wie Ameisen das Peer-to-Peer Netz durchlaufen, verteilen Informationen über die Last der Knoten an andere Knoten im Peer-to-Peer System. Dieser Algorithmus benutzt als einziger kein Distributed Hash Tables System, in dem die Last verteilt wird.

Nachdem die Algorithmen vorgestellt wurden werden diese verglichen und bewertet.

4.1 Last

Zuerst soll in diesem Abschnitt festgelegt werden wann in den folgenden Kapiteln ein Knoten als überlastet gilt und wann das System optimal balanciert ist.

Dabei können unter dem Begriff Last verschiedene Parameter verstanden werden. Es kann sowohl die Prozessorlast, die durch Anfragen innerhalb eines Zeitraums entsteht, als auch der benötigte Speicherbedarf durch die Daten sein. Diese Last soll dabei möglichst auf alle Rechner gleich verteilt werden. Meist wird die Last jedoch anhand der gespeicherten Daten pro Knoten bestimmt.

Im Folgenden ist die Last eines Knotens die Summe der in diesem Knoten gespeicherten Daten (zum Beispiel Dateien oder Dokumente).

Die Gesamtlast eines P2P Systems ist die Summe der Lasten aller Knoten, die sich im P2P System befinden.

Die Last in einem P2P System mit N Knoten ist optimal balanciert oder verteilt wenn die Last in jedem Knoten des Systems genau $1/N$ der Gesamtlast entspricht.

Ein Knoten ist überlastet oder schwer wenn er mehr Last hat als er bei optimaler Verteilung der Last hätte. Ein Knoten ist unterbeschäftigt oder leicht wenn er weniger Last als bei optimaler Verteilung hat.

4.2 Virtuelle Server

Der erste Algorithmus, der in dieser Arbeit vorgestellt werden soll, ist ein Verfahren von Ananth Rao et al. von der Universität Berkeley.

In diesem Verfahren werden so genannte virtuelle Server [42] benutzt. Dabei verhält sich ein Knoten in diesem Verfahren wie mehrere Knoten in einem der vorgestellten Distributed Hash Tables Systemen.

4.2.1 Idee

Die Grundlegende Idee des Verfahrens ist es, einem Knoten mehrere Intervalle eines Distributed Hash Tables System bearbeiten zu lassen. Ein Knoten in diesem modifizierten System verhält sich also wie mehrere verschiedene Knoten eines Standard DHTs.

Ein virtueller Server wird vom unterliegenden Distributed Hash Tables System als eigenständiger Knoten angesehen. Mehrere virtuelle Server können sich aber auf einem physikalischer Knoten befinden.

In Chord wäre dann zum Beispiel ein virtueller Server für ein Intervall zuständig, der dazugehörige Knoten kann aber mehrere virtuelle Server, und damit verschiedene Intervalle bearbeiten, ohne dass die Intervalle zusammenhängend sein müssen.

Der Hauptvorteil ist, dass die virtuellen Server von Knoten zu anderen Knoten leicht verschoben werden können. Dieser Vorgang ist für das Distributed Hash Table System wie ein Verlassen und Hinzufügen eines Knotens, das alle DHT Systeme

unterstützen. Dabei kann die Last in Form von Intervallen an viele verschiedene Knoten abgegeben werden, denn jeder Knoten kennt alle Nachbarn der virtuellen Server die er verwaltet, bei Chord also sämtliche Finger in den Routing-Tabellen.

4.2.2 Schwere und leichte Knoten

Um die Knoten zu klassifizieren zu können, wird eine Aufteilung der Knoten in „schwere“ und „leichte“ Knoten vorgenommen.

Im folgenden ist L_i die Last des Knotens i. Diese Last L_i ist die Summe aller Lasten aller virtuellen Server die Knoten i gespeichert hat.

Jeder Knoten hat eine Ziellast (Target load) T_i, die vorher festgelegt wird.

Ein Knoten i ist nun **schwer** (heavy) , wenn $L_i > T_i$ gilt, ansonsten ist er **leicht** (light).

Das Ziel ist es nun, so wenig schwere Knoten im Distributed Hash Tables System wie möglich zu haben, indem Last von schweren Knoten in Form von ganzen virtuellen Servern auf leichte Knoten abgegeben wird.

4.2.3 Transfer von virtuellen Servern

Zum Ausgleich der Last zwischen den Knoten werden ganze virtuelle Server von schweren Knoten zu leichten anhand von drei Regeln transferiert.

Zu einem schweren Knoten h (heavy) und einen leichten Knoten l (light), ist der **beste** virtuelle Server v, der von h zu l transferiert werden kann, derjenige der folgende drei Bedingungen erfüllt:

- Der Transfer vom virtuellen Server v vom Knoten h zu Knoten l, macht Knoten l nicht schwer.
- Der virtuelle Server v ist der leichteste virtuelle Server der h leicht macht
- Wenn kein virtueller Server v existiert, durch dessen Transfer h leicht wird, wird der schwerste virtuelle Server von h zu l transferiert.

4.2.4 Varianten

Es werden im Folgenden drei verschiedene Varianten vorgestellt wie die Last durch die virtuellen Servern verteilt werden kann.

In allen drei Varianten wird die Last dadurch verteilt, dass überlastete Knoten ihre virtuellen Server an diejenigen abgeben, die weniger Last haben.

Der Unterschied bei den Varianten liegt darin, wieviele Informationen ein Knoten benötigt, um einen solchen Transfer durchzuführen.

4.2.4.1 One-to-One

In dieser Variante wird die Last zwischen zwei zufällig gewählten Knoten ausgeglichen. Die leichten Knoten wählen zufällig einen Knoten aus dem Chord-Ring aus.

Treffen die leichten Knoten so zufällig auf einen schweren Knoten, findet ein Transfer eines virtuellen Servers nach den oben beschriebenen Regeln statt.

Der Vorteil ist, dass wenn alle Knoten schwer sind, keine zusätzlichen Nachrichten versendet werden müssen.

4.2.4.2 One-to-Many

Beim One-to-Many Schema werden die Daten eine schweren Servers an einen leichten Knoten abgegeben. Dieser leichte Knoten wird aber aus mehreren leichten Knoten direkt ausgewählt.

Der Knoten h sei ein schwerer Knoten, und die Menge $l_1, l_2, \ldots, l_k$ leichte Knoten die h bekannt sind. Für jedes Paar (h, l_i) mit $1 \leq i \leq k$ wird nach dem beschriebenen Regeln ein virtueller Server bestimmt, der transferiert werden könnte.

Aus der Menge dieser virtuellen Server wird schließlich der leichteste virtuelle Server ausgewählt und transferiert, der h leicht macht.

Es wird also nur ein virtuelle Server verschoben, die Auswahl für den schweren Knoten ist aber größer als beim One-to-One Schema und so die Wahrscheinlichkeit höher einen leichten Knoten zu finden, der einen virtuellen Server abnehmen kann.

4.2.4.3 Many-to-Many

Beim Many-to-Many Schema wird die Last von mehren schweren Knoten auf mehrere leichte Knoten verteilt. Die Menge der Knoten, die am Austausch beteiligt sind, wird zufällig bestimmt.

Ein Knoten i wählt eine beliebige Anzahl von Knoten aus und schickt ihnen seine Informationen über seine Lastsituation, und erwartet dann deren Lastsituation.

Anschließend wartet der Knoten bis er von genügend vielen Knoten deren Lastsituation erhalten hat. Nun kann er nach einem Algorithmus berechnen welche Server transferiert werden müssen, und leitet daraufhin einen Austausch der Last zwischen diesen Knoten ein.

Dieser Algorithmus besteht aus drei Schritten.

Als erstes gibt jeder schwere Knoten solange seine virtuellen Server in einen Pool ab bis er leicht ist.

In der nächsten Phase wird rundenweise immer der schwerste virtuelle Server aus dem Pool an den leichten Knoten abgegeben, der mit dem zusätzlichen virtuellen Server am besten an seine gewünschte Ziellast T kommt. Dies wird solange fortgeführt, bis der Pool leer ist oder kein virtueller Server mehr transferiert werden kann, da keine leichten Knoten mehr die restlichen virtuellen Server aufnehmen können, ohne selbst schwer zu werden.

Nun können nicht transferierte virtuelle Server an den Knoten zurückgegeben werden von dem er kommt. Dann kann in der nächsten Runde, wenn andere Knoten gewählt werden, vielleicht die Last abgegeben werden.

Als Alternative kann zusätzlich eine „Vertreibungs-“ (dislodge) Phase mit den restlichen virtuellen Servern eingeleitet werden. In dieser dritten Phase wird der größte Server v aus dem Pool mit einem anderen leichteren virtuellen Server v’ eines leichten

Knotens getauscht. Dabei darf der Knoten aber nicht schwer werden, und v muss schwerer sein als v'. Es wird immer der virtuelle Server von den Knoten genommen, der den leichtesten Server beim Tausch abgeben würde.

Gibt es schließlich keinen Knoten mehr, so dass virtuelle Server verschoben werden können terminiert der Algorithmus. Die übrigen Server behalten dann ihre virtuellen Server.

4.2.5 Ergebnisse

In diesem Abschnitt werden Ergebnisse gezeigt, die bei der Simulation des oben beschriebenen Ansatzes der virtuellen Server entstanden sind und anschließend diskutiert. Der Algorithmus der virtuellen Server wurde von den Entwicklern bisher nur simuliert, eine Implementierung existiert noch nicht.

Die folgende Abbildung 4.1 zeigt das Simulationsergebnis bei einer vorherigen Pareto-Verteilung der Daten. Sie zeigt die verschobene Last (als Anteil der gesamten Last im System) in Abhängigkeit von dem Verhältnis Last zu Ziellast. Dies bedeutet, dass die Last mit einem Faktor von zum Beispiel 0,8 vom idealen Zustand verteilt ist.

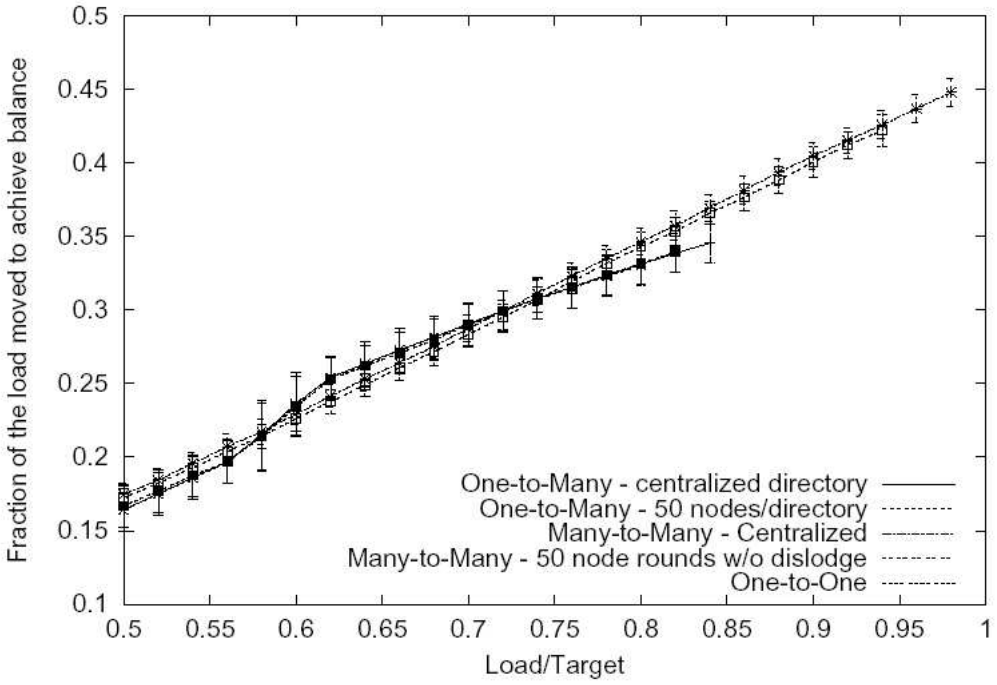


Abbildung 4.1: Verschobene Last: Die Grafik gibt den Anteil der verschoben Last an der gesamten Last beim vorgestellten Lastbalancierungsalgorithmus mit virtuellen Server an [42].

Genauer entspricht diese Verteilung $1 - \delta$, die als

$$T_i = (\frac{\bar{L}}{\bar{C}} + \delta)C_i$$

definiert ist. Dabei ist C_i die Kapazität und L_i die Summe der Lasten aller virtuellen

Server von einem Knoten i.

$$\bar{L} = (\sum_{i=1}^{N} L_i)/N$$

ist die durchschnittliche Last über N Knoten im System. Weiterhin gilt für die durchschnittliche Kapazität

$$\bar{C} = (\sum_{i=1}^{N} C_i)/N.$$

Es wurden insgesamt fünf Ausprägungen der drei vorgestellten Hauptvarianten verglichen. Ein Punkt wurde nur eingetragen wenn alle Simulationen in einem Scenario endeten, indem alle Knoten leicht sind. In allen Varianten wird dabei fasst immer die gleiche Last verschoben.

Nur mit den Many-to-Many Schema wird eine Verteilung erzeugt, die in etwa 97% vom idealen Zustand entfernt ist. Dabei wird aber nahezu 45% der gesamten Last verschoben. Mit den anderen Verfahren wird nur höchstens eine ideale Verteilung von ca. $80 - 84\%$ erreicht. Die Performance von der Many-to-Many Variante ist besser, weil die Chance höher ist, dass ein sehr leichter Knoten einen virtuellen Server von einem schweren Knoten bekommt, der ihn nahe an die Ziellast bringt.

Die folgende Abbildung 4.2 zeigt die Anzahl der Runden die benötigt werden, um eine bestimmte Last/ Zielverteilung zu erreichen.

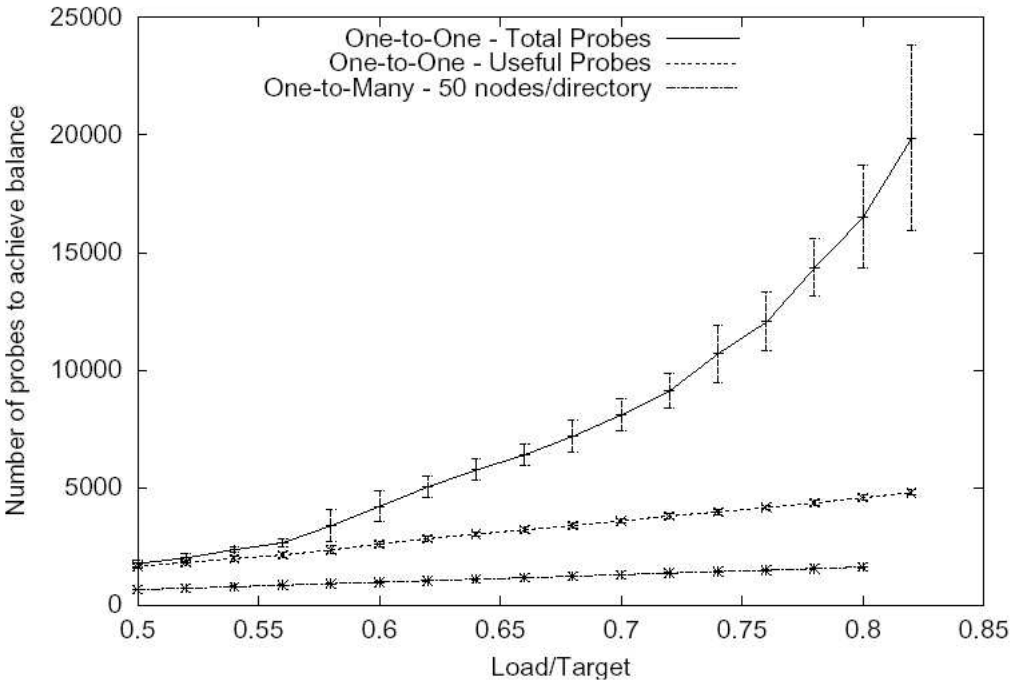


Abbildung 4.2: Virtuelle Server: Die Anzahl der Runden die benötigt werden alle Knoten leicht zu machen [42]

Es werden für one-to-one oder one-to-many im Durchschnitt ca. 1000 Runden benötigt, damit alle Knoten leicht sind und die Last mit 50% von der Idealverteilung abweicht. Bei einer Idealverteilung von 80% werden schon 20.000 Runden benötigt,

wovon nur in ca 5.000 eine Kombination gefunden wurde, die Hilft die Last zu verteilen.

Weiterhin wird von den Autoren gezeigt, dass sich bei many-to-many der Mehraufwand beim dislodgen nicht lohnt. So gibt es kaum Unterschiede zu der Variante ohne diesen Zusatzschritt.

4.2.6 Bewertung

Mit dem Prinzip der virtuellen Server werden mehr Knoten simuliert als tatsächlich physisch vorhanden sind, um den Raum besser aufteilen zu können, und kleinere Einheiten zur Lastbalancierung verschieben zu können.

Aber dies löst das Problem der Lastbalancierung nicht vollständig. Hauptkritikpunkt ist der große Aufwand den ein Knoten leisten muss, um alle seine Routinginformationen in den Finger-Tabellen etc. konsistent zu halten.

Sei N die Anzahl der Knoten im Netzwerk. Wenn die vorgeschlagene Anzahl von $O(logN)$ virtuelle Server mit zum Beispiel jeweils $O(logN)$ Einträgen in den Finger-Tabellen im Falle von Chord verwendet wird, verwaltet jeder Knoten $O(log^2N)$ Einträge. Bei einer Anzahl von nur 100.000 Knoten (was für große Peer-to-Peer Systeme wenig ist) verwaltet jeder Knoten 400 Einträge.

Diese 400 Einträge können intern gut verwaltet werden, es wird aber ein großer Datenverkehr nötig, um alle Einträge konsistent zu halten. So entsteht alleine durch das periodische überprüfen der Einträge in Chord ein regelmäßiger starker Netzverkehr.

Außerdem wird mit dem System nahezu 50% der gesamten Last verschoben. Bei großen Peer-to-Peer Systemen bedeutet die Verschiebung von so vielen Daten eine starke Beanspruchung des Verbindungnetzes.

4.3 Power of two Choices

Der zweite vorgestellte existierende Algorithmus in diesem Kapitel ist der Algorithmus „Power of two Choices“ („Macht der zwei Möglichkeiten“). Er wurde von John Byers, Jeffrey Considine und Michael Mitzenmacher an den Universitäten von Boston und Cambridge entworfen [9].

Die Idee ist, einfach mehrere Hashfunktionen zu benutzen, um die Dokumente in einem Distributed Hash Tables System zu speichern. Diese Methode, die zur Verteilung der Last in Distributed Hash Tables verwendet wird, wurden schon vorher in verschiedenen Standard Hashing Anwendungen benutzt, um die Last zu balancieren.

4.3.1 Idee

Der Algorithmus Power of two Choices beruht darauf, dass es zwei oder mehr Hashfunktionen gibt.

Beim Einfügen und Suchen von Einträgen werden alle Hashfunktionen nacheinander berechnet. Beim Einfügen wird das Dokument dann in dem Knoten gespeichert, der die geringere Last hat.

Mit Hilfe von [5, 32] kann für diese Idee mathematisch, die mit hoher Wahrscheinlichkeit zu erwartende höchste Last von $\log\log N/\log d+O(1)$ gezeigt werden. Dabei ist d die Anzahl der verwendeten Hashfunktionen, und N die Anzahl der Knoten im Distributed Hash Tables System.

Beim Suchen werden alle Knoten parallel angefragt, wobei der Knoten, der den Eintrag gespeichert hat, das Ergebnis schließlich zurückliefert.

4.3.2 Funktionsweise

Es wird im Folgenden gezeigt, wie Power of two Choices in Distributed Hash Tables System Chord angewandt werden kann.

Sei h_0 eine universelle Hashfunktion die Knoten auf den Ring abbildet. Mit Hilfe dieser Hashfunktion wird allen Knoten, wie auch im ursprünglichen Chord vorgesehen, ein fester Platz im Chord-Ring zugeteilt.

Weiterhin seien $h_1, h_2, \ldots, h_d$ ebenfalls universelle Hashfunktionen, die Einträge auf den virtuellen Ring abbilden.

Wenn nun ein Knoten das Element x einfügen will, werden von diesem Knoten sämtliche Ergebnisse der d Hashfunktionen $h_1(x), h_2(x), \ldots, h_d(x)$ berechnet. Um das Einfügen schneller zu bearbeiten zu können, sollten die Funktionen eventuell parallel berechnet werden. Daraufhin werden d parallele Suchanfragen nach den zuständigen Knoten in Chord gestartet, das heißt für jeden durch $h_1(x), h_2(x), \ldots, h_d(x)$ berechneten Punkt im Chord-Ring wird der successor gesucht. Jeder dieser gefundenen Knoten liefert an den ursprünglichen Knoten seine Last zurück. Gespeichert wird nun x in dem Knoten, der die geringste Last hat.

Damit beim Suchen nach dem Dokument x nicht wieder alle Knoten parallel angefragt werden müssen, speichert jeder berechnete Knoten einen Zeiger (Pointer). Damit können alle Knoten, die für dieses Dokument x, durch eine der d Hashfunktionen zuständig sind, direkt ihre Anfragen weiterleiten. Nun muss die Anfrage nur an einen beliebigen der berechneten Knoten erfolgen, und dieser sendet, wenn er die Daten nicht selber gespeichert hat, die Anfrage mit Hilfe seines Pointers an den Knoten weiter, der das Element gespeichert haben muss.

Dabei wird bei diesem Ansatz ein so genannter soft state Ansatz [10] gewählt. Der Besitzer eines Schlüssels muss diesen periodisch wieder ins System einfügen, um Fehler zu verhindern. Wenn nun ein Knoten mehr Last bekommt, wird dadurch das Element auch auf Knoten mit weniger Last verschoben.

Alternativ könnten die Knoten die wenig Last haben auch eine Kopie des Elements speichern, um Anfragen effizienter beantworten zu können und die anderen Knoten dadurch zu entlasten.

4.3.3 Ergebnisse

In diesem Abschnitt werden Ergebnisse gezeigt, die bei der Simulation des oben beschriebenen Ansatzes Power of two Choices in [9] entstanden sind. Diese werden mit den virtuellen Servern verglichen und anschließend diskutiert.

Die Simulation wurde mit 10^4 Knoten durchgeführt. Diese Knoten hatten zwischen 10^5 und 10^6 Elemente gespeichert. Es werden im folgenden die Ergebnisse von

- 1) $\lfloor log_2 n \rfloor$ virtuellen Servern,
- 2) einer beliebig großen Anzahl von virtuellen Servern und
- 3) dem Power of two Choices Schema (mit d=2, also 2 Hashfunktionen)

verglichen wobei alle Versuche 10^4 mal wiederholt wurden.

Die Abbildung 4.3 zeigt die minimale, mittlere und maximale Last bei verschiedenen Strategien.

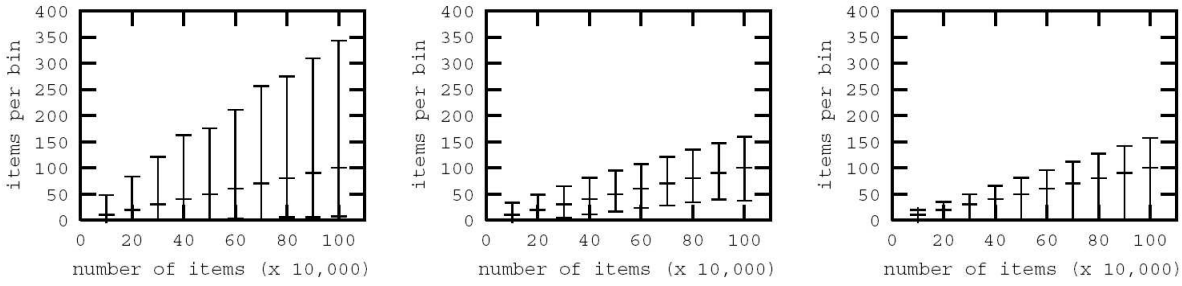


Abbildung 4.3: Power of two Choices: Minimale, ideale und maximale Last bei verschiedenen Lastbalancierung Strategien in Chord [9].

Die Last bei Power of two Choices wird in etwa gleich verteilt, wie bei einer unlimitierten Anzahl von virtuellen Knoten. Es haben aber immer noch Knoten keine Last, während bei anderen Knoten viel Last gespeichert wird.

4.3.4 Bewertung

In Power of two Choices hat jeder Knoten Verweise auf die Elemente in anderen Knoten gespeichert. So hat jeder Knoten unter Umständen riesige Pointerlisten zu verwalten. Insgesamt wird zu jedem Element im System zusätzlich d-1 Pointer im Distributed Hash Table System gespeichert, wobei d die Anzahl der gewählten Hashfunktionen ist. Insgesamt werden also bei n Elementen im System, n Einträge und $(d-1) \cdot n$ Pointer gespeichert. Durch diese zusätzliche Pointer, hat jeder Knoten wiederum deutlich mehr Daten zu verwalten, und seine Last steigt unter Umständen. Wenn, um die Last weiter verteilen zu wollen, die Anzahl der gewählten Hashfunktionen um eins erhöht wird, muss für jedes Element im System ein weiterer Verweis in einem anderen Knoten angelegt werden.

Außerdem sieht das Verfahren ein ständiges wiederholtes Einfügen der Dokumente vor, damit einzelne Knoten nicht zu viel Last speichern müssen. Bei großen Mengen von Dokumenten verursacht dies einen starken Netzverkehr und hohen Aufwand der Knoten.

Wenn bei diesem Verfahren mittels d Hashfunktionen die Dokumente gespeichert werden, reduziert sich zusätzlich die Anzahl der Dokumente die verwaltet werden können. Der Raum, in den die Dokumente kollisionsfrei eingefügt werden können, ist nur noch $\frac{1}{n}$ mal so groß, als wenn nur eine Hashfunktion verwendet wird.

4.4 Autonome Agenten

Der letzte existierende Algorithmus zur Balancierung von Last in einem Peer-to-Peer System, der in dieser Arbeit vorgestellt werden soll, benutzt autonome Agenten.

In [33, 34] wird der Algorithmus Messor beschrieben, der so genannte autonome Agenten, die als Ameisen angesehen werden können, einsetzt.

Diese autonomen Agenten stammen aus dem Gebiet der Swarm Intelligence [63, 4]. Ein Schwarm dieser Agenten soll die Last zwischen den Knoten im Netz verteilen.

Der Algorithmus entstand im Anthill Projekt [61, 6]. Anthill ist ein Framework, das autonome Agenten für Peer-to-Peer Netze bereitstellt, die dann von Algorithmen für verschiedene Zwecke genutzt werden können.

Im folgendem Abschnitt wird zuerst das Anthill Framework kurz vorgestellt, bevor auf Messor, den eigentlichen Algorithmus zur Lastbalancierung, eingegangen wird.

4.4.1 Anthill

Anthill [61, 6] bildet ein Overlay Netz für ein Peer-to-Peer System mit Hilfe von so genannten Nestern. Das System ist vollkommen selbstorganisierend. In jedem dieser Nester befinden sich Ameisen (autonome Agenten), die je nach Programmierung unterschiedliche Aufgaben übernehmen können.

Die folgende Abbildung 4.4 zeigt verschiedene Ameisen in einem Peer-to-Peer Netzwerk mit sechs Nestern.

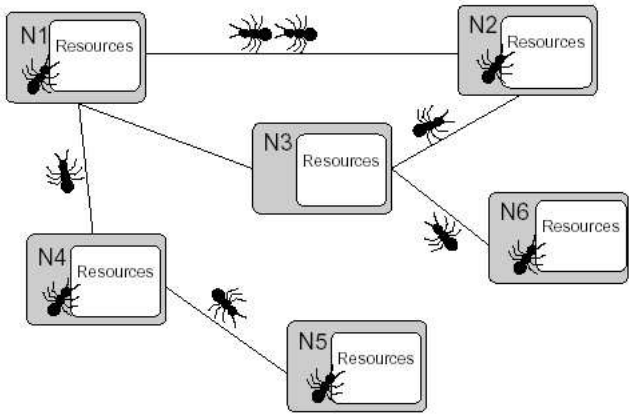


Abbildung 4.4: Autonome Agenten: Die Ameisen (autonome Agenten) bewegen sich zwischen ihren Nestern (Knoten) und verrichten dabei die Arbeit [33].

Jedes Nest interagiert mit den lokalen Applikationen und stellt ihnen Services zur Verfügung, die mit Hilfe der Nestern bewältigt werden können.

Für die Bewältigung der Aufgaben setzten die Nester ihre Ameisen ein, die jeweils für eine Aufgabe neu erzeugt werden. Diese Ameisen bewegen sich autonom im Netz, das die Nester verbindet. Jedes Nest hat dabei eine unbestimmte Anzahl an Nachbarn. Nachbarn sind alle anderen Nester, die ein Nest kennt. Unterwegs zwischen den Nestern nehmen die Ameisen Objekte, die je nach Aufgabengebiet unterschiedliche Dinge sein können auf und geben sie bei einem anderen Knoten wieder ab.

Anschließend kehren die Ameisen wieder, wenn es die Aufgabe erfordert, zum Ursprungsnest zurück oder sterben. Zusätzlich sterben sie nach einer Time-to-Live (TTL). Dies bedeutet, dass sie nach einer bestimmten Wegstrecke im Netz, zum Beispiel nach dem Besuch von n Knoten sterben. Damit kann ein unendliches Kreisen von Ameisen im System verhindert werden.

Die Ameisen kommunizieren zum Verrichten ihrer Arbeit nicht direkt untereinander. Um anderen Ameisen Informationen zukommen zu lassen, legen sie diese in Nestern ab. Andere Ameisen, die an diesem Nest anschließend vorbeikommen können die dort abgelegten Informationen dann aufnehmen.

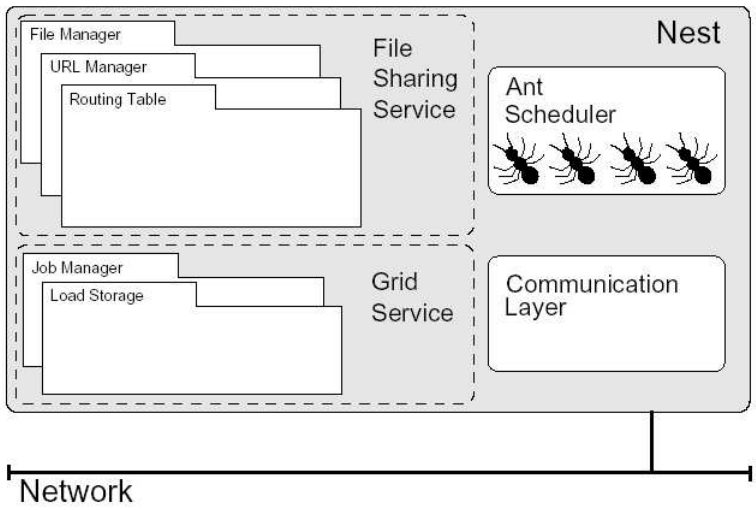


Abbildung 4.5: Autonome Agenten: Aufbau eines Nests in Anthill mit Scheduler. Der Knoten ist über den Communication Layer an das Netz angebunden, durch das die Ameisen laufen und ihre Arbeit verrichten [33].

Ein Nest besteht, wie in der Abbildung 4.5 dargestellt, aus mehreren Einheiten. Der Ant Scheduler übernimmt die Erzeugung der Ameisen und nimmt die Informationen von vorbeikommenden Ameisen auf. Der Communication Layer übernimmt die Verwaltung der Nachbarn. Die verschiedenen Services sind dem Benutzer vom Knoten zur Verfügung gestellten Dienste.

4.4.2 Lastbalancierung

Im Anthill Projekt wird mit Messor eine Komponente zur Lastbalancierung in diesem beschriebenen Szenario vorgestellt.

Für diese Aufgabe haben die entsprechend programmierten Ameisen mit SearchMax und SearchMin zwei verschiedene Zustände. Im initialen SearchMax Zustand suchen die Ameise solange im Netzwerk der verbundenen Nester, bis sie ein überlastetes Nest finden. Dabei kann die Überlastung zum Beispiel durch die Anzahl der gespeicherten Daten oder ähnlichem gemessen werden.

Nachdem eine Ameise ein solches Nest gefunden hat wechselt sie in den SearchMin Zustand. In diesem Zustand sucht die Ameise solange im Netz, bis sie ein unterbeschäftigtes Nest findet. Solange die Ameise keine Informationen über die Verteilung der Nester im Systems hat läuft sie das Netz zufällig ab.

Hat die Ameise ein solches Nest gefunden veranlasst sie den Lasttransfer direkt zwischen den beiden gefundenen Knoten. Eine Ameise transportiert also immer nur die Informationen über die Last der einzelnen Nester, nicht die Last direkt.

Nachdem ein solches Nest mit Unterbeschäftigung gefunden wurde wechselt die Ameise wieder in den SearchMax Zustand und beginnt von vorne.

Die Ameisen laufen dabei im Netz aber nicht immer komplett zufällig. Bei ihrer Reise speichern sie die Informationen über die Last der letzten n Knoten. Damit kann wenn der Zustand wechselt unter Umständen ein schon besuchter Knoten als neuer Zielpunkt der Reise benutzt werden. Wenn die Ameise zum Beispiel im SearchMax Zustand ist, können diese Informationen verwendet werden, um direkt einen Knoten mit hoher Last zu finden.

Die Informationen, die die Ameisen über die besuchten Knoten aufgesammelten haben, speichert jede Ameise auch in den allen besuchten Nestern ständig für andere Ameisen ab. Jede Ameise kann daraufhin durch Zufall bestimmen, ob sie einen gespeicherten Weg aus einem Knoten verfolgt, oder einen anderen zufälligen Weg einschlägt, um neue Nester im Netz finden zu können und deren Last aufzunehmen.

4.4.3 Ergebnisse

Die folgende Abbildung 4.6 zeigt die Ergebnisse von Messor, die mit dem Anthill Simulator erzeugt wurden. Bei dieser Simulation wurden 100 Nester benutzt.

Die gesamte Last von 10.000 Einheiten, wie zum Beispiel Jobs oder Daten, wurde am Anfang einem Nest zugeteilt. Nach 50 Iteration stellt sich eine Gleichverteilung der Last ein. Eine Iteration entspricht dabei einer Menge von 20 Ameisen, die jeweils ein einzelnen Schritt, zum Beispiel Last abfragen und zum nächsten Nest gehen, ausgeführt haben. Dabei konnten maximal 200 Objekte von einem Nest an ein anderes abgegeben werden. Bis 10 Iterationen wird die Last nur in der Umgebung des ausgelasteten Knotens verteilt. Ab 20 Iterationen haben alle Nester Last erhalten, und bei 50 Iterationen ist die Last fast komplett gleichverteilt.

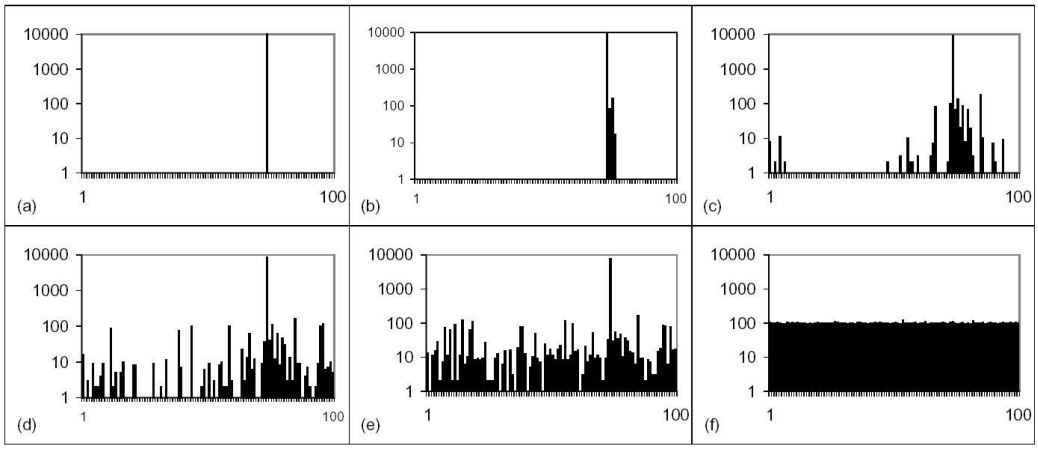


Abbildung 4.6: Autonome Agenten: Lastverteilung durch die Ameisen nach 0, 5, 10, 15, 20 und 50 Iterationen [33].

4.4.4 Bewertung

Der Algorithmus mit autonomen Agenten funktioniert bei Lastbalancierung von Jobs. Bei der Lastbalancierung von Daten in Distributed Hash Tables aber ist die Idee nicht uneingeschränkt einsetzbar.

Einzelne Daten können in DHTs nicht beliebig zwischen den Knoten verschoben werden, da sonst die Suche mit den Routing-Einträgen in den Knoten nicht funktioniert. In Chord können Einträge nicht aus ihrem vorgesehen Intervall entfernt werden, da sonst nur durch Ablaufen des Rings das Dokument wiedergefunden werden kann. In einem DHT System können nur die Grenzen der Intervalle oder direkt ganze Intervalle verschoben werden.

Wenn die Agenten zur Verschiebung ganzer Intervalle benutzt werden, entspricht dies jedoch dem One-to-One Schema der virtuellen Server, die in Abschnitt 4.1 vorgestellt wurden.

4.5 Vergleich

Folgende Tabelle gibt einen kurzen Vergleich, der in diesem Kapitel vorgestellten existierenden Lastbalancierungsalgorithmen.

Algorithmus	Modell	Vorteil	Nachteil
Power of two Choices	Mehrere Hashfunktionen verteilen Schlüssel im DHT	Einfach	Viele Zeiger die verwaltet werden müssen
Virtuelle Server	Knoten haben mehrere virtuelle Server, die untereinander getauscht werden	Leichtes Verschieben von Last möglich	Erhöhter Kommunikationsaufwand für Routing-Tabellen
Autonome Agenten	Ameisen suchen Knoten, die ihre Last austauschen können	Knoten brauchen keine Routing-Tabellen erstellen	Entspricht virtuellen Servern bei Lastverteilung von Daten in einem DHT

Tabelle 4.1: Lastbalancierungsalgorithmen: Vergleich der Algorithmen Power of two Choices, virtuelle Server und autonome Agenten. Für jeden Algorithmus wird das zu Grunde liegende Modell, sowie der größte Vorteil und Nachteil aufgeführt.

4.6 Fazit

Es wurden in diesem Kapitel verschiedene existierende Algorithmen zur Lastbalancierung vorgestellt.

Power of two Choices ist zwar einfach, erzeugt aber einen erhöhten Verwaltungsaufwand. Die Last wird außerdem wieder nur mit Hilfe von Hashfunktionen verteilt.

Die virtuellen Server sind dagegen besser zur Lastbalancierung geeignet. Das Problem ist aber, dass nicht zu viele virtuelle Server zwischen den Knoten verschoben werden dürfen, um nicht zu viel Last über das Netz verschicken zu müssen. Die vorgestellten virtuellen Server speichern außerdem so viele virtuelle Server, dass der Aufwand alle Routing-Einträge konsistent zu halten stark erhöht wird.

Im nächsten Kapitel wird deshalb ein eigener Lastbalancierungsalgorithmus vorgestellt. Dabei wird die Last wie bei einer Wärmeausbreitung weitergeben

Wenn die Tatsachen mit der Theorie
nicht übereinstimmen
- um so schlimmer für die Tatsachen.

– Herbert Marcuse (1898 - 1979)

Kapitel 5

Verwendeter Lastbalancierungsalgorithmus

Im letzten Kapitel wurden bereits einige existierende Lastbalancierungsalgorithmen vorgestellt. Im Rahmen dieser Arbeit entstand dabei ein neu entwickelter Algorithmus, der die Last innerhalb der Distributed Hash Tables besser als die vorgestellten Algorithmen verteilen soll.

In diesem Kapitel wird dieser neu entwickelte Lastbalancierungsalgorithmus vorgestellt. Er balanciert, ebenso wie die meisten im vorherigen Kapitel vorgestellten Lastbalancierungsalgorithmen, die Last in Distributed Hash Tables Systemen (siehe Kapitel 3). Wie gezeigt, haben Distributed Hash Tables bereits geeignete Verfahren zum effizienten Suchen von Daten.

Dabei wird die Last bei diesem Algorithmus wie bei einer Wärmeausbreitung weitergeben [48]. Bei einer Wärmeausbreitung gibt ein gegenüber der Umgebung wärmeres Material o.Ä. die Wärme an die umgebenden Bereiche ab. Diese Bereiche fahren so fort, bis eine ausgeglichene Verteilung der Wärme im gesamten System entsteht.

Dieses Prinzip auf DHTs angewendet bedeutet, dass die Last der Knoten als die Wärme angesehen wird. Ein Knoten mit zu seiner Umgebung erhöhter Last verteilt diese an die Knoten im umgebenen Bereich. Der umgebende Bereiche eines Knoten ist seine Nachbarschaft, also die Knoten aus dem DHT System, die dem Knoten bekannt sind. Diese Nachbarschaft besteht zum Beispiel bei Chord aus den Finger-Einträge der Routing-Tabellen. In CAN besteht sie aus den Knoten, der direkt im virtuellen mehrdimensionalen Raum angrenzenden Intervalle.

Als Beispiel entspräche diese Lastverteilung in CAN der Abbildung 5.2 aus [48] auf der nächsten Seite.

Der Knoten 0 mit erhöhter Last, gibt seine Dokumente an die umgebenden Nachbarn ab. Diese fahren so fort, bis ein einigermaßen ausgeglichenes System entsteht.

Bei anderen DHTs wie zum Beispiel Chord, die Finger-Tabellen zu entfernten Knoten im System (bei Chord im virtuellen Ring) erstellen und pflegen, können diese genutzt werden, um die Last direkt in entfernte Regionen im virtuellen Netzwerk zu verschieben.

In Distributed Hash Tables Systemen können die Knoten aber nicht einfach beliebige Dokumente an ihre Nachbarn abgeben. Würden diese so vorgehen, könnte mit

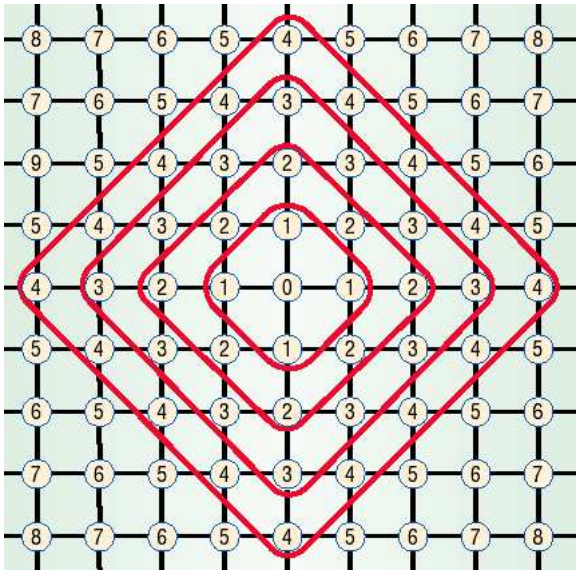


Abbildung 5.1: Wärmeausbreitung in CAN: Jeder einzelne Kreis entspricht im Distributed Hash Tables System CAN einem Knoten, der für ein Bereich zuständig ist, und zu allen Seiten ein Nachbar kennt. Die Ausbreitung der Last beginnt beim Knoten mit der Bezeichnung 0. Dieser gibt die Last an die Knoten mit der Bezeichnung 1 ab. Im nächsten Schritt geben diese wiederum die Last an ihre Nachbarn weiter. So werden durch jede Runde mehr Knoten erreicht [48].

Hilfe der Regel für die Nachfolger von Dokumenten und den Finger-Tabellen die Dokumente nicht mehr effizient gefunden werden. So können in DHT nur ganzen Intervalle oder Teilen von Intervallen verschoben werden, wie bei der Verschiebung der Last durch den Algorithmus mit virtuellen Servern. Um die Last zu verteilen, teilen die Knoten deshalb ihre bearbeiteten Intervalle untereinander auf.

Dieser Lastbalancierungsalgorithmus wird in diesem Kapitel zur Veranschaulichung anhand von Chord erklärt. Er funktioniert jedoch mit geringen Modifikationen in fast allen Distributed Hash Tables Systemen, da fast alle DHTs ihre Dokumente durch eine Hashfunktion in ein vorgegebenes abgeschlossenes Intervall abbilden. Dieses Intervall wird aufgeteilt und an die im Peer-to-Peer System beteiligten Knoten verteilt.

Zuerst wird in diesem Kapitel die Idee des verwendeten Lastbalancierungsalgorithmus detailliert vorgestellt. Es werden die Voraussetzungen des Algorithmus und die benötigten Änderungen an den Knoten erklärt. Danach wird der Lastbalancierungsalgorithmus im Detail vorgestellt. Abschließend werden die Eigenschaften des Algorithmus näher beschrieben.

Während in diesem Kapitel der Algorithmus zur Lastbalancierung nur theoretisch vorgestellt wird, beschäftigt sich das folgende Kapitel 6 mit der Simulation dieses Verfahrens und der bereits vorgestellten existierenden Lastbalancierungsalgorithmen. Das darauf folgende Kapitel 7 zeigt die Implementierung des beschriebenen Verfahrens als eigenständiger Client in einem Chord-Ring. Das sich der Implementierung anschließende 8. Kapitel beschäftigt sich schließlich mit den Messungen der vorgestellten Implementierung.

5.1 Architektur

Die Idee bei diesem Lastbalancierungsalgorithmus ist es, dass die Last wie bei der Wärmeausbreitung weiter gegeben wird. Dabei werden die Knoten in Intervallen mit großer Last zusammengezogen, während aus Intervallen mit geringerer Last die Knoten abgezogen werden, oder die Intervalle vergrößert werden, um frei werdende Knoten an überlastete Regionen abgeben zu können.

In Abbildung 5.2 wird ein Beispiel für eine Wärmeausbreitung gezeigt.

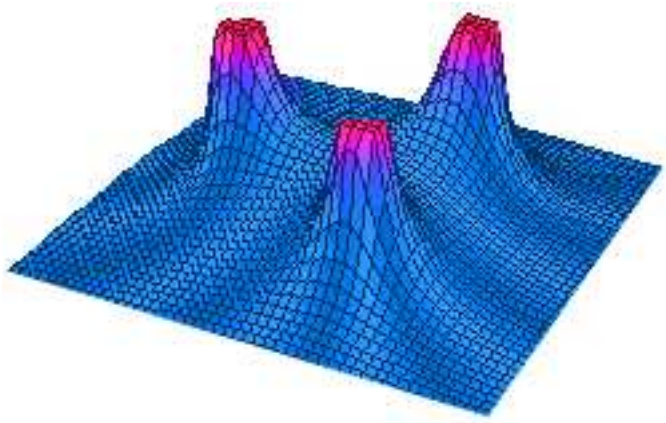


Abbildung 5.2: Wärmeausbreitung: Die Höhe eines Punktes entspricht der Höhe der Wärme in diesem Bereich. Je höher ein Punkt gelegen ist, desto wärmer ist er im Vergleich zur Umgebung. In diesem Beispiel sind drei Bereiche verzeichnet, die stark erhöhte Wärme haben. Diese Wärme wird langsam an die umgebenden Bereiche abgegeben, bis eine Gleichverteilung der Wärme erreicht wird.

Wenn die Abbildung 5.2 auf die Distributed Hash Tables angewendet wird, haben drei Regionen oder Knoten eine zur Umgebung stark erhöhte Last, die sie an die umgebenen Regionen abgeben möchten. Dazu können aus den Intervallen die wenig Last haben Knoten abgegeben werden. Die Intervalle, denen die Knoten zugeteilt werden können ihre Intervalle nun besser aufteilen, oder Intervallgrenzen zu ihren Nachbarn hin verschieben.

Jedem Intervall werden beim vorgestellten Algorithmus minimal f verschiedene Knoten zugeordnet. Dies dient gleichzeitig zur Erhöhung der Fehlertoleranz und zur Lastbalancierung. Wenn ein Intervall mehr als f Knoten hat, kann ein Knoten abgegeben werden, der dann andere Knoten bei ihrer Arbeit unterstützt. Hat ein Intervall 2f Knoten und mehr Dokumente gespeichert als gewünscht, kann das Intervall etwa mittig geteilt werden. So kann jeder Knoten ca. die Hälfte seiner Dokumente löschen.

Die Lastbalancierung kann zum Beispiel nach dem Einfügen von neuen Daten ins System angestoßen werden. Um die Last der umgebenen Knoten zu ermitteln, kann ein Knoten neben dem direkten Vorgänger und Nachfolger auch die Knoteneinträge in der Finger-Tabellen im Distributed Hash Tables System benutzen.

An den Distributed Hash Tables Systemen selber sollte so wenig wie möglich geändert werden, um den Lastbalancierungsalgorithmus, so universell wie möglich einsetzbar zu machen, und um die effizienten Suchalgorithmen der Distributed Hash Tables Systemen benutzen zu können.

5.1.1 Änderungen am Aufbau eines Knotens

Jedes Intervall kann von mehreren Knoten bearbeitet werden. Um dies zu gewährleisten, speichert jeder Knoten im Distributed Hash Tables System zusätzlich zu den Finger-Einträgen in Chord oder den Nachbarn in CAN noch Verweise auf Knoten, die das gleiche Intervall bearbeiten.

Jeder Knoten speichert in seiner Routing-Tabelle immer noch für jedes Intervall nur einen Knoten, der beliebig durch Zufall oder mit geeigneten Entscheidungskriterien aus der Menge der vorhandenen Knoten für ein Intervall ausgewählt wird. So kann als Entscheidungskriterium zum Beispiel die räumliche Nähe der Knoten untereinander, die Bandbreite der Knoten oder die Latenzzeit für Nachrichten zwischen den Knoten gewählt werden. Auch kann wieder der Knoten gewählt werden, der in einem Intervall bisher die geringste Last hat, oder der den anderen Rechnern in diesem Intervall in einer anderen Eigenschaft überlegen ist.

Beim Einfügen eines neuen Knotens in ein Intervall meldet sich der Knoten bei einem bekannten Knoten des Intervalls an, und bekommt von diesem alle anderen, für das Intervall ebenfalls zuständigen Knoten als Antwort zurückgeliefert. Zusätzlich werden alle gespeicherten Daten des Intervalls übertragen. Alle Knoten anderen Knotens des Intervalls werden ebenfalls informiert. Sollte in Zukunft ein Dokument in das Intervall eingefügt werden, so wird dies vom Knoten an alle anderen Knoten des selben Intervalls ebenfalls zur Speicherung versendet.

Wenn ein Knoten ausfällt, und die Knoten des Intervalls dies bemerken, kann die Stablisierungsroutine von Chord angewandt werden. Die Vorgänger und Nachfolger werden informiert und anschließend werden falsche Einträge in den Fingertabellen bei der periodischen Überprüfung der Finger von Chord festgestellt und korrigiert.

5.1.2 Virtuelle Server

Jeder Knoten kann, in Anlehnung an 4.2 auf Seite 38, mehrere virtuelle Server haben. Dabei hat jeder Knoten aber standardmäßig nur einen virtuellen Server. Die Möglichkeit mehrere virtuelle Server zu verwalten, ist für die Knoten gedacht, die eine zu den anderen im gleichem Intervall angesiedelten Knoten, höhere Rechnerausstattung haben.

Dann kann der Knoten sich für zwei oder mehrere Intervalle zuständig erklären, und so mehrere virtuelle Server haben, die jeweils für einen eigenen Bereich zuständig sind. Die Intervalle für die ein Rechner zuständig ist, dürfen aber nicht identisch sein, da sonst die Fehleranfälligkeit steigt.

Die Abbildung 5.3 zeigt wie ein Knoten zwei Intervalle bearbeitet.

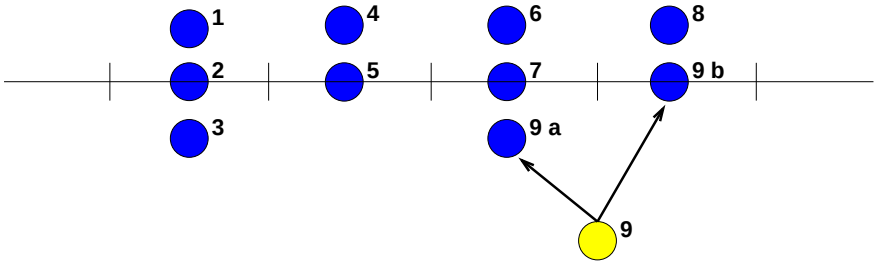


Abbildung 5.3: Knoten in Intervallen: Die Abbildung zeigt ein Teil eines Chord-Ringes. Die Kreise stellen Knoten im Chord-Ring dar. Jedem Intervall sind verschiedene Knoten mit der Bezeichnung 1 bis 9 zugeteilt. Das erste Intervall bearbeiten zum Beispiel die Knoten 1 bis 3 gemeinsam. Der Knoten 9 hat eine höhere Performance als die anderen Knoten, und bearbeitet deshalb zwei Intervalle. Der Knoten 9 erzeugt dazu für jedes Intervall ein virtuellen Server.

5.1.3 Teilen von Intervallen

Weiterhin besteht für die Knoten die Möglichkeit ihre Intervalle zu teilen. Wenn ein Intervall zu viele Knoten bearbeiten, kann dass Intervall ungefähr in der Mitte geteilt werden. Die Mitte kann entweder anhand der Grenzen des Intervalls einfach berechnet werden, oder durch den höchsten Hashwerte der Hälfte der gespeicherten Dokumente bestimmt werden.

Jedem der Knoten aus dem ursprünglichen Intervall wird eine Hälfte des neuen Intervalls zugeteilt.

Dabei muss keine Last in Form von Dokumenten oder ähnliches verschoben werden. Alle Knoten müssen nur ca. die Hälfte ihrer Last löschen. Es muss allen Knoten nur die ID mitgeteilt werden bei der geteilt wird, und welche Knoten für welchen Teil des Intervalls nun zuständig sind. Abschließend müssen die Vorgänger (predecessor) und Nachfolger (successor) angepasst werden.

5.1.4 Auswahl der Routing-Einträge

Um zusätzlich die Last durch Anfragen zwischen den beteiligten Knoten besser verteilen zu können kann jeder Knoten bei der Generierung der Finger-Einträge in den Finger-Tabellen verschiedene Strategien anwenden. Jeder Knoten kann dabei für sich entscheiden, welches Kriterium er anwendet. Dadurch wird die Last der einzelnen Knoten weiter verteilt, indem die Anfragen von verschiedene Knoten weitergeleitet werden.

Die Menge aller möglichen Knoten für ein Intervall erhält ein Knoten, indem er den successor eines Punktes bestimmt, und sich von diesem successor alle Adressen der direkten Nachbarn, die das gleiche Intervall bearbeiten, zurückgeben lässt.

Es gibt vier einfache Möglichkeiten ein Routing-Eintrag zu wählen.

- **Auswahl durch Zufall**
  Die einfachste Möglichkeit ist, ein beliebigen Knoten aus der Menge der Nachbarn des Nachfolgers für den gewünschten Routing-Eintrag auszuwählen.
- **Auswahl anhand der Last**
  Dabei wählt der Knoten den Knoten als Finger aus, der die geringste Last hat.
- **Auswahl anhand der Lokalität**
  Wenn ein Knoten in der Nähe des Knotens liegt, kann dieser als Routing-Eintrag gespeichert werden. Die Nähe kann dabei aber nur zum Beispiel anhand der IP-Adresse oder des Domainnamens der Knoten ausgewählt werden, was aber nicht unbedingt auch räumlich nahe sein muss.
- **Auswahl anhand der Latenzzeit**
  Der Knoten kann zu allen möglichen Knoten für einen Routing-Eintrag die Round-Trip Time messen, und den Knoten mit der kleinsten auswählen. Die Round-Trip Time ist die Zeit, die vergeht bis eine Nachricht, die an einen Knoten gesendet wurde, wieder beim Sender eintrifft. Wenn der Empfänger eine solche Nachricht erhält, wird diese umgehend beantwortet und an den Sender zurück geschickt.

5.2 Lastbalancierungsalgorithmus

Im Folgenden soll nach der Beschreibung der grundlegenden Idee und Architektur der Algorithmus zur Verteilung der Last genauer beschrieben werden.

In einem neuen Distributed Hash Tables System nimmt der erste Knoten eine beliebige Stellung im Ring ein. Ein möglicher Platz wäre zum Beispiel die Stelle mit dem Hashwert 0.

Es wird anschließend eine beliebige aber feste ganze positive Zahl f gewählt, die angibt wieviele Knoten ein Intervall mindestens bearbeiten sollen. Sollte die Zahl nachträglich geändert werden sollen, müsste sie allen Knoten bekannt gemacht werden. Daraufhin müssten alle Knoten ihre Intervalle anpassen und mit anderen Intervallen gegebenenfalls fusionieren.

Dokumente werden, wie beim original DHT, beim für Intervall zuständigen Knoten gespeichert oder gefunden. Wenn ein Intervall von mehreren Knoten bearbeitet wird, sendet der Knoten die neu zu speichernden oder zu löschenden Dokumente an die anderen Knoten des Intervalls.

Die dem Distributed Hash Table System später beitretende Knoten werden einem beliebigen bestehenden Knoten zugeordnet. Dieser Knoten wird allen ebenfalls für das Intervall zuständigen Knoten bekanntgemacht. Die Dokumente der Knoten, die das Intervall bisher bearbeitet haben werden anschließend kopiert. Es wird dem neuen Knoten ein successor und predecessor gezeigt, woraufhin dieser seine Finger-Tabelle selbständig berechnen, und für jedes Intervall selber einen beliebigen Knoten wählen kann. Dann können den Vorgängern und Nachfolgern des Intervalls mitgeteilt werden, das ein neuer Knoten das Intervall zusätzlich bearbeitet. Diese wählen dann ihre Routing-Einträge nach dem in 5.1.4 vorgestellten Möglichkeiten.

Es gibt nun drei Möglichkeiten für einen Knoten zusammen mit den anderen Knoten in gleichem Intervall die Last zu balancieren.

2f Knoten in einem Intervall und zu viel Last in den Knoten

Wenn ein Intervall 2f verschiedene Knoten bearbeiten, und jeder Knoten im Intervall mehr Dokumente speichert als von ihm erwünscht, wird das Intervall geteilt.

Das Intervall wird anhand der Hashwerte der gespeicherten Dokumente etwa mittig geteilt. Jedem der Knoten aus dem ursprünglichen Intervall wird eine Hälfte des neuen Intervalls zugeteilt.

Dabei muss keine Last in Form von Dokumenten oder ähnliches verschoben werden. Alle Knoten müssen nur ca. die Hälfte ihrer Last löschen. Es muss allen Knoten dafür nur die Zahl mitgeteilt werden bei der das Intervall geteilt wurde.

Abschließend müssen die Vorgänger und Nachfolger angepasst werden.

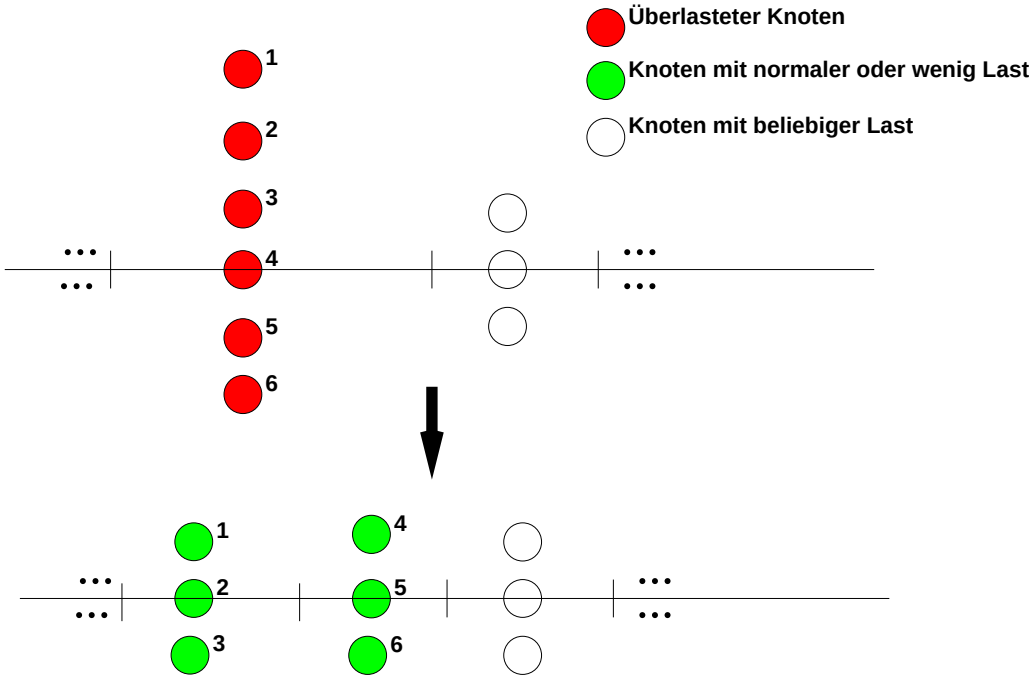


Abbildung 5.4: Algorithmus: Die Knoten 1 bis 6 bearbeiten zusammen ein Intervall. Alle Knoten haben zu viel Last. Da nur mindestens 3 Knoten ein Intervall bearbeiten müssen, können die Knoten dass Intervall aufteilen.

Vorstehende Abbildung zeigt ein Beispiel für eine solche Teilung der Intervalle, wenn drei Knoten ein Intervall mindestens bearbeiten müssen.

Mehr als f Knoten in einem Intervall

Intervalle, die mehr als f, aber weniger als 2f Knoten haben, die dies Intervall bearbeiten können eventuell Knoten abgeben.

Wenn die Knoten des Intervalls aber selber Überlast haben, unternehmen sie aber nichts und warten darauf, dass ihnen weitere Knoten zugeteilt werden.

Wenn diese Knoten eines Intervalls dagegen wenig Last haben, prüfen sie periodisch die Informationen über Anzahl der Knoten in einem anderen Intervall und die Last in diesem Intervalle. Diese Intervalle können durch die Knoten gefunden werden, die ein Knoten als Finger-Einträge in seiner der Routing-Tabelle gespeichert habt. Wenn dort ein Intervall auch noch nicht 2f Knoten hat, wird dem Intervall das Überlast hat, die überschüssigen Knoten abgegeben.

Die Knoten im neuen Intervall können dann, wenn nun genügend Knoten vorhanden sind ihr Intervall mit den in der ersten Regel beschriebenen Mitteln teilen. Ansonsten müssen sie darauf warten, dass mehr Knoten ihrem Intervall beitreten. Aber dadurch, dass nun ein Knoten mehr das Intervall bearbeitet, sinkt auch die Zahl der Anfragen pro Knoten, wenn die Gesamtanzahl der Anfragen, die ein Intervall betreffen konstant bleibt, da Knoten aus anderen Intervallen nun einen Knoten mehr zur Wahl haben, an den sie ihre Anfragen senden können.

Die folgende Abbildung zeigt ein Beispiel für eine solche Verschiebung von Knoten in Regionen mit höherer Last.

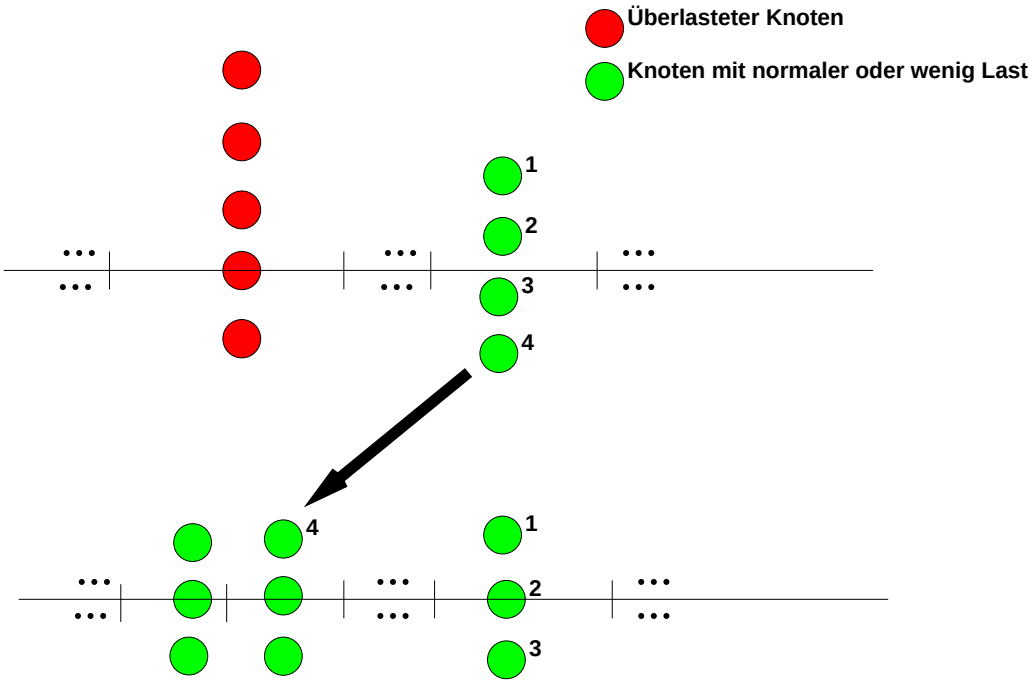


Abbildung 5.5: Algorithmus: Die Knoten 1 bis 4 bearbeiten zusammen ein Intervall und haben nicht mehr Dokumente als gewünscht in ihrem Intervall. Da nur mindestens 3 Knoten ein Intervall bearbeiten müssen können, kann der Knoten 4 an ein anderes Intervall mit Überlast abgegeben werden, dass dann eventuell aufteilen werden kann.

Nicht mehr als f Knoten in einem Intervall

Als zusätzliche Alternative können noch Intervallgrenzen verschoben werden. Die Knoten vergleichen dabei ihre Last mit der Last der direkten Vorgänger und Nachfolger. Wenn ein Bereich mehr Last als sein Nachbarbereich hat, wird ein Teil der Last abgegeben, indem die Intervallgrenze zwischen den beiden Bereichen verschoben wird.

Die folgende Abbildung 5.6 zeigt ein Beispiel für eine solche Verschiebung der Intervallgrenzen.

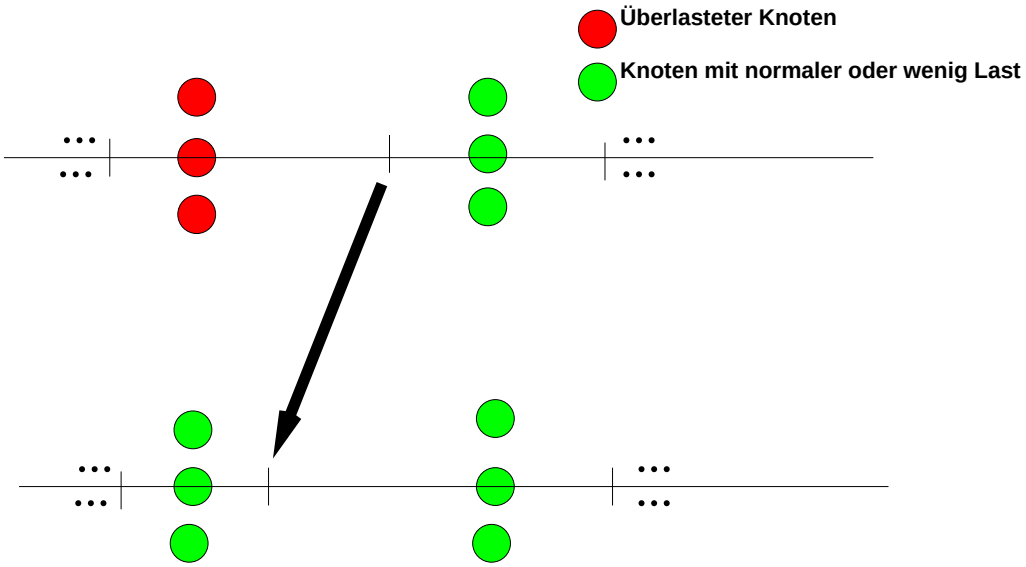


Abbildung 5.6: Algorithmus: Die Knoten im rechten Intervall bearbeiten zusammen ein Intervall und haben nicht mehr Dokumente als gewünscht gespeichert. Da die Knoten im Intervall vor ihnen überlastet sind, werden die Intervallgrenzen verändert, so dass alle Knoten nicht mehr überlastet sind.

Dieser letzte Schritt ist, wenn überhaupt, aber nur selten einzusetzen. Er kann zwar zu einer kompletten Gleichverteilung der Last im Ring führen, braucht dazu aber viele Runden, und verschiebt viel Last über das Netz. Bei der im nächsten Kapitel folgenden Simulation wird deshalb gezeigt, dass auch ohne diesen Schritt die Last zufriedenstellend verteilt werden kann.

Wenn ein Knoten feststellt, das im Intervall seines Vorgängers oder Nachfolger zu wenige Knoten vorhanden sind, zum Beispiel durch den Ausfall von Knoten, werden beide Intervalle zu einem verschmolzen, um die Fehlertoleranz wieder herzustellen.

5.3 Eigenschaften

Im Folgenden werden ein paar Eigenschaften des verwendeten neuen Lastbalancierungsalgorithmus vorgestellt. Es werden unter anderem die Fehlertoleranz und der zusätzliche Aufwand, der durch den Algorithmus entsteht untersucht.

5.3.1 Verteilung der Intervalle

Ein Hauptproblem bei der Lastbalancierung mit Distributed Hash Tables sind die unter Umständen nicht gleichmäßig verteilten Intervalle. Trotz der Hashfunktion können einige Intervalle kleiner sein als andere. Außerdem können Dokumente gehäuft in einem Bereich des Systems auftauchen.

Bei diesem Algorithmus können die Knoten ihren Platz im Distributed Hash Tables System selbst wählen und so die Intervallgrößen und die Elemente pro Intervall direkt beeinflussen. Durch das Aufteilen der Intervalle haben die Intervalle ungefähr gleich viele Dokumente.

5.3.2 Fehlertoleranz

Durch die in 5.1.1 auf Seite 54 beschriebene Änderung am Aufbau eines Knotens wird das System ausfallsicherer gegenüber einzelnen Ausfällen von Knoten, da immer mindestens f Knoten ein Intervall bearbeiten müssen.

Das System kann ein Ausfall von f-1 Knoten im gleichen Intervall verkraften. Wenn zu wenige Knoten ein Intervall bearbeiten wird durch den oben beschriebenen Algorithmus dafür gesorgt, dass das Intervall mit den angrenzenden Intervallen verschmolzen wird. Dabei werden wieder f Kopien der Dokumente erzeugt.

5.3.3 Zusätzlicher Aufwand

Es soll der zusätzliche Aufwand betrachtet werden, der durch den vorgestellten Lastbalancierungsalgorithmus entsteht. Zunächst wird dabei auf den höheren Kommunikationsaufwand eingegangen. Dieser entsteht dadurch, dass mehrere Knoten für ein Intervall zuständig sind, und durch die Nachrichten über die Lastsituation in den Intervallen. Danach wird der erhöhte Verwaltungsaufwand bestimmt, der durch den Lastbalancierungsalgorithmus entsteht.

Der erhöhte Kommunikationsaufwand beim vorgestellten Lastbalancierungsalgorithmus liegt vor allem beim Synchronisieren der Daten aller Knoten in einem Intervall. Durch Wahl des Parameters f kann aber eingestellt werden, wie viel zusätzlicher Aufwand entsteht.

Der Mehraufwand besteht im weiteren im Verteilen von Nachrichten über in einem Intervall hinzugekommene oder entfernte Knoten, die an die anderen Knoten des selben Intervalls bekannt gemacht werden müssen.

Jeder Knoten hat aber nur eine geringe Anzahl an virtuellen Servern, im Normalfall nur einen. Damit hat ein Knoten nicht mehr Aufwand als in Chord ohne Lastbalancierung alle Finger-Einträge konsistent zu halten und zu verwalten. Zusätzlich muss jeder Knoten nur pro virtuellen Server f-1 Nachbarn speichern.

Als letztes soll beim Aufwand die Last untersucht werden die zwischen Knoten im System verschoben wird. Last wird dabei nur in wenigen Fällen direkt verschoben. Wenn ein Intervall geteilt wird müssen die Knoten nur Dokumente (also Last) löschen. Beim Verschieben von Knoten muss auf diesem die Last ebenfalls erstmal nur gelöscht werden. Nur wenn ein Knoten einem bestehenden Intervalls beitritt müssen die Daten des Intervalls kopiert werden.

5.3.4 Nachteil

Es soll am Ende des Kapitels auf einen mögliches Szenario eingegangen werden, dass die vorgestellte Art von Lastverteilung unter Umständen erzeugen kann und die Laufzeit von Chord beeinflusst.

Da in mit diesem Lastbalancierungsalgorithmus die Intervalle aktiv verändert werden können zufällig ungünstige Intervalle entstehen.

Die folgende Abbildung 5.7 zeigt diesen Fall.

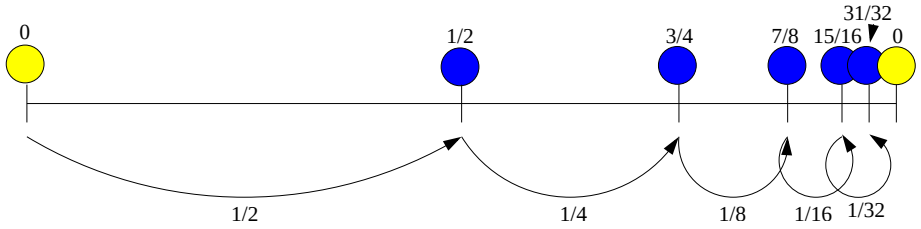


Abbildung 5.7: Problem: Die Abbildung zeigt eine Verteilung der Knoten im kompletten virtuellen Chord-Ring. Bei der dargestellten Verteilung der Intervalle kann die Suche zu einer Laufzeit von $O(n)$ führen, wenn die Suche am Knoten 0 beginnt. Die Intervalle werden immer halbiert, angefangen mit dem halben Kreis.

Das Intervall ab dem Knoten, der ein Dokument sucht, entspricht dem halben Kreis in Chord. Daraufhin werden die folgenden Intervalle immer weiter halbiert. Wenn der Knoten nun genau ein Dokument sucht, das sein predecessor gespeichert hat, erhöht sich die Laufzeit auf $O(n)$.

Dies kann aber nicht passieren wenn die Intervalle ungefähr gleichmäßig erzeugt werden, da die Dokumente im gesamten Kreis angeordnet sind. Sollte ein solches Szenario dennoch auftreten kann durch Wahl eine anderen Knoten bei der Suchen dieses Dokuments die Suche wieder gewohnt schnell ablaufen.

5.4 Fazit

Es wurde in diesem Kapitel ein Algorithmus vorgestellt, der mit nur drei Regeln die Last innerhalb eines Distributed Hash Tables System verschieben kann. Dabei wurde zugleich die Fehlertoleranz des Distributed Hash Tables System bei Bedarf erhöht, indem mehr Knoten einem Intervall zugeordnet werden.

Das folgende Kapitel 6 beschäftigt sich nun mit der Simulation. Damit soll die Eigenschaften des Lastbalancierungsalgorithmus genauer untersucht werden, und die Ergebnisse mit den bereits existierenden Lastbalancierungsalgorithmen für Distributed Hash Tables verglichen werden. Das darauf folgende Kapitel 7 stellt die Implementierung dieses vorgestellten Algorithmus vor. Das sich der Implementierung anschließende 8. Kapitel beschäftigt sich schließlich mit den Messungen der vorgestellten Implementierung.

One of the most interesting methods of predicting the future is simulation.

– Richard Dawkins in „The Selfish Gene“ (1990)

Kapitel 6

Simulation

Im Rahmen dieser Arbeit entstand zu den vorgestellten Lastbalancierungsalgorithmen ein Simulator. Dieser simuliert im einem Chord-Ring die Lastbalancierung mit den verschiedenen vorher vorgestellten Lastbalancierungsalgorithmen.

Diese Lastbalancierungsalgorithmen werden simuliert, um deren Verhalten bei großer Anzahl von Knoten und Dokumenten zu beobachten, und abschließend vergleichen zu können.

Im Brockhaus in einem Band [8] wird die Idee der Simulation erklärt.

„Darstellung technischer, biologischer, ökonomischer u. a. Prozesse oder Systeme durch (math.) Modelle. Simulationen erlauben Untersuchungen oder Manipulationen, deren Durchführung am eigentlichen System zu gefährlich, zu teuer oder anderweitig nicht möglich ist.“

Aus diesem Grund wurde auch der Simulator entwickelt. Eine realer Test mit einer großen Anzahl von Knoten ist kaum möglich. So wurden bei der Simulation bis zu 32.000 Knoten und bis zu einer Millionen Dokumente simuliert.

In dieser Simulation werden dabei fast alle in den letzten beiden Kapiteln vorgestellten Lastbalancierungsalgorithmen untersucht. Es wird dazu ein Chord-Ring variabler Größe simuliert. Die verschiedenen Algorithmen zur Lastverteilung wurden mit der Verteilung der Last wie sie Chord ohne Lastbalancierung durchführt, verglichen.

Die simulierten Algorithmen sind die bereits vorgestellten Lastbalancierungsalgorithmen Power of two Choices (siehe Abschnitt 4.3), virtuelle Server (Abschnitt 4.2) sowie der neu entwickelte Lastbalancierungsalgorithmus (Kapitel 5). Diese Algorithmen wurden zur Balancierung der Last in einem unbalancierten Chord-Ring verwendet.

Der Simulator ist in der Programmiersprache Java [59] geschrieben. Im Anhang auf Seite 100 befindet sich die Abbildung A.2 mit einem UML-Klassendiagramm vom Aufbau des Simulators. Dieser Aufbau wird später noch detailliert im Abschnitt Architektur erklärt.

Dieses Kapitel beschreibt die Architektur des Chord-Simulator, seine Benutzung und die Ergebnisse der Simulation der verschiedenen Lastbalancierungsalgorithmen.

6.1 Chord-Simulator

Dieser Abschnitt beschreibt den im Rahmen dieser Arbeit für Chord entwickelten Simulator. Dieser ist in der objektorientierten Programmiersprache Java [59] geschrieben. Mit diesem Simulator, kann ein Chord-Ring variabler Größe simuliert werden. Zusätzlich können die verschiedenen Lastbalancierungsalgorithmen gewählt werden. Der Simulator kann Chord ohne Lastbalancierung, als auch mit virtuellen Servern oder Power of two Choices simulieren. Außerdem kann er den verwendeten Lastbalancierungsalgorithmus verwenden.

Dieser Abschnitt beschreibt den Aufbau des Simulators, seine Benutzung und ein Beispiel.

6.1.1 Architektur

Im folgenden soll der Aufbau des Simulators beschrieben werden.

Auf Seite 100 befindet sich ein UML-Diagram A.2 vom Simulator, das den Aufbau zeigt.

Mit der Klasse **Simulation** wird die Simulation gestartet und ausgewertet. Es hält die einmal gewählten und dann festen Daten, die für alle Knoten im Ring gleich sind (wie die Größe des Chord-Rings), bereit, um diese nicht unnötig oft kopieren zu müssen. Auch mit Hilfe eines **Logobjekts** werden die Statistiken der Simulation zusammengesucht, ausgewertet und gespeichert.

Von der Simulation wird ein Objekt von **Chord** erzeugt, dass den Ring darstellt. Dieser Ring erzeugt die Knoten und Dokumente zufällig und hat als einziger einen Überblick über alle gespeicherten Knoten im Ring zur Auswertung. Diesen Überblick haben die einzelnen Knoten nicht, sie kennen nur ihre Nachbarn. Die Chordklasse erzeugt dabei die Knoten und Dokumente nach den jeweiligen Vorgaben der Lastbalancierungsalgorithmen. So müssen beim Erzeugen und Einspeisen von Dokumenten in den Ring zum Beispiel bei Power of two Choices zusätzliche Pointer zum Dokument erzeugt werden, und beim entsprechenden Knoten gespeichert werden.

Ein **Document** speichert die Daten eines Dokuments im Chord-Ring, wie den Namen und den Hashwert, der den Platz im Ring festlegt.

Ein **Node** (Knoten) verwaltet die virtuellen Server. Im Standard Chord ohne Lastbalancierung oder bei Power of two Choices, hat jeder Knoten immer genau einen virtuellen Server. Node kann verschiedene Lastbalancierungsalgorithmen ausführen, je nachdem welcher Algorithmus beim Start gewählt werden. Bei der Lastbalancierung mit dem neu vorgestellten Algorithmus speichert der Knoten ebenfalls die Knoten, die das gleiche Intervall bearbeiten.

Die **VirtualServer** verwalten die Dokumente bzw. die Pointer, für die dieser virtueller Server mir seiner ID im Ring zuständig ist. Jeder virtuelle Server besitzt eine Fingertable.

Die **Fingertable** verwaltet alle Daten die zum navigieren im Ring notwendig sind. Es speichert die Vorgänger und Nachfolger, des zugehörigen Knotens, sowie die im Kapitel 3 vorgestellten Finger von Chord.

Ein **Finger** besteht aus den in Tabelle 3.4 gezeigten Attributen Start, Beginn des Intervalls, Ende des Intervalls und zugehöriger Knoten node.

6.1.2 Benutzerinterface

Die Simulation wird von der Kommandozeile gestartet. Dabei hat der Simulator folgende Optionen.

```
          chord simulator for loadbalancing algorithms
-----------------------------------------------------------------------

usage:
      java Ring nodes doc loadbalancing ring (output) (minNeighbors)

where:
      nodes         = number of nodes in ring
      doc           = number of documents
      loadbalancing = 0 none,
                      1 power of two choices,
                      2 virtual server,
                      3 new algorithm
      ring          = ring size
      output        = 0 output to standard out (default),
                      1 output to file
      minNeighbors  = minimum number of neighbors in new
                      algorithm mode (default 3)
```

Abbildung 6.1: Optionen: Alle möglichen Optionen des Simulators. Wenn die minimale Anzahl an Nachbarn eingestellt werden soll muss auch für den den Output eine Zahl gewählt werden.

Sollten beim Aufruf mit der Konsole dem Simulator keine Parameter übergeben worden sein fragt das Programm diese interaktiv ab.

Während der Simulation werden Statusmeldungen erzeugt und am Ende die Ergebnisse der Simulation ausgegeben und im Gnuplot fähigem Format zur Darstellung auf der Festplatte gespeichert.

6.1.3 Beispiel

Im folgenden wird als Beispiel ein Teil der Ausgabe bei der Simulation eines Chord-Ringes ohne Lastverteilung mit 3 Knoten und 5 Dokumenten gezeigt. Der Ring hat eine Größe von $m = 5$, dass entspricht einer Ringgröße von maximal $2^5 = 32$ Knoten.

```
Node 16 with 1 Virtual Server(s)
  Load: 1
  VirtualServer V16 with HashID 16
    storing 1 document(s):
      [D11:11]
    FingerTable from VirtualServer V16 with HashID 16
      predecessor: V8:8 (@ Node 8) successor: V25:25 (@ Node 25)
      Start       Interval                  Virtual Server
              17  [        17,         18]  V25:25 (@ Node 25)
              18  [        18,         20]  V25:25 (@ Node 25)
              20  [        20,         24]  V25:25 (@ Node 25)
              24  [        24,          0]  V25:25 (@ Node 25)
               0  [         0,         16]  V8:8   (@ Node 8)

Node 8 with 1 Virtual Server(s)
  Load: 2
  VirtualServer V8 with HashID 8
    storing 2 document(s):
      [D26:26, D29:29]
    FingerTable from VirtualServer V8 with HashID 8
      predecessor: V25:25 (@ Node 25) successor: V16:16 (@ Node 16)
      Start       Interval                  Virtual Server
               9  [         9,         10]  V16:16 (@ Node 16)
              10  [        10,         12]  V16:16 (@ Node 16)
              12  [        12,         16]  V16:16 (@ Node 16)
              16  [        16,         24]  V16:16 (@ Node 16)
              24  [        24,          8]  V25:25 (@ Node 25)

Node 25 with 1 Virtual Server(s)
  Load: 2
  VirtualServer V25 with HashID 25
    storing 2 document(s):
      [D21:21, D18:18]
    FingerTable from VirtualServer V25 with HashID 25
      predecessor: V16:16 (@ Node 16) successor: V8:8 (@ Node 8)
      Start       Interval                  Virtual Server
              26  [        26,         27]  V8:8   (@ Node 8)
              27  [        27,         29]  V8:8   (@ Node 8)
              29  [        29,          1]  V8:8   (@ Node 8)
               1  [         1,          9]  V8:8   (@ Node 8)
               9  [         9,         25]  V16:16 (@ Node 16)
```

Abbildung 6.2: Beispiel: Die Abbildung zeigt die Ausgabe der Simulation eines Chord-Ringes ohne Lastverteilung mit 3 Knoten und 5 Dokumenten. Der Ring hat eine Größe von $m = 5$.

6.2 Simulation

Es wurden mit dem beschriebenen Simulator verschiedene Simulationen durchgeführt. Dabei wurde vor allem wurde die Verteilung der Dokumente auf die Knoten untersucht.

Jede Messung wurde mehrmals wiederholt und der Mittelwert der Messungen berechnet. Es wurden jeweils 4.096 Knoten in den einzelnen Messreihen simuliert. Diese Zahl wurde gewählt, um ein Vergleich zu den Messungen in den anderen vorgestellten Arbeiten zu haben (gleich viele Knoten wie bei den virtuellen Servern, die Hälfte der Knoten wie bei den Power of two Choices Messungen). Diese Knoten mussten zusammen insgesamt zwischen 100.000 und 1.000.000 Dokumente speichern. Der Chord-Ring hat eine Größe von $m = 22$, d. h. es können jeweils $2^{22} = 4.194.304$ Dokumente und Knoten gespeichert und im Chord-Ring verwaltet werden. Als Last eines Knotens wurde in dieser Simulation die Anzahl der gespeicherten Dokumente pro Knoten benutzt.

Bei der Simulation der Lastbalancierungsalgorithmen Power of two Choices, virtuelle Server und bei Chord ohne Lastbalancierung werden zuerst alle Knoten und dann alle Dokumente erzeugt, da die Systeme die Daten immer auf die gleichen Knoten am Ende verteilen, egal wie die Reihenfolge beim Erzeugen der Objekte ist. Bei der Simulation vom neuen Lastbalancierungsalgorithmus wurden die Dokumente und Knoten zufällig in der Reihenfolge erzeugt, da dadurch die Lastbalancierung schneller abgeschlossen ist. Ansonsten würden sich alle Knoten zuerst im gleichen Intervall befinden, weil noch keine Last im System ist die balanciert werden müsste.

6.2.1 Chord ohne Lastbalancierung

Als erstes wird das Ergebnis der Simulation von Chord im Standardfall ohne jegliche zusätzliche Lastbalancierung vorgestellt.

Die Abbildung 6.3(i)-(iv) zeigt die Ergebnisse die unter oben genannten Bedingungen erzielt wurden.

Die Abbildung 6.3(i) zeigt zuerst als Beispiel die Verteilung von 500.000 Dokumenten in Chord ohne Lastverteilung. Es wurden die 4.096 Knoten mit ihrer jeweiligen Last eingezeichnet, die bei der Verteilung der 500.000 zufällig erstellten Dokumenten von Chord erzeugt wurden. Es ist eine starke Unregelmäßigkeit in der Verteilung der Dokumente zu erkennen. Die helle Linie zeigt den optimalen Wert an, den eigentlich jeder Knoten haben sollte. Da so viele Knoten nebeneinander eingetragen wurden, sind nur die Ausschläge nach oben deutlich zu sehen. Es wird deutlich, dass die Dokumente zu unregelmäßig auf die einzelnen Knoten verteilt wurden.

Die Abbildung 6.3(ii) zeigt daher die Verteilung der Dokumente auf die Knoten im gleichen Versuch. Es wurde dazu in die Grafik eingetragen wieviele Knoten eine bestimmte Anzahl von Dokumenten gespeichert haben. Die helle Linie zeigt wieder den optimalen Wert von ca. 122 Dokumenten pro Knoten (500.000 Dokumente auf 4.096 Knoten = 122 Dokumente / Knoten). Es ist zu sehen, dass einige wenige Knoten viel Last haben, während die meisten keine oder wenig Last haben. Verschieden Knoten haben fast 10 mal so viel Last wie bei der optimalen Verteilung.

Der nächste Teil der Abbildung 6.3 mit der Bezeichnung (iii) gibt nun die Verteilung von Dokumenten auf die Knoten bei verschiedener Gesamtanzahl der Dokumente

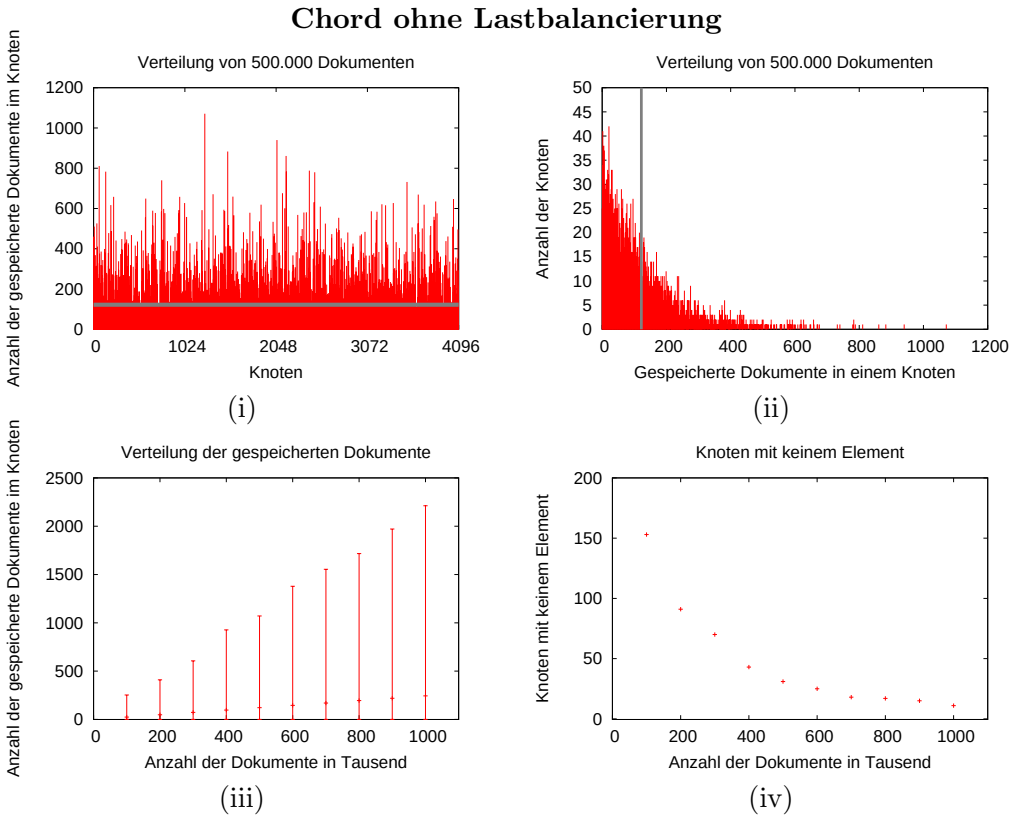


Abbildung 6.3: Simulation: Chord ohne Lastverteilung

an. Es wurden auf die 4.096 Knoten zwischen 100.000 und 1.000.000 Dokumente verteilt. Der oberste Wert gibt an, wieviele Dokumente maximal, der untere wieviele minimal auf einem Knoten gespeichert wurden. Der mittlere Wert gibt die optimale Verteilung der Dokumente an. Auch wenn die Dokumente mehr werden, gibt es immer noch Knoten, die keine Last haben. Die Last einzelner Knoten steigt stark an. Verschieden Knoten haben immer noch fast 10 mal so viel Last wie bei der optimalen Verteilung.

In der letzten Abbildung 6.3(iv) dieses Abschnitts wird abschließend vor der Bewertung gezeigt, wieviele Knoten kein Element gespeichert haben. Es haben immer Knoten überhaupt keine Dokumente gespeichert. Je mehr Dokumente dem System hinzugefügt werden, umso weniger Knoten haben aber keine Dokumente mehr gespeichert, da die Wahrscheinlichkeit immer mehr steigt, dass dem Knoten ein Element zugeteilt wird.

6.2.1.1 Bewertung

Es wurde gezeigt, dass die Lastbalancierung von Chord nur über die Hashfunktion nicht ausreicht. Es hängt beim Chord ohne Lastbalancierung zu sehr davon ab, wie groß das Intervall ist, das sie bearbeiten, und welche Hashwerte die Dokumente bekommen. Es wird durch die Hashfunktion keine gleichmäßige Verteilung der Knoten auf dem Ring und somit eine gleichmäßige Verteilung der Dokumente erreicht.

6.2.2 Power of two Choices

Als nächstes wird die Simulation von Chord mit Power of two Choices vorgestellt. Als Last eines Knotens wurde in dieser Simulation die Anzahl der gespeicherten Dokumente pro Knoten benutzt.

Die Abbildung 6.9(i)-(v) zeigen nun die Ergebnisse dieser Simulation.

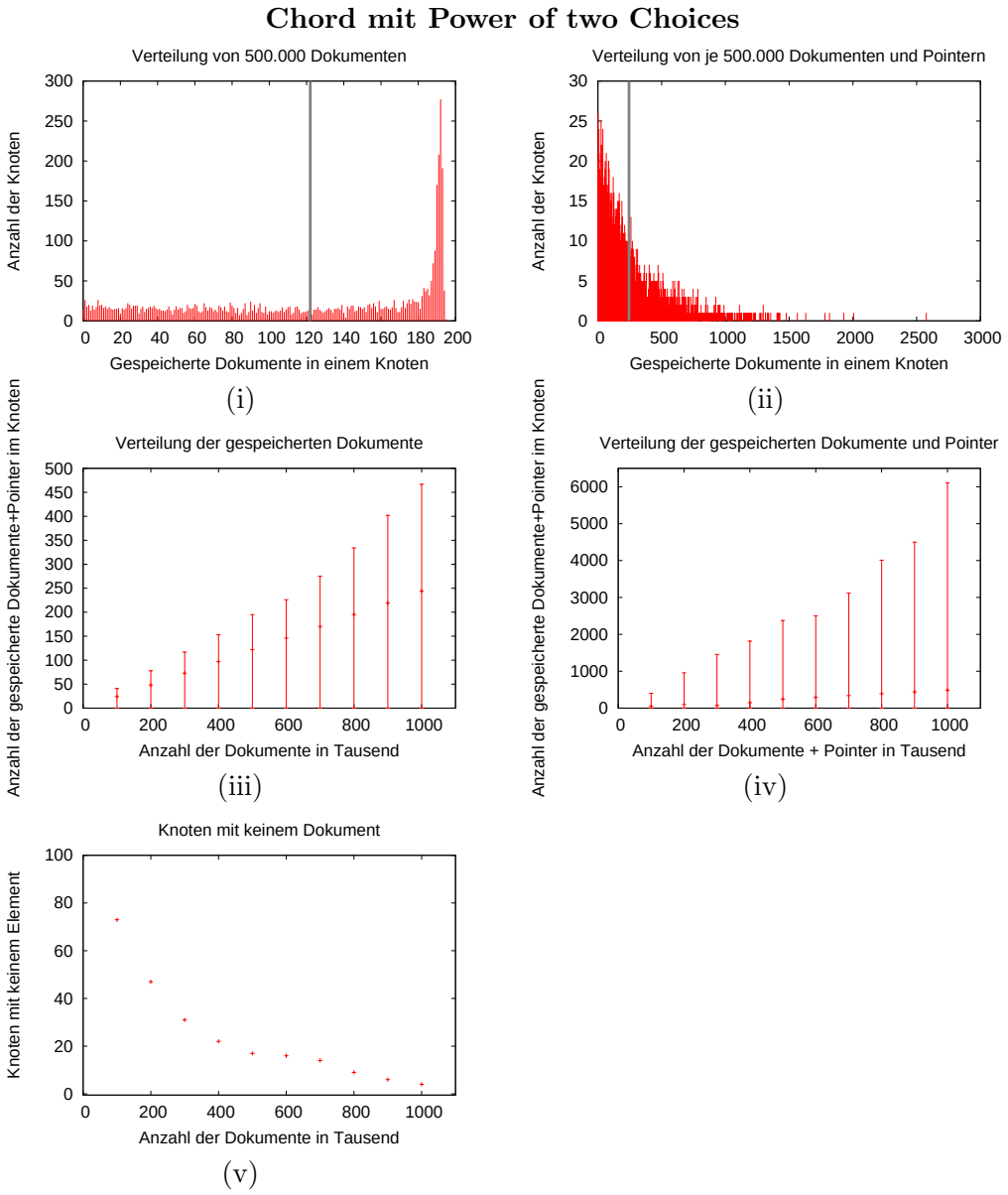


Abbildung 6.4: Simulation: Chord mit Power of two Choices

Der erste Teil (i) in Abbildung 6.9 zeigt wieder die Verteilung der Dokumente auf die beteiligten Knoten. Der Ausschlag am Ende sind Knoten, die alle sehr große Intervalle haben. Dadurch werden dort häufiger Dokumente gespeichert, wenn die Pointer auch auf Knoten mit großen Intervallen fallen. Außerdem existieren Knoten die neben dem Dokument auch das gleiche Dokument zusätzlich als Pointer speichern müssen, da beide berechneten Hashwerte in den gleichen Bereich fallen. Das Maximum an Dokumenten ist aber nur ein sechstel so groß wie bei Chord ohne

Lastbalancierung.

Die Abbildung 6.9(ii) zeigt die Verteilung der gesamten Last, also Dokumente und Pointer auf die Knoten. Es ist zu sehen, dass die Last besser verteilt wurde als in Chord ohne Lastbalancierung. Es gibt aber immer noch starke Schwankungen in der Last die ein Knoten zu bewältigen hat. Die Linie die wieder das Optimum anzeigt ist nun doppelt so hoch, da nun Dokumente und Pointer verteilt werden müssen.

Die Abbildung 6.9(iii)-(iv) geben nun die Verteilung von Dokumenten auf die Knoten bei einer verschiedenen Gesamtanzahl von Dokumenten an.

Es wurden wieder zwischen 100.000 und 1.000.000 Dokumente verteilt. Der oberste Wert gibt an, wieviele Dokumente maximal, der untere wieviele minimal auf einem Knoten gespeichert wurden. Der mittlere Wert gibt die optimale Auslastung an.

In Teil (iii) ist wieder die Verteilung der Knoten abgebildet, im nächsten Bild die gesamte Last von Dokumenten und Pointern. Da bei der Simulation von Power of two Choices die Last anhand der gespeicherten Dokumente pro Knoten verteilt wurde, sind die Dokumente gleichmäßiger verteilt, die gesamte Last variiert dagegen stärker. Die Ergebnisse entsprechen dabei in etwa den Ergebnissen aus Abbildung 4.3 von Seite 45, dort existierten ungefähr 2,5 mal so viele Knoten.

Aber auch wenn die Dokumente mehr werden, gibt es immer noch Knoten, die keine Last haben. Dies wird in Teil (v) vorgestellt. Die Werte haben aber nicht so hohe Ausschläge wie bei Chord ohne Lastbalancierung.

6.2.2.1 Bewertung

Es wurde gezeigt, dass die Lastbalancierung Power of two Choices deutlich besser ist als Chord ohne Lastbalancierung. Es gibt aber noch immer deutliche Unterschiede in der Last der einzelnen Knoten und einige Knoten haben immer noch keine Daten gespeichert.

6.2.3 Virtuelle Server

Als letztes der existierenden Lastbalancierungsalgorithmen für DHTs wird die Simulation von Chord mit virtuellen Servern vorgestellt. Jeder Knoten hat dabei $\log_2(N)$ virtuelle Server, die er verwalten muss. Zum Abschluss der Simulation wurde mehrere Runden lang die Last weiterhin balanciert.

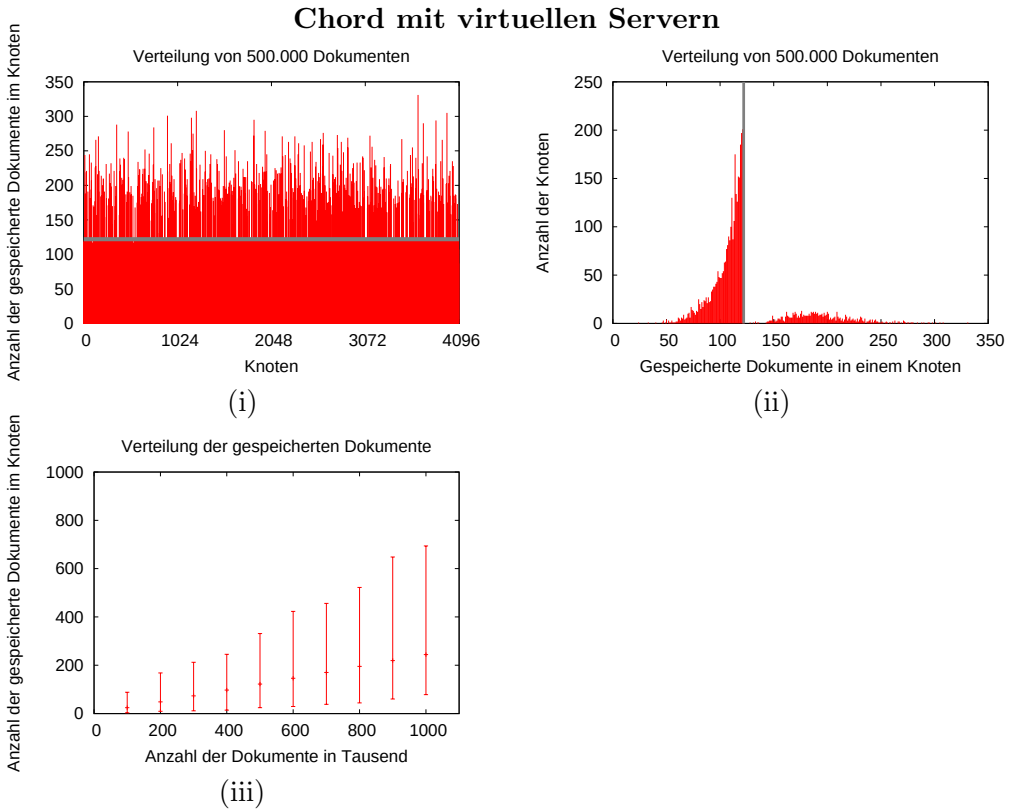


Abbildung 6.5: Simulation: Chord mit virtuellen Servern

Die Abbildung 6.5(i) zeigt als erstes wieder die Verteilung von Dokumenten in Chord mit Virtuellen Servern. Es wurden 500.000 Dokumente zufällig auf die 4.096 Knoten verteilt. Es wird deutlich, dass die Dokumente regelmäßiger auf die einzelnen Knoten verteilt wurden als ohne Lastbalancierung.

Die Abbildung 6.5(ii) zeigt die Verteilung der Dokumente auf die Knoten im gleichen Versuch. Es ist zu sehen, dass die Last besser verteilt wurde, es gibt aber immer noch starke Schwankungen in der Last, die ein Knoten zu bewältigen hat. Die größte Anzahl an Dokumenten ist in etwa gleich zu Power of two. Viele Knoten befindet sich direkt unter dem Durchschnitt bei der Anzahl der Knoten. Direkt über dem Durchschnitt sind kaum Knoten und einige Knoten immer noch sehr viele Daten gespeichert. Das direkt hinter dem Durchschnitt kaum Knoten sind, liegt daran, dass versucht wird virtuelle Server zu verschieben die einen Knoten leicht machen. Bei Knoten mit knapp mehr Dokumenten als die Ziellast kann so ein passender virtueller Server leicht gefunden werden und an einen leichten Knoten abgegeben werden. Da bei dieser Verschiebung der leichte Knoten nicht schwer werden darf häufen sich die Knoten unterhalb des Durchschnittes. Knoten die viel Last haben finden dagegen kaum einen Knoten, der ihre Last noch abnehmen kann.

Die Abbildung 6.5(iii) gibt nun eine Verteilung von Dokumenten auf den Knoten bei einer verschiedenen Gesamtanzahl von Dokumenten an. Es wurden auf die 4.096 Knoten zwischen 100.000 und 1.000.000 Dokumente verteilt. Der oberste Wert gibt an, wieviele Dokumente maximal, der untere wieviele minimal auf einem Knoten gespeichert wurden. Der mittlere Wert gibt die optimale Auslastung an. Auch wenn die Dokumente mehr werden, gibt es immer noch Knoten, die kaum Last haben Last haben. Die Werte sind aber bis jetzt die besten, bedeuten aber auch starken Aufwand.

6.2.3.1 Bewertung

Die virtuellen Server liefern bis jetzt das beste Ergebnis bei der Verteilung der Last. Für diesen Erfolg ist aber ein starker Aufwand nötig. Es werden viele virtuelle Server pro Knoten gespeichert, die alle verwaltet werden müssen. Doch muss der neue vorgestellte Lastbalancierungsalgorithmus vor allem gegen diese Ergebnisse bestehen. Es muss dabei vor allem die Spannbreite der gespeicherten Last in den verschiedenen Knoten geringer werden.

6.2.4 Verwendeter Lastbalancierungsalgorithmus

Als letzter Algorithmus in der Simulation nun der vorgestellte Lastbalancierungsalgorithmus für DHTs.

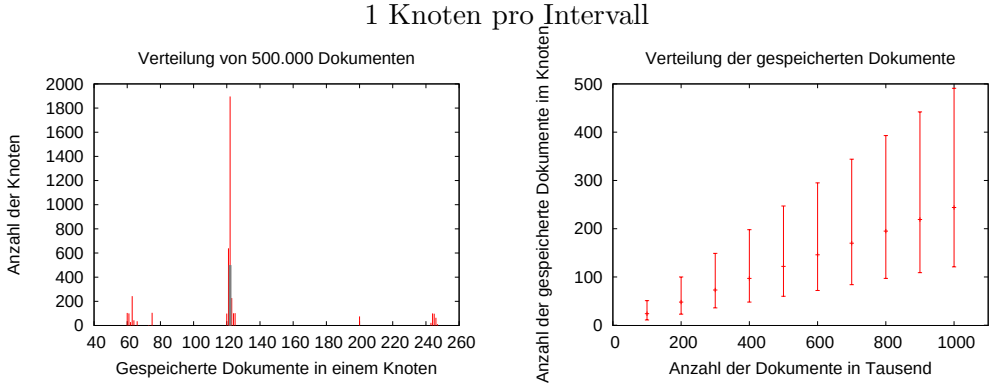
Bei der Simulation wird das Intervall immer ungefähr bei der Hälfte der Dokumente geteilt. Bei der Auswahl der Einträge bei predecessor, successor oder Fingern in den Finger-Tabellen wird ein Knoten zufällig aus den Knoten ausgewählt, die ein Intervall bearbeiten. Es wird keine Betrachtung nach Nähe oder Latenzzeit gemacht. Es wird versucht neue Knoten zu den Intervallen hinzuzufügen, die noch wenig Knoten haben. Wenn genug Knoten ein Intervall bearbeiten, wird es nach den in Abschnitt 6.2.4 beschriebenen Regeln aufgeteilt.

Es wurde bei der Simulation des Lastbalancierungsalgorithmus darauf verzichtet, die Last mit den direkten Nachfolger zu vergleichen und die Intervalle eventuell neu aufzuteilen (Dritter Schritt in Abschnitt). Wenn dieser Schritt oft genug ausgeführt wird kann eine nahezu Gleichverteilung der Last erreicht werden. Dabei werden aber viele Dokumente verschoben und viel Last im Netz erzeugt.

Die Abbildung 6.6 zeigt nun die Ergebnisse der Simulation. Als erstes wurde zum besseren Vergleich mit den anderen Algorithmen festgelegt, dass nur ein Knoten ein Intervall mindestens zu bearbeiten braucht. Später wird noch gezeigt, wie die Ergebnisse aussehen, wenn mindestens 3 Knoten ein Intervall bearbeiten müssen.

Zum Abschluss der Simulation wurde die Last genauso viele Runden lang weiter balanciert, wie bei den virtuellen Servern.

Chord mit dem verwendeten Lastbalancierungsalgorithmus
1 Knoten pro Intervall



(i) (ii)

Abbildung 6.6: Simulation: Chord mit dem verwendeten Lastbalancierungsalgorithmus

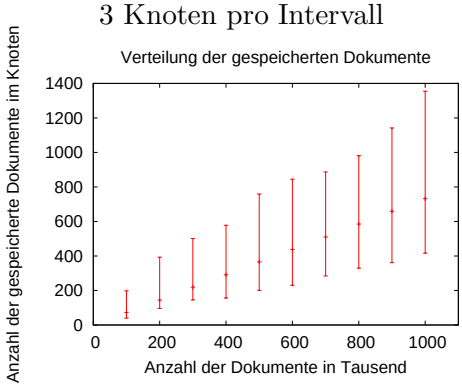
Die Abbildung 6.6(i) zeigt als Beispiel die Verteilung der 500.000 Dokumente auf die Knoten. Die meisten Knoten haben dabei Last um das Optimum herum. Eine kleinere Anzahl hat mehr oder weniger Dokumente. Der Anstieg am Ende ist damit zu erklären, dass diese Knoten zu viel Last haben, aber keine Knoten mehr zum Aufteilen. Die Last ist aber weitaus besser verteilt als bei den Algorithmus mit virtuellen Servern.

Die Abbildung 6.6(ii) gibt nun an wie die Last von 100.000 bis 1.000.000 Dokumenten auf 4.096 Knoten verteilt wurde. Der oberste Wert gibt an wieviele Dokumente maximal, der untere wieviele minimal auf einem Knoten gespeichert wurden. Der mittlere Wert gibt die optimale Auslastung an. Es haben weniger Knoten als bei

den virtuellen Servern keine Last und die Schwankungen sind geringer.

Es wird nun gezeigt, wie die Ergebnisse aussehen, wenn mindestens drei Knoten ein Intervall bearbeiten müssen.

Chord mit dem verwendeten Lastbalancierungsalgorithmus
3 Knoten pro Intervall



(ii)

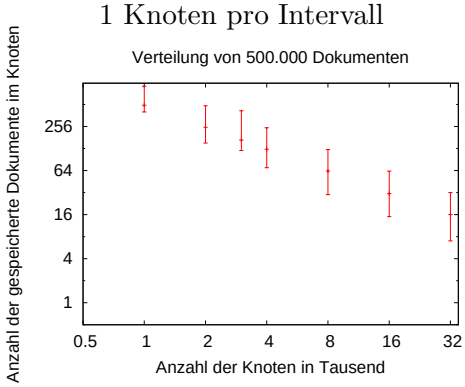
Abbildung 6.7: Simulation: Chord mit dem verwendeten Lastbalancierungsalgorithmus

Die Abbildung 6.7 zeigt die Verteilung der Dokumente auf die 4.096 Knoten. Es ergeben sich die gleichen Aussagen, wie bei den Experimenten mit 1 Knoten pro Intervall, die Last ist wieder ausgeglichener als ohne Lastbalancierung verteilt. Die Optimale Last ist jetzt jedoch drei mal so hoch, da eigentlich nur noch ein Drittel der Knoten für neue Intervalle zur Verfügung stehen. Die Last wird aber insgesamt schlechter als bei Intervallen mit nur mindestens einem Knoten verteilt, da viel weniger Knoten zur Verfügung stehen und die Verteilung der Knoten im System dadurch unflexibler wird.

6.2.4.1 Auswirkungen der Knotenanzahl

Als nächstes wurde die Auswirkungen der Knotenanzahl auf die Verteilung der Dokumente untersucht. In Abbildung 6.8 wurden 500.000 Dokumente in Systemen mit 1000 bis 32000 Knoten verteilt.

Chord mit dem verwendeten Lastbalancierungsalgorithmus
1 Knoten pro Intervall



(i)

Abbildung 6.8: Simulation: Chord mit dem verwendeten Lastbalancierungsalgorithmus

Bei jeder Knotenanzahl haben einige Knoten höchstens doppelt soviel Last wie bei optimaler Verteilung.

6.2.4.2 Verschobene Last

Als letztes soll die Last beschrieben werden, die zwischen Knoten verschoben wurde. Last wird nur in wenigen Fällen verschoben. Wenn ein Intervall geteilt wird, müssen die Knoten nur Dokumente (also Last) löschen. Beim verschieben von Knoten muss auf diesem die Last ebenfalls erstmal nur gelöscht werden. Die einzige Last die kopiert wird ist die, wenn ein Knoten einem bestehenden Intervall hinzugefügt wird.

6.2.4.3 Bewertung

Es wurde gezeigt, dass die Last bei dem vorgestellten Algorithmus am besten verteilt wurde. Dabei werden Dokumente nur zwischen Nachbarn kopiert. Dagegen verschieben die Knoten bei den virtuellen Server immer ganze Server, so dass die verschobene Last insgesamt ausgeglichen ist. Die Knoten bei den virtuellen Server haben aber zusätzlich mehr Verwaltungsaufwand.

6.3 Fazit

Es wurde mit der Simulation die Ergebnisse der einzelnen Verfahren gezeigt, verglichen und bewertet. Mit der Simulation konnten große Knoten- und Datenmengen im Chord-Ring beobachtet werden.

Es hat sich gezeigt, dass der vorgestellte Lastbalancierungsalgorithmus die Last am Besten von allen vorgestellten und getesteten Algorithmen auf die zur Verfügung stehenden Knoten verteilt.

Da aber in dem realen Einsatz unter Umständen andere Faktoren zum Tragen kommen wurde der Algorithmus implementiert. Bei Tests mit der Implementation kann aber nicht eine solche große Anzahl von Knoten und Dokumenten erzeugt werden. Die nächsten beiden Kapitel beschreiben die Implementierung und die Ergebnisse der Messungen.

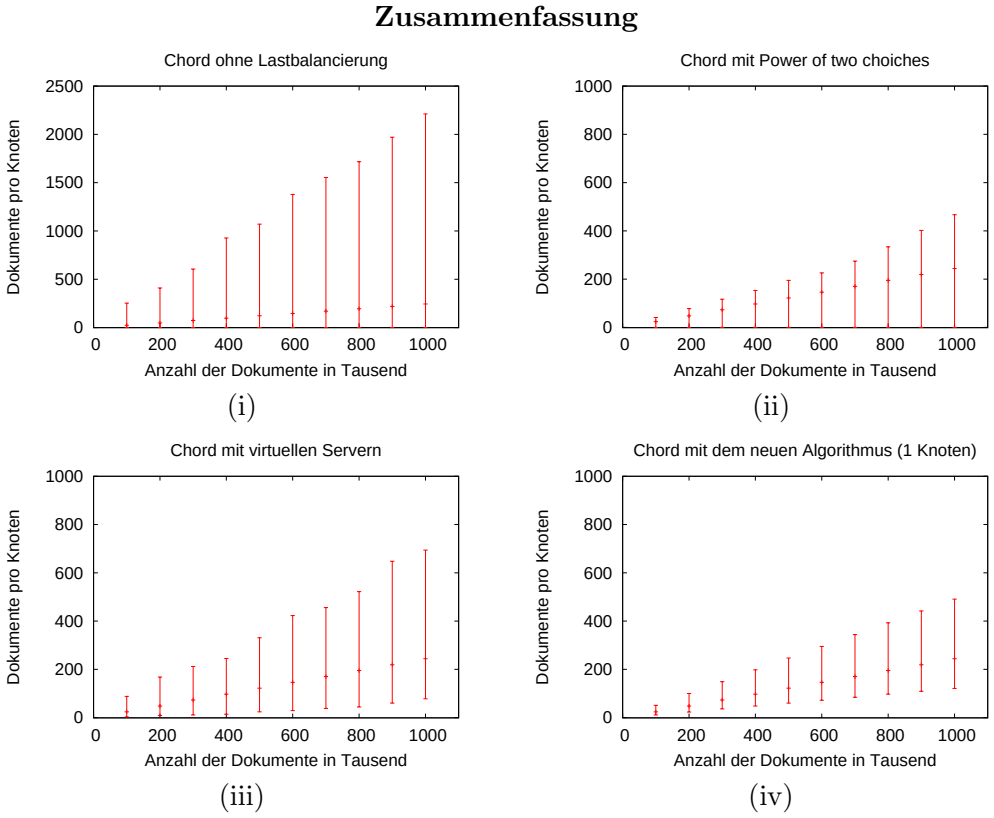


Abbildung 6.9: Simulation: Zusammenfassung der Ergebnisse der vorherigen Messungen.

It was new. It was singular.
It was simple. It must succeed!

– Horatio Nelson (1805)

Kapitel 7

Implementierung

Dieses Kapitel beschreibt nach der Simulation nun die Implementierung des in Kapitel 5 vorgestellten Lastbalancierungsalgorithmus.

Durch die Implementation soll die Praxistauglichkeit des Algorithmus beobachtet und bewertet werden. Dazu werden im nächsten Kapitel die erfolgten Messungen des Systems vorgestellt.

Für den vorgestellten Lastbalancierungsalgorithmus wurde ein Client wie schon in der Simulation in der Programmiersprache Java erstellt. Es wurde ein neuer eigenständiger Client entwickelt, um die Lastbalancierung im Distributed Hash Tables System ohne eventuelle andere äußere Einflüsse durch andere Programmteile messen zu können.

Als Distributed Hash Tables System wurde, wie schon in der Simulation, das bereits vorgestellte System Chord verwendet.

Zuerst wird in diesem Kapitel auf die verwendete Technologie in der Java Implementierung des Clients eingegangen. So wurde das Kommunikationsframework Java Remote Method Invocation [60] zur Kommunikation der Knoten im Peer-to-Peer Netz Chord verwendet. RMI wird sowohl zur Kommunikation der Knoten untereinander im Chord-Ring, als auch zur Kommunikation des Benutzers mit beliebigen Knoten benutzt.

Anschließend wird der Client mit seinen Konfigurationsmöglichkeiten näher beschrieben. Es wird das Benutzerinterface sowohl des Clients, als auch eines Testwerkzeuges gezeigt, durch das mit dem durch die Knoten aufgebauten Chord-System interagiert werden kann. Es wird zusätzlich ein Beispiel zur Benutzung und Erzeugung eines kleinen Peer-to-Peer Systems mit Chord gegeben.

Abschließend wird die Implementierung und Interfaces der Klassen zur Kommunikation genauer beschrieben.

7.1 Verwendete Technologie

Es wurden bei der Implementation des Peer-to-Peer Clients bereits bestehende Technologie verwendet.

Das Java Remote Method Invocation Framework [60] wird zur Kommunikation der Knoten im Peer-to-Peer Netz verwendet, und im folgenden kurz vorgestellt.

7.1.1 Remote Method Invocation

Das Java Remote Method Invocation (RMI) [60] Protokoll ermöglicht die Kommunikation zwischen verteilten Objekten in Java [59]. Es befindet sich seit Version 1.1 im Java Software Development Kit (SDK).

Verschieden Java-Objekte rufen dabei Methoden anderer Objekte in derselben Weise auf, wie Methoden eines Objekts aufgerufen werden, das sich auf derselben virtuellen Maschine befindet.

Dabei werden Parameter und Rückgabewerte in RMI aber nur als Kopie übergeben. Dies bedeutet dass alle übergebenen Parameter oder Rückgabewerte serialisiert werden müssen. Dazu muss ein Objekt serialisierbar sein, oder nur Attribute aus Java Standardtypen besitzen.

Zwei wichtige Bestandteile von RMI sind die Schnittstelle und die Registrierung, die im Folgendem kurz vorgestellt werden.

7.1.1.1 Schnittstelle

RMI wird in diesem Abschnitt mit der klassische Client-Server Architektur beschrieben. Im Verlaufe dieses Kapitels wird bei der konkreten Beschreibung der Implementierung, genau auf die Möglichkeit eingegangen, wie mit RMI die Knoten sowohl Server als auch Client, also Knoten im Peer-to-Peer Netz sein können.

Jeder Server muss in RMI eine festgelegte Schnittstelle implementieren. In dieser werden alle öffentlichen (public) Methoden beschrieben, die andere Teilnehmern nutzen können sollen.

Diese Schnittstelle muss die Klasse java.rmi.Remote erweitern, damit von Java automatisch die Verbindungen zu anderen Knoten hergestellt werden können. Zudem müssen alle im Interface beschriebenen Methoden die java.rmi.RemoteException auslösen können, indem diese im throws-Teil des Methodenkopfes aufgenommen wird.

Die Implementation des Servers muss die Klasse java.rmi.Server.UnicastRemoteObject erweitern und die selbst erstellte, oben beschriebene, Schnittstelle implementieren.

Mit dem Programm Java RMI stub compiler (rmic) wird daraufhin aus der Klasse, die das entfernte Objekt implementiert, die benötigten Stub- und Skeleton-Klassen erzeugt.

Ein **Stub** ist ein clientseitiger Proxy zur Kommunikation der Objekte. In diesem Stub werden alle Schnittstellen des entfernten Objektes implementiert.

Dagegen ist der **Skeleton** der serverseitige Proxy. Dieser Proxy sucht bei Aufrufen mit der Remote Schicht nach dem Ort des entfernten Objektes, und leitet den Aufruf

zum entsprechenden Server weiter und serialisiert die Argumente und Rückgabewerte.

Die folgende Abbildung 7.9 zeigt die Kommunikation zwischen verschiedenen Objekten in RMI.

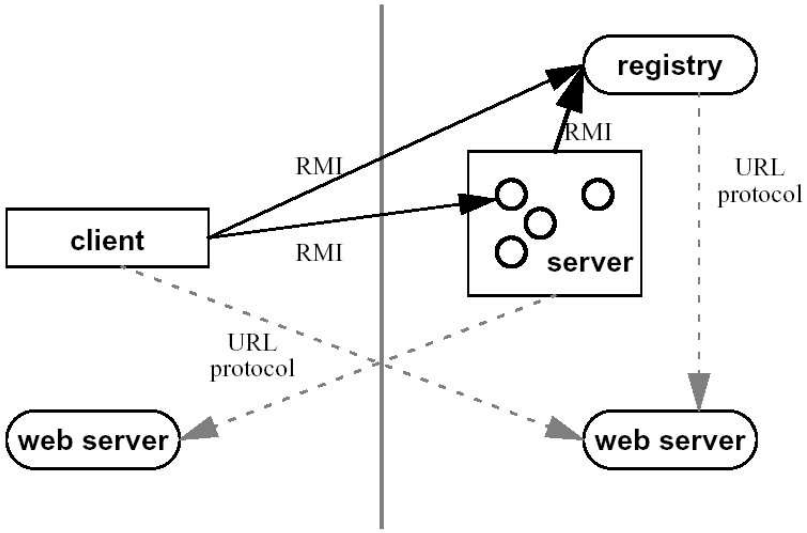


Abbildung 7.1: Java RMI: In dieser aus [60] entnommenen Abbildung wird die Kommunikation zwischen dem Client und dem Server über das RMI Protokoll gezeigt. Der Server registriert alle seine Objekt, die er veröffentlichen möchte in der Registry (siehe Abschnitt 7.1.1.2). Der Client fragt anschließend über RMI bei der Registry nach Objekten auf dem Server an und kann daraufhin direkt per RMI mit diesen kommunizieren. In dieser Abbildung kommunizieren der Client und der Server noch zusätzlich mit Webservern, was in dieser Implementierung nicht benötigt wird.

7.1.1.2 Registrierung

Jedes Objekt das entfernt aufrufbar sein soll, muss sich beim Java Namensdienst mit seiner IP-Adresse und seinem Namen anmelden. Dazu muss die Java Registrierung „Java remote object registry (rmiregistry)“ auf dem Rechner gestartet werden. Diese wartet daraufhin standardmäßig am Port 1099 auf Anfragen von anderen Objekten.

In dieser RMI-Registry können daraufhin beliebige Objekte anfragen, auf welche Objekte entfernt zugegriffen werden kann.

Dazu muss auf jedem Server der ein Objekt anbieten will eine solche RMI-Registry gestartet werden, auf einem Rechner können aber mehrere Objekte auch in mehreren virtuellen Maschinen eine einzige laufende Registry benutzen.

7.1.2 Hashfunktion

Zusätzlich zu RMI wurde noch die Hashfunktion aus Java benutzt.

Für die Platzierung der Dokumente in Chord wurde dazu die Standardhashfunktion für Strings verwendet. Diese generiert aus einem String eine eindeutige ganze natürliche Zahl i mit $0 \leq i \leq 2^{31} - 1$.

Diese Hashfunktion lässt sich laut Java API für die Java Version 1.4.2 wie folgt berechnen:

public int hashCode() berechnet einen Hashwert aus einem String mit Hilfe der Formel

$$s[0] \cdot 31^{(n-1)} + s[1] \cdot 31^{(n-2)} + \ldots + s[n-1]$$

wobei $s[i]$ der i-te Buchstabe des Strings, n die Länge des Strings ist.

7.2 Benutzung

Es wurde im Rahmen dieser Arbeit ein Client entwickelt, der mit Hilfe von RMI ein Chord-Ring aufbaut. Dabei wird zwischen diesen Clienten der vorgestellte neuentwickelte Lastbalancierungsalgorithmus zur Verteilung der Last eingesetzt.

Es wurde ein Client geschrieben welcher als Knoten im Distributed Hash Tables System Chord fungiert. Jeder dieser Knoten kann für sich autonom arbeiten und speichert nur IP-Adressen und Namen von anderen Knoten, wie zum Beispiel seinem successor.

Diese Knoten registrieren sich bei ihrem Namensdienst und warten dann auf Anfragen anderer Knoten oder des Benutzers über RMI.

Zur Kommunikation des Benutzers mit den Knoten ist ein zusätzliches Programm entstanden, das sich mit beliebigen Knoten im Chord-Ring verbinden kann. Der Testclient kann zum einem Statistiken, wie die Finger-Tabellen des Knoten, anzeigen, zum anderen Dokumente im Chord-System erzeugen, suchen oder löschen.

Bevor im nächsten Abschnitt detailliert auf die Implementierung der Knoten und des Testclienten eingegangen wird, soll in diesem Abschnitt zuerst die Konfiguration und Benutzung der Knoten und des Testclienten beschrieben werden. Abschließend folgt ein Beispiel.

7.2.1 Konfiguration

Der Benutzer kann beim Start seines ersten Knotens einige Parameter des Chord-Ringes einstellen, die die folgenden Knoten dann übernehmen. Im Testclient können dann die verschiedenen Knoten betrachtet, benutzt und ausgewertet werden.

7.2.1.1 Knoten

Beim Start eines Knotens können mittels Parameter die wichtigsten Einstellungen für den Chord-Ring übergeben werden. Für die anderen Einstellungsmöglichkeiten werden dann Vorgabewerte gewählt.

Auf einem Rechner können auch mehrere Knoten gestartet werden, um eine leichteren Test von mehreren Knoten auf einer begrenzten Anzahl von Computern zu ermöglichen.

Es besteht zusätzlich die Möglichkeit, wenn der Knoten ohne Parameter gestartet wird, eine individuelle Konfiguration des Ringes interaktiv vorzunehmen.

Im Einzelnen kann der Benutzer dabei folgende Einstellungen direkt beeinflussen:

- Die erste Konfigurationseinstellung verlangt vom Benutzer die Eingabe ob der Knoten der erste im Chord-Ring ist oder schon weitere existieren. Wenn schon weitere existieren wird nach der IP-Adresse eines Knotens gefragt, alle existierenden Knoten auf diesem Rechner angezeigt und vom Benutzer eine Auswahl des gewünschten Knotens verlangt.
- In allen Fällen wird ein Name für den Knoten verlangt, unter dem sich der Knoten im Ring anmeldet. Mit diesem Namen und der IP-Adresse können die Knoten untereinander in Kontakt treten und Daten austauschen.

Alle folgenden Punkte können nur dann gewählt werden, wenn noch kein Knoten im Ring existiert.

- Die Anzahl der Knoten, die ein Intervall mindestens zusammen bearbeiten müssen.
- Die Anzahl der Dokumente, ab dem ein Knoten als überlastet gilt.

7.2.1.2 Testclient

Beim Start des Test-Clienten kann die IP-Adresse (zum Beispiel localhost für den aktuell benutzten Rechner) des Knotens gewählt werden, mit dem der Benutzer verbunden werden möchte. Anschließend werden alle auf diesem Rechner laufenden Knoten angezeigt aus dem der Benutzer einen wählen kann.

7.2.2 Benutzerinterface

Beide Applikationen die entwickelt wurden sind zur Zeit reine textbasierte Applikationen auf der Konsole.

Das Benutzerinterface beider Komponenten wird in diesem Abschnitt näher beschrieben.

7.2.2.1 Knoten

Es existieren zwei Möglichkeiten, um eine neuen Knoten in einem Chord-Ring zu starten.

Entweder durch Angabe von Parametern beim Aufruf der main-Methode des Knotens oder interaktiv durch eine Abfrage der benötigten Daten durch den Knoten.

Der Aufruf mit Parametern eignet sich besonders zum automatisierten Erstellen von neuen Knoten. Wenn die Daten beim Aufruf direkt eingegeben werden hat der Kommandoaufruf folgende festgelegte Syntax.

Für den ersten Knoten wird mit „ java client.chord name“ ein neuer Knoten bei der lokalen Registry registriert und ein neuer Chord-Ring erzeugt.

Die folgenden Knoten können mit „ java client.chord existingNodeAdress existingNodeName name“ erzeugt werden. Dabei gibt existingNodeAdress die IP-Adresse eines

```
rieche@csr-pc6:~/Diplom/Implementierung> java client.Chord
  First Node ( y / n ) ? y
  TargetLoad:  200
  MinNeighbours:  10
  Name:  FirstNode
Wait ...
```

Abbildung 7.2: Neuer Chord-Ring: Erzeugung eines ersten Knotens in einem neuen Ring.

Knotens im Chord-Ring und existingNodeName den Namen des Knotens, mit dem er registriert wurde an. Schließlich gibt name an unter welchem Namen der neue Knoten auf der lokalen Maschine registriert werden soll.

7.2.2.2 Testclient

Der Testclient bietet, nachdem der Benutzer sich an einem Knoten angemeldet hat, ein Menü mit möglichen Operationen an, die durch den Benutzer im Ring ausgeführt werden können.

```
      chord client for the new loadbalancing algorithm
------------------------------------------------------------

  Node (e.g. localhost): localhost
  Nodes at localhost: FirstNode
  Which node to connect: FirstNode
  Success, id of node: 0

  Node:      (0) change Node (1) list of neighbours
  Document:  (2) create      (3) find
             (4) delete      (5) list of documents
  Routing:   (6) successor   (7) predecessor
             (8) fingerTable (9) list of successors

             (10) exit
```

Abbildung 7.3: Optionen: Alle möglichen Optionen des Testclienten nachdem ein beliebiger Knoten kontaktiert wurde. Diese und die folgenden Ausgaben wurden zum Teil auf die Seitenbreite dieser Arbeit angepasst.

Mit der Option 0 kann der Client mit einem neuen Knoten verbunden werden. Mit der Option 1 wird eine Liste aller Knoten ausgegeben, die das gleiche Intervall bearbeiten.

Mit der Option 2 kann der Benutzer ein neues Dokument erzeugen und in den Chord-Ring einfügen lassen, mit 3 kann ein Dokument gesucht werden, und schließlich mit 4 ein Dokument aus dem Ring gelöscht werden. Bei Wahl der 5. Option wird eine

Liste aller auf diesem Knoten gespeicherten Dokumente ausgegeben.

Bei der Option 6 und 7 wird der successor bzw. predecessor ausgegeben, bei 8 die komplette Finger-Tabelle des Knotens. Bei Wahl der Option 10 wird vom Knoten eine Kettenabfrage der Nachfolger gestartet, bis der Ring einmal umrundet wird. Jeder Knoten gibt dabei eine Statistik über seine Adresse, Nachbarn und Dokumente zurück.

7.2.3 Beispiel

Als Beispiel soll mit den Clienten ein Chord-Ring mit sechs Knoten erstellt werden.

Zuerst wird die Konfiguration und Erzeugung der Knoten gezeigt, abschließend der Test des erzeugten Chord Systems mit dem Testclienten.

7.2.3.1 Knoten

Es sollen sechs Knoten auf zwei Computern gestartet werden, die untereinander einen Chord-Ring aufbauen sollen.

Zuerst wird ein erster Knoten erstellt, der das Chord System erzeugt.

```
rieche@lin220: java client.Chord FirstNode
```

Abbildung 7.4: Beispiel: Erzeugung eines neuen Knotens auf dem Rechner lin220 mit dem Namen FirstNode.

Als Ziellast wurde drei Dokumente pro Intervall gewählt. Außerdem reicht es wenn mindestens ein Knoten ein Intervall bearbeitet.

Daraufhin können beliebige Knoten am Distributed Hash Tables System teilnehmen. Bei der Anmeldung kann dazu ein beliebiger existierender Knoten verwendet werden.

```
rieche@lin220: java client.Chord 160.45.45.220 FirstNode  SecondNode
rieche@lin220: java client.Chord 160.45.45.220 SecondNode ThirdNode
rieche@lin213: java client.Chord 160.45.45.220 FirstNode  FourthNode
rieche@lin213: java client.Chord 160.45.45.220 SecondNode FifthNode
rieche@lin213: java client.Chord 160.45.45.213 FourthNode SixthNode
```

Abbildung 7.5: Beispiel: Erzeugung weiterer Knoten auf den Rechnern lin220 (IP-Adresse 160.45.45.220) und lin213 (IP-Adresse 160.45.45.213) mit dem Namen SecondNode bis SixthNode. Den Knoten werden unterschiedliche existierende Knoten genannt.

Dabei wird ein Chord-Ring mit sechs Knoten erzeugt. Da noch überhaupt keine Last im Ring vorhanden ist bearbeiten alle Knoten gemeinschaftlich das ganze Intervall.

Die folgende Abbildung zeigt den Aufbau des Rings, der durch dieses Beispiel entstanden ist.

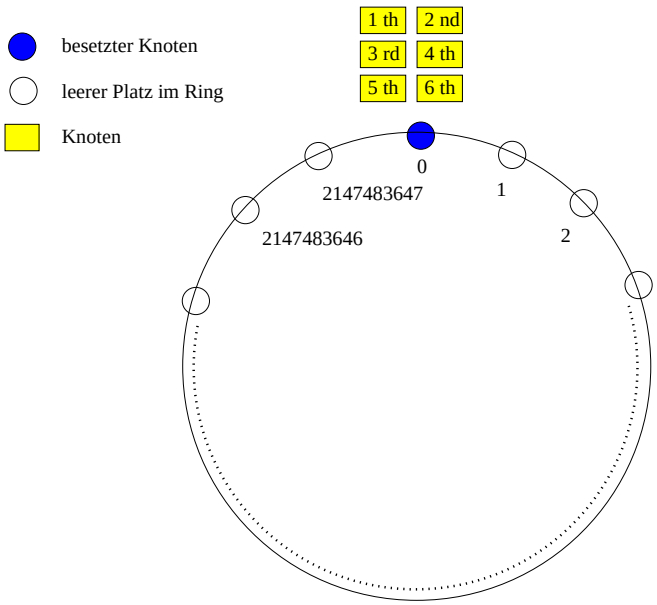


Abbildung 7.6: Beispiel: Die sechs Knoten sind alle bei der HashID 0 im Ring angeordnet. Der Ring hat eine Größe von 2^{31}.

7.2.3.2 Test-Client

Das oben beschriebene System hat einen Chord-Ring erzeugt. Um den Ring zu testen und Dokumente einzufügen, muss eine Verbindung des Test-Clienten mit einem beliebigen Knoten, in diesem Fall mit dem ersten Knoten hergestellt werden. Dabei ergibt sich zum Beispiel folgendes Bild.

```
rieche@linux:~/Diplom/Implementierung> java client.TestClient

      chord client for the new loadbalancing algorithm
------------------------------------------------------------

  Node (e.g. localhost): 160.45.45.220
  Nodes at 160.45.45.220: FirstNode  SecondNode  ThirdNode
  Which node to connect: FirstNode
  Success, id of node: 0
...
```

Abbildung 7.7: Beispiel: Verbindung des Test-Clienten mit dem Knoten FirstNode auf Rechner 160.45.45.220 (lin220).

Anschließend können zum Beispiel beliebig viele Dokumente in das System eingefügt werden. Nachdem nun zwölf Dokumente erzeugt und im Ring gespeichert wurden sind die Knoten im Ring verteilt worden. Durch Wahl der successor-Liste ergibt sich nun folgendes Bild über die Verteilung der Knoten im Ring.

```
Please choose action: 9
 //160.45.45.213/SixthNode with 0 Neighbours
   with ID 0 and storing 2 Documents
 //160.45.45.220/ThirdNode with 1 Neighbours
   with ID 122 and storing 3 Documents
 //160.45.45.213/FourthNode with 0 Neighbours
   with ID 115312 and storing 2 Documents
 //160.45.45.213/FifthNode with 0 Neighbours
   with ID 3152400 and storing 3 Documents
 //160.45.45.220/FirstNode with 0 Neighbours
   with ID 3747706 and storing 2 Documents
```

Abbildung 7.8: Beispiel: Während 12 Dokumente erzeugt wurden, haben die Knoten den Ring untereinander in 5 Intervalle aufgeteilt. Da kein weiteres Intervall zu viele Dokumente speichert, bearbeiten zwei Knoten weiterhin ein Intervall.

Die folgende Abbildung veranschaulicht nochmals die neue Verteilung der Knoten im Chord-Ring.

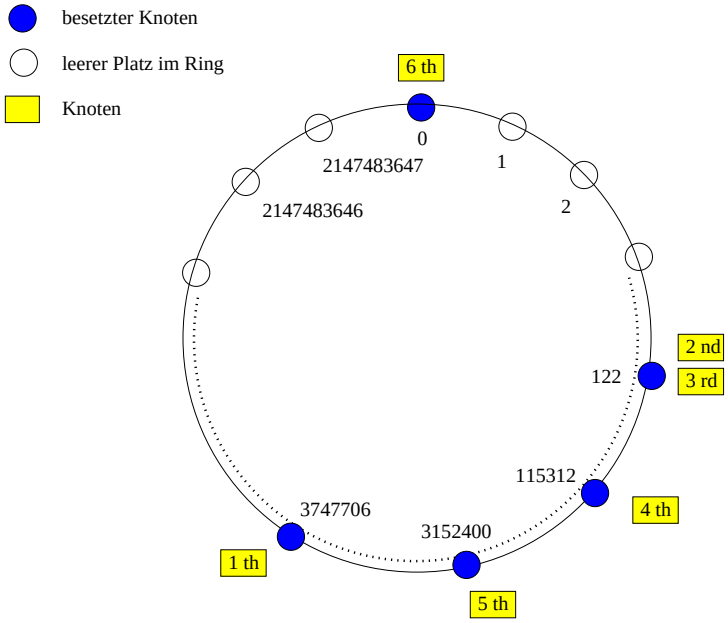


Abbildung 7.9: Beispiel: Die sechs Knoten sind im Chord-Ring der Größe 2^{31} verteilt.

Da das System die Lastbalancierung anhand der Last der Knoten vornimmt, also anhand der gespeicherten Dokumente, kann das System bei unterschiedlich erzeugten Dokumenten geringfügig voneinander abweichen.

7.3 Implementierung

Die Abbildung A.1 auf Seite 99 zeigt ein UML-Klassendiagramm der Implementierung.

7.3.1 Chord

Alle Klienten sind Knoten eines Chord-Ringes. Der Ring entspricht dem für Chord in Abschnitt 3.4 beschrieben Aufbau.

Jeder Knoten speichert die Adresse seines Vorgängers und Nachfolgers im Ring sowie seine Nachbarn, die das gleiche Intervall bearbeiten. Als successor wird zufällig ein Knoten aus der Menge der Knoten gewählt, die das nächste Intervall bearbeiten. In der Finger-Tabelle speichert jeder Knoten die Adresse des entsprechenden Knotens für jeden Finger.

Der originale Algorithmus von Chord wurde dabei komplett in den Knoten implementiert, nicht jedoch die gesamten Stabilisierungsroutinen.

Ausgefallene Nachbarn werden erkannt und aus der Liste der Knoten entfernt die das gleiche Intervall bearbeiten. Beim Ausfall des successors oder predecessors wird versucht ein Ausweichknoten zu finden. Ein Ausfall ganzer Regionen des Rings kann das System jedoch nicht kompensieren.

7.3.2 Lastverteilung

Die Lastverteilung der Knoten entspricht dem in dieser Arbeit in Kapitel 5 beschriebenen Lastbalancierungsalgorithmus.

Ein Knoten versucht die Last immer nur dann zu balancieren, wenn ein Dokument bei ihm gespeichert oder gelöscht wurde.

Nachdem alle Dokumente eingefügt wurden wird noch weiter versucht die Last der Knoten besser zu balancieren.

7.3.3 Hashfunktion

Als Hashfunktion für die Platzierung der Dokumente in Chord wurde die Standardhashfunktion von Java benutzt. Diese generiert aus einem String eindeutig eine ganze natürliche Zahl i mit $0 \leq i \leq 2^{31} - 1$.

7.3.4 Kommunikation

Alle Knoten sind in der Implementierung sowohl Client als auch Server. Dazu verbindet sich ein Knoten, wenn er Daten von anderen Knoten benötigt, anhand der gespeicherten Adresse über RMI mit dem anderen Knoten. Der Client registriert seinen Knoten beim Namensdienst, an den sich beliebige andere Knoten nun wenden können.

7.3.5 Interface

Das Interface Chord-Client aus der UML-Abbildung bildet für andere Knoten oder Objekte die Zugangsmöglichkeit über RMI, um mit dem Knoten zu kommunizieren. Die Abbildung 7.1 auf der nächsten Seite zeigt eine Auswahl der wichtigsten Methoden aus der mit Javadoc erzeugten Dokumentation des Interfaces. Die Beschreibungen wurden aus dem Englischen zurückübersetzt.

7.3.6 Fazit

Es wurde eine Implementierung von Chord mit dem Lastbalancierungsalgorithmus aus Kapitel 5 vorgestellt. Es wurden die Konfigurationsmöglichkeiten, die Benutzung und der Aufbau sowohl des Clienten, der einen Knoten im Chord-Ring darstellt, als auch eines Testwerkzeuges zur Interaktion mit dem Distributed Hash Tables System vorgestellt.

Im folgenden Kapitel werden die am System vorgenommen Messungen erläutert und bewertet.

Rückgabewert	Signatur und Beschreibung
void	deleteDocument(long HashID) Ein Dokument mit dem Hashwert HashID aus dem System löschen.
java.lang.String	findDocument(long ID) Sucht ein Dokument mit dem Hashwert ID.
java.lang.String	findPredecessor(long ID) Findet die Adresse des predecessors (Vorgänger) vom Hashwert ID.
java.lang.String	findSuccessor(long ID) Findet den successor (Nachfolger) vom Hashwert ID.
java.lang.String	getAdress() Gibt die RMI-Adresse des Knotens vollständig zurück.
int	getDimension() Die Dimension des Chord-Ringes (m aus der Beschreibung).
java.util.Vector	getDocuments() Gibt alle im Knoten gespeicherten Dokumente als Vektor zurück.
long	getID() Gibt den Hashwert des Knotens zurück.
int	getMinNeighbours() Minimale Anzahl an Knoten, die ein Intervall mindestens bearbeiten müssen.
java.util.Vector	getNeighbours() Gibt ein Vektor mit Adressen von Knoten zurück, die das gleiche Intervall bearbeiten.
java.lang.String	getPredecessor() Gibt die Adresse des aktuellen predecessors (Vorgänger) zurück.
java.lang.String	getSuccessor() Gibt die Adresse des aktuellen successors (Nachfolger) zurück.
int	getTargetLoad() Gibt die gewünschte Ziellast des Knotens zurück.
boolean	isHeavy() True (wahr) wenn der Knoten mehr Daten als gewünscht gespeichert hat.
void	storeDocument(java.lang.String name, long HashID) Speichert ein Dokument mit gegebenen Namen und Hashwert im Ring.
java.util.Vector	successorList() Gibt eine Liste aller Nachfolger aus, bis der Ring umrundet wurde.
java.lang.String	writeFingerTable() Gibt einen String mit der Finger-Tabelle des Knotens zurück.

Tabelle 7.1: Interface: Die wichtigsten Methoden aus dem Interface ChordClient. In der 1. Spalte sind die Rückgabewerte beschrieben, in der 2. Spalte die Signatur und die Aufgabe der Funktion.

The way you measure is more important
than what you measure.

– Art Gust, Audio Equipment Developer

Kapitel 8

Messungen

Das letzte Kapitel hat eine Implementierung des vorgestellten Lastbalancierungsalgorithmus in der Programmiersprache Java vorgestellt.

Durch die Implementation soll die Praxistauglichkeit des Algorithmus beobachtet und bewertet werden können. In diesem letzten Kapitel vor der Zusammenfassung soll nun die Messungen, an den von den Knoten der Implementierung erstellten Distributed Hash Tables System Chord beschrieben werden.

Als erstes soll in diesem Kapitel die Testumgebungen vorgestellt werden.
Die Messungen wurden zum einem an ElFie [66], einem Linuxcluster mit 16 Rechnern und insgesamt 32 Prozessoren im ZIB durchgeführt, sowie auf Linux-PCs des Instituts für Informatik der Freien Universität Berlin durchgeführt. Die genauere Ausstattung der Rechner wird in dem Abschnitt 8.1 vorgestellt.

Anschließend werden die Testparameter vorgestellt. Das sind die Einstellungen, die zwischen den Messungen am System verändert wurden. Dabei wurden hauptsächlich die Anzahl der Knoten, die Anzahl der Dokumente und die Anzahl der Knoten, die ein Intervall mindestens bearbeiten müssen verändert.

Nach einem ersten Beispiel, werden die Ergebnisse der einzelnen Messungen vorgestellt und abschließend bewertet.

8.1 Testumgebung

Sämtliche folgende Messungen wurden an ElFie [66] und Linux-PCs vorgenommen.

- **ElFie**
  ElFie ist ein vom Zuse Institute Berlin (ZIB) betriebener Linux-Cluster.

  Der Cluster besteht aus 16 DELL 530 MT Knoten, die jeweils zwei Pentium 4 Xeon Prozessoren haben. Ein DELL PowerEdge 4600 dual Xeon ist der Server mit 0.5 Terabyte Plattenplatz. Alle Knoten sind mit 1 GB Arbeitsspeicher ausgestattet und sind untereinander via Myrinet [36] und Fast-Ethernet verbunden.

  Als Java Virtual Maschine (VM) wurde die Version 1.4.2.02 des Java Software Development Kits (SDK) verwendet.

- **Linux-PCs**
  Die verwendeten Linux PCs lin210 bis lin220 sind Rechner des Instituts für Informatik der Freien Universität.

  Sie besitzen eine Intel Pentium III 1000 MHz Prozessor und verfügen über 256 MB RAM. Die Linux PCs haben einen Linux-Kernel 2.4.21.

  Als Java VM wurde die Version 1.4.2.01 des Java SDKs verwendet.

8.2 Testparameter

Bei den Test wurden verschiedene Parameter verändert.

- **Die Anzahl der beteiligten Knoten**
  Durch Veränderung der Anzahl der insgesamt im Distributed Hash Tables System beteiligten Knoten, bei zum Beispiel gleichbleibender Anzahl von Dokumenten, kann untersucht werden, ob die Last mit steigender Anzahl von Knoten eventuell besser auf die beteiligten Knoten im DHT verteilt wird.

- **Die Anzahl der Knoten die ein Intervall mindestens bearbeiten müssen.**
  Durch Erhöhung der Fehlertolranz des System, stehen weniger Knoten für neue Intervalle zu Verfügung. Je mehr Knoten ein Intervall mindestens bearbeiten müssen, um so unwahrscheinlicher ist es Knoten zu finden, die Last von einem anderen Knoten abnehmen können.

- **Die Anzahl der im System gespeicherten Dokumente.**
  Wenn bei gleichbleibender Knotenanzahl die Anzahl der Dokumente erhöht oder verringert wird, müssen die Knoten die Last neu verteilen.

8.3 Beispiel

Als erstes soll ein Test als Beispiel dafür gezeigt werden, wie die Messergebnisse erzielt wurden.

Folgende Abbildung 8.1 zeigt die Last der Knoten mit folgenden Parametern.

- Insgesamt 5 Rechner mit je 15 Knoten auf den Rechner im Cluster, also 75 Knoten insgesamt.
- Nur einem Knoten, der ein Intervall mindestens bearbeiten muss.
- 1500 Dokumenten, die in das System eingefügt wurden.

Die folgende Abbildung 8.1 zeigt beispielhaft die Ergebnisse der Messung.

Dabei sind auf der Abszisse die Knoten von 1 bis 75 durchnummeriert. Auf der Ordinate sind die Anzahl der Dokumente, die ein Knoten gespeichert hat eingezeichnet. Die mittlere Linie in der Abbildung gibt die Last bei idealer Verteilung der 1500 Dokumenten auf die 75 Knoten an.

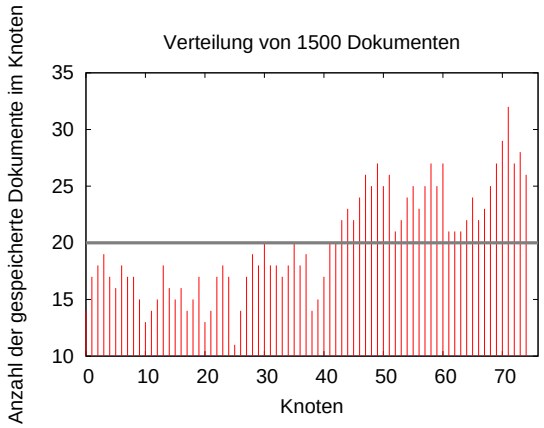


Abbildung 8.1: Beispiel: Die Last von 1500 Dokumenten wurde auf 75 Knoten in einem Chord-Ring mit dem vorgestelltem Lastbalancierungsalgorithmus verteilt. als Ergebnis ist zu sehen, dass es keine Knoten ohne Last und wenige mit stark erhöhter Last gibt.

Die Last ist fast ausgeglichen Verteilt. Die Last am Ende ist erhöht weil, die Knoten keine anderen Intervalle mehr gefunden haben, die ihnen Knoten abgeben können. Der dritte Schritt des Algorithmus wurde nur dann angewandt wird, wenn der Nachfolger oder Vorgänger stark erhöhte Last hat, um nicht zu viel Last zu verschieben zu müssen.

8.4 Messungen

Im Folgenden werden die Ergebnisse der vorgenommenen Messreihen detailliert vorgestellt.

8.4.1 Variation der Dokumente bei einem Knoten pro Intervall

Als erstes wurde die Anzahl der Dokumente variiert während die Knotenanzahl konstant geblieben ist. Es muss nur mindestens ein Knoten ein Intervall bearbeiten.

Das Ziel der Messung ist die Untersuchung der Auswirkung der Anzahl der Dokumenten auf der Verteilung der Last.

Die Parameter wurden nach Abschnitt 8.2 wie folgt gewählt:

- Insgesamt 75 Knoten auf dem Rechner im Cluster oder der FU Berlin.
- Nur einem Knoten, der ein Intervall mindestens bearbeiten muss.

8.4.1.1 Ergebnisse

Die folgende Abbildung zeigt die Ergebnisse der vorgestellten Messung.

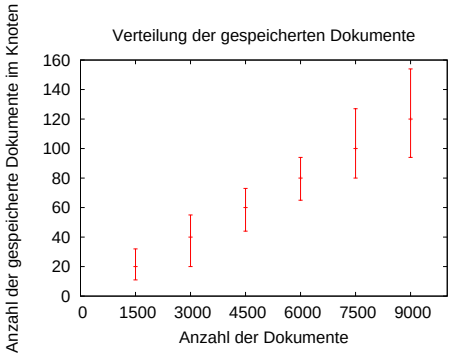


Abbildung 8.2: Messung: Variation der Dokumente mit 1 Knoten pro Intervall.

Der mittlere Strich bei jedem Ergebnis gibt den idealen Wert der Verteilung der Dokumente auf die 75 Knoten an. Der obere Strich markiert die höchste Last, die in einem der 75 Knoten vorkommt, der untere die geringste Last.

8.4.1.2 Bewertung

Die Last ist ausgeglichener verteilt, als die anderen Verfahren virtuelle Server und Power of two Choices angeben. Das Ergebnis entspricht in etwa denen der Simulation. Es gibt keine Knoten die keine Last haben.

Die Abweichungen nach oben und unten sind bei jeder Anzahl von Dokumenten in etwa gleich groß, was aber bei großer Anzahl von Dokumenten prozentual weniger ist.

8.4.2 Variation der Dokumente bei drei Knoten pro Intervall

Als nächstes wurde wieder die Anzahl der Dokumente variiert, während die Knotenanzahl weiterhin konstant geblieben ist. Es bearbeiteten nun aber mindestens drei Knoten ein Intervall.

Es soll mit der Messung untersucht werden, wie sich die Verteilung der Last ändert, wenn mehr Dokumente ins System eingefügt werden. Dabei sollen zur Erhöhung der Fehlertoleranz mindestens 3 Knoten ein Intervall bearbeiten.

Die Parameter wurden nach Abschnitt 8.2 wie folgt gewählt:

- Insgesamt 75 Knoten auf dem Rechner im Cluster.
- Drei Knoten, die ein Intervall mindestens bearbeiten müssen.

8.4.2.1 Ergebnisse

Die folgende Abbildung zeigt die Ergebnisse der vorgestellten Messung.

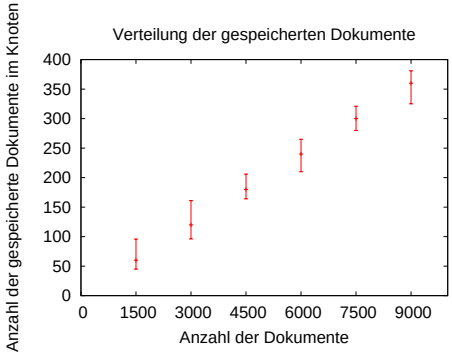


Abbildung 8.3: Messung: Variation der Dokumente mit 3 Knoten pro Intervall.

Der mittlere Strich bei jedem Ergebnis gibt wieder den idealen Wert der Verteilung der Dokumente auf die 75 Knoten an. Der obere Strich markiert die höchste Last, die in einem der 75 Knoten vorkommt, der untere die geringste Last.

8.4.2.2 Bewertung

Die Verteilung der Dokumente bei 3 Knoten, die ein Intervall mindesten bearbeiten müssen, weicht kaum von den Ergebnissen bei einem Knoten pro Intervall ab.

8.4.3 Variation der Knoten pro Intervall

Es wurde anschließend die Anzahl der Dokumente und der Knoten konstant gehalten. Es bearbeiteten nun aber unterschiedlich viele Knoten mindestens ein Intervall.

Mit dieser Messung, soll untersucht werden, wie sich die Verteilung der Last ändert, wenn die Anzahl der Knoten die ein Intervall bearbeiten verändert wird.

Die Parameter wurden nach 8.2 wie folgt gewählt:

- Insgesamt 75 Knoten auf dem Rechner im Cluster.

- 5000 Dokumente, die in das System eingefügt wurden.

8.4.3.1 Ergebnisse

Die folgende Tabelle zeigt die Ergebnisse der vorgestellten Messung.

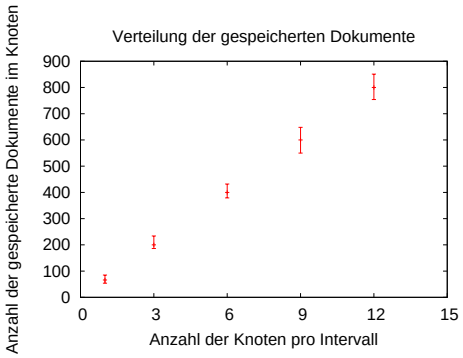


Abbildung 8.4: Messung: Variation der Knoten pro Intervall.

Die einzelnen Unterteilungen haben die gleiche Bedeutung, wie bei den letzten Messungen. Diesmal ist aber auf Abszisse die unterschiedliche Anzahl der Knoten eingetragen, die ein Intervall bearbeiten.

8.4.3.2 Bewertung

Bei unterschiedlich vielen Knoten, die ein Intervall mindestens bearbeiten, ändert sich das beschriebene Verhalten der Lastbalancierung nicht.

8.4.4 Größere Anzahl von Dokumenten

Weiterhin wurde untersucht wie das vorgestellte Distributed Hash Tables System auf größere Anzahl von Dokumenten und Knoten reagiert. Es könne aber einer Implementierung nicht so viele Knoten und Dokumente wie in der Simulation verwendet werden.

Es soll damit untersucht werden, wie sich die Verteilung der Last ändert, wenn mehr Knoten und Dokumente ins System eingefügt werden.

Die Parameter wurden nach Abschnitt 8.2 wie folgt gewählt:

- Es wurde die Anzahl der Knoten für diesen Versuch erhöht. Es befinden sich nun 100 Knoten auf den Rechner im Cluster.
- Nur einem Knoten, der ein Intervall mindestens bearbeiten muss.

8.4.4.1 Ergebnisse

Die folgende Abbildung zeigt die Ergebnisse der vorgestellten Messung.

Auf der Abszisse sind jetzt wiederum, die Anzahl der Dokumente im System eingezeichnet.

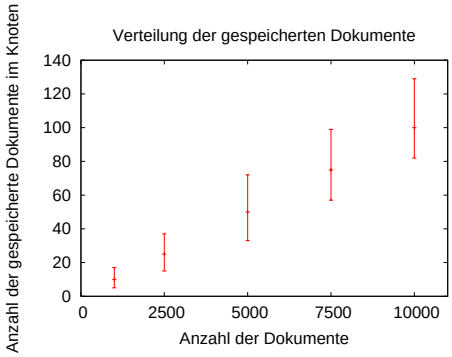


Abbildung 8.5: Messung: Größere Anzahl von Knoten und Dokumenten.

8.4.4.2 Bewertung

Auch bei Erhöhung der Anzahl der Dokumente auf 10.000 verändert sich das Ergebnis der Lastverteilung nicht zu dem aus Abschnitt 8.4.1.

8.4.5 Variation der Knoten bei gleicher Anzahl von Dokumenten

Zum Abschluss wird die Anzahl der Knoten variiert, während die Anzahl der Dokumente konstant bleibt. Es bearbeitet wieder nur mindestens ein Knoten ein Intervall.

Es soll untersucht werden, wie sich die Verteilung der Last ändert, wenn mehr Knoten zur Verteilung der Last vorhanden sind.

Die Parameter wurden nach Abschnitt 8.2 wie folgt gewählt:

- Insgesamt 3000 Dokumente, die in das System eingefügt wurden.
- Nur einem Knoten, der ein Intervall mindestens bearbeiten muss.

8.4.5.1 Ergebnisse

Die folgende Tabelle zeigt die Ergebnisse der vorgestellten Messung.

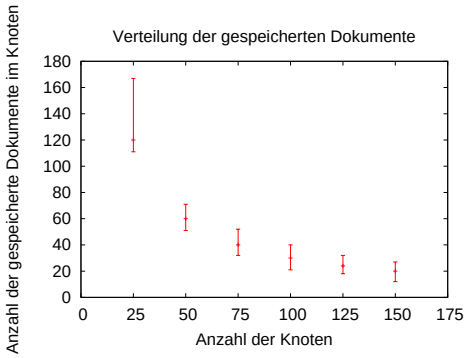


Abbildung 8.6: Messung: Variation der Knoten bei gleicher Anzahl von Dokumenten.

8.4.5.2 Bewertung

Die abschließende Messung zeigt, dass die Dokumente, je mehr Knoten im Chord System vorhanden sind ausgeglichener verteilt werden.

8.5 Fazit

Es wurde mit der Messung der Implementation die Ergebnisse anhand von verschiedenen Gesichtspunkten gezeigt und bewertet.

Mit einer Simulation konnten im vorherigen Kapitel große Knoten- und Datenmengen im Chord-Ring beobachtet werden. Dies ist mit einer realen Implementierung im Einsatz nur schwer möglich.

Es zeigt sich aber die Ähnlichkeit der Ergebnisse, mit denen der Simulation. Die Last wird in fast allen Fällen so verteilt, dass kaum Knoten viel weniger oder mehr Dokumente zu verwalten haben als andere. Durch Erhöhung der Knoten pro Intervall wird zusätzlich die Fehlertoleranz des Systems erhöht.

Die den Messungen wurde gezeigt, dass wenn der dritte Schritt des Algorithmus eingesetzt wird (Verschiebung der Intervallgrenzen mit angrenzenden Intervallen), auch bei mehreren Knoten pro Intervall eine ausgeglichene Verteilung der Last möglich ist.

Denn es ist nun einmal nicht anders, dass man, sobald man fertig ist, gleich wieder was Neues im Sinne haben müsse.

– Johann Wolfgang Goethe (1797)

Kapitel 9

Zusammenfassung

9.1 Ergebnisse

In dieser Arbeit wurde ein Verfahren zur Lastbalancierung in Peer-to-Peer Systemen vorgestellt. Dabei wurde ein Verfahren entwickelt, welches die Last innerhalb eines Distributed Hash Tables System effizienter als bestehende Lastbalancierungsalgorithmen verteilt.

Die Distributed Hash Tables, wie Chord, CAN oder DKS(N, k, f), wurden wegen ihres effizienten Verfahrens zum Auffinden von Daten verwendet. Es wurde gezeigt, dass diese Systeme deutlich Netzschonender sind, als klassische Peer-to-Peer Systeme, wie zum Beispiel Gnutella. Es wurden einige wichtige Distributed Hash Tables für Peer-to-Peer Systeme in dieser Arbeit detailliert vorgestellt und bewertet.

Dabei wird die Last beim vorgestellten Lastbalancierungsalgorithmus wie bei einer Wärmeausbreitung weitergeben. Der vorgestellt Algorithmus, benötigt dabei nur drei Regeln, um die Last innerhalb eines Distributed Hash Tables System zu verschieben. Zugleich wurde mit dem Verfahren die Fehlertoleranz des Distributed Hash Tables System erhöht, indem mehr Knoten einem Intervall zugeordnet werden.

Ebenso wurden die existierenden Lastbalancierungsalgorithmen virtuelle Server, Power of two Choices und autonome Agenten, die alle in Peer-to-Peer Systemen Last balancieren, vorgestellt. Die meisten der Algorithmen wurden hauptsächlich zur Arbeit in Distributed Hash Tables Systemen entworfen.

Anschließend wurden bereits existierende Systeme zur Lastbalancierung mit dem verwendeten Verfahren auch durch Simulation verglichen und bewertet. Dabei wurde gezeigt, dass mit dem vorgestellten Verfahren die Last besser als mit den existierenden Algorithmen auf die vorhandenen Knoten im System balanciert wurde.

Abschließend wurde das System in Java implementiert. Zur Kommunikation der Knoten untereinander und mit dem Benutzer wurde Java RMI verwendet. In Versuchen in der Praxis wurde das System beobachtet. Es zeigten sich ähnlich gute Ergebnisse wie bei der Simulation.

9.2 Ausblick

Das Verfahren zur Lastbalancierung wurde für eine große Anzahl von Knoten und Dokumenten nur simuliert.

Tests mit realen Maschinen wurden in einem Cluster und einzelnen PCs vorgenommen. Es wäre zu prüfen, wie sich das System im Einsatz mit einer größeren Anzahl von Nutzern mit verschiedensten Geräten und Netzanbindungen verhält.

Abschließend soll das System, bestehend aus Chord als Distributed Hash Table und dem vorgestellten Lastbalancierungsalgorithmus in ZIBDMS (siehe Anhang B auf Seite 101) zur Verteilung von Daten in einem Peer-Peer-System eingebaut und eingesetzt werden.

By drawing a picture of it,
he made it possible for us to see what it is....
We all understand pictures.

– Charles E. Jefferson (1925)

Anhang A

UML

A.1 Implementierung des verwendeten Lastbalancierungsalgorithmus

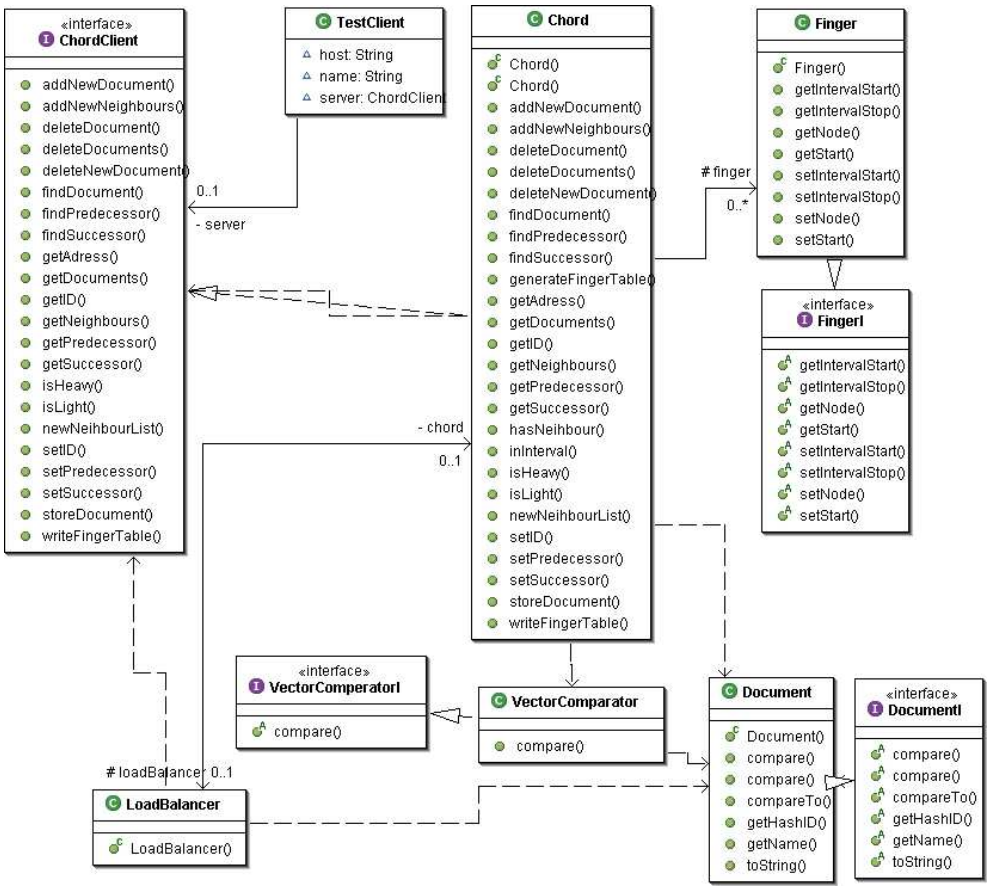


Abbildung A.1: Implementierung: Das UML-Diagramm der Implementierung zeigt die Klassen und deren Beziehungen zueinander. Bei den Klassen wurden nur öffentliche Attribute und Methoden eingetragen.

A.2 Chord Simulator für Lastbalancierungsalgorithmen

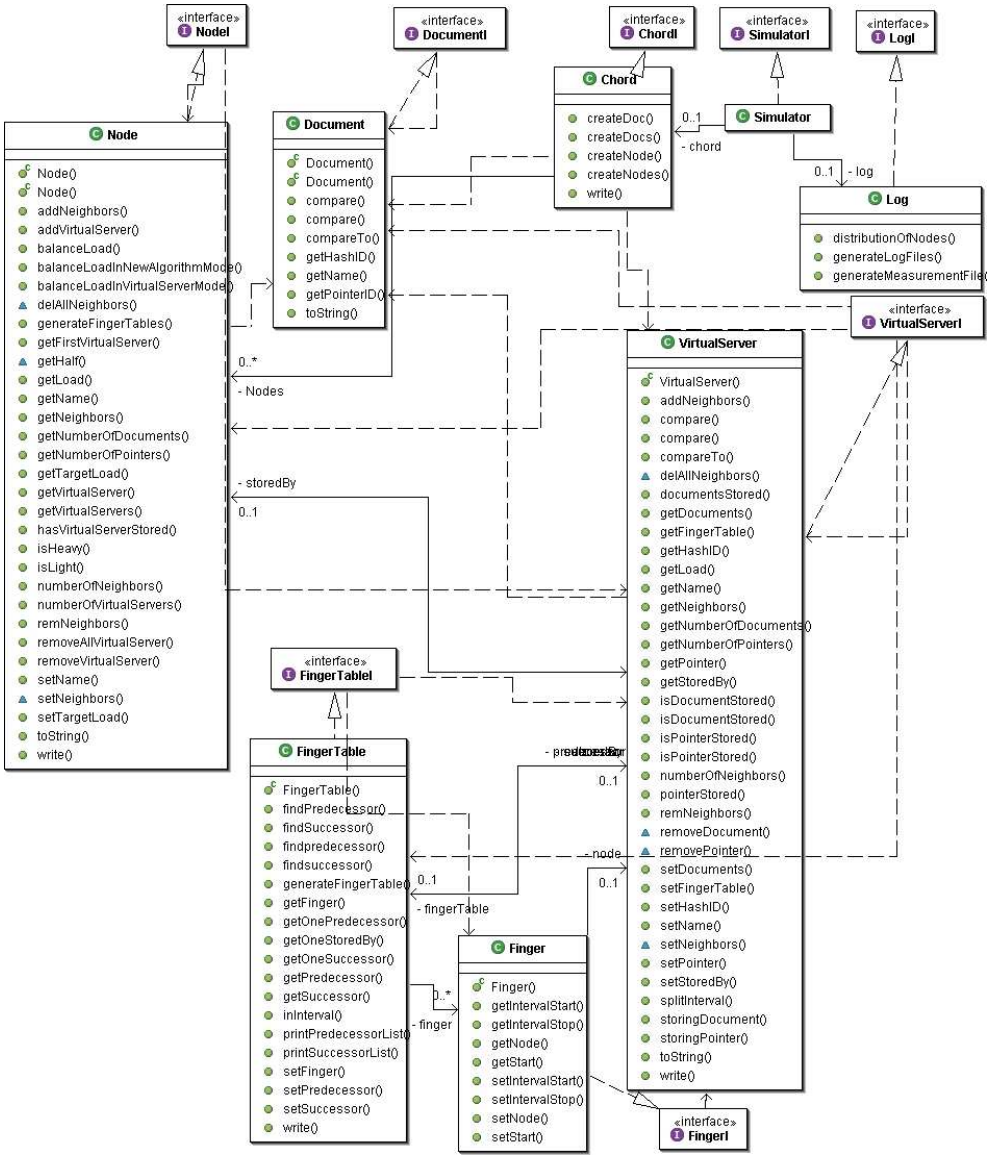


Abbildung A.2: Simulation: UML-Diagramm des Simulators, der für die verschiedenen Lastbalancierungsalgorithmen verwendet wurde. Die Beschreibung befindet sich in Abschnitt 6.1.1 auf Seite 64.

Eine Sache, welche vielen gehört,
wird schlechter verwaltet als
eine Sache, die einem
einzelnen gehört.

– Aristoteles (384 - 322 v.Chr.)

Anhang B

ZIBDMS

Der im Kapitel 5 ab Seite 51 vorgestellte Algorithmus zur Lastbalancierung in Peer-to-Peer Systemen soll in Zukunft in ZIBDMS eingebaut und verwendet werden.

ZIBDMS (ehemals CSR-DMS) bedeutet „Zuse Institute Berlin - Data Management System“ und ist ein Projekt der Computer Science Research Arbeitsgruppe im Konrad-Zuse-Zentrum. Dabei soll ein System zur Verwaltung großer Datenmengen im Grid geschaffen werden.

Es sollen von ZIBDMS mehrere Kopien von Daten selbständig angelegt und verwaltet werden. Dies soll dabei vollkommen Transparent für den Benutzer sein. Dies bedeutet, dass der Benutzer mit den Daten so arbeiten kann, als seien sie lokal auf seinem Computer vorhanden. Da die Datenmengen, zum Beispiel bei physikalischen Versuchen, aber so groß werden können, ist eine lokale Speicherung aller Daten kaum möglich. Das System soll deshalb auch mit Petabyts an Datenvolumen und Millionen von Dateien skalieren.

Zur Interaktion mit dem Benutzer gibt es in ZIBDMS eine hierarchische und eine attributbasierte [22] Sicht auf die Daten. Der so genannte Replica Location Service ist dabei für die Verwaltung der Replikate zuständig. Dafür speichert er zu jedem Dokument einen eindeutigen Namen und seine verschiedenen Speicherorte. Dieser Replica Location Service kann, damit dieser besser skaliert, zum Beispiel mit Peer-to-Peer Techniken realisiert werden.

Distributed Hash Tables kommen dafür sehr gut in Frage, da sie genau die erforderlichen Aufgaben übernehmen können. So können die einzelnen Kataloge zum Beispiel in einem Chord-Ring angeordnet werden. Dadurch ist eine ressourcenschonende Suche von Daten möglich.

Im Anschluss an diese Arbeit soll deshalb Chord als Distributed Hash Table System in ZIBDMS implementiert werden. Zur besseren Verteilung der Last, als durch Chord ohne Lastbalancierung, wird der in dieser Arbeit vorgestellt Algorithmus verwendet.

Dabei kann die Anzahl der Knoten, die ein Intervall bearbeiten sollen anhand von [51] bestimmt werden, wenn eine bestimmte Verfügbarkeit erreicht werden soll.

Die Replica Location Services, die gleiche Daten bearbeiten, sollen dabei mit geeigneten Synchronisationswerkzeugen, wie zum Beispiel nsync [54], konsistent gehalten werden.

Die folgende Abbildung B.1 zeigt die Struktur eines Replica Location Service in ZIBDMS.

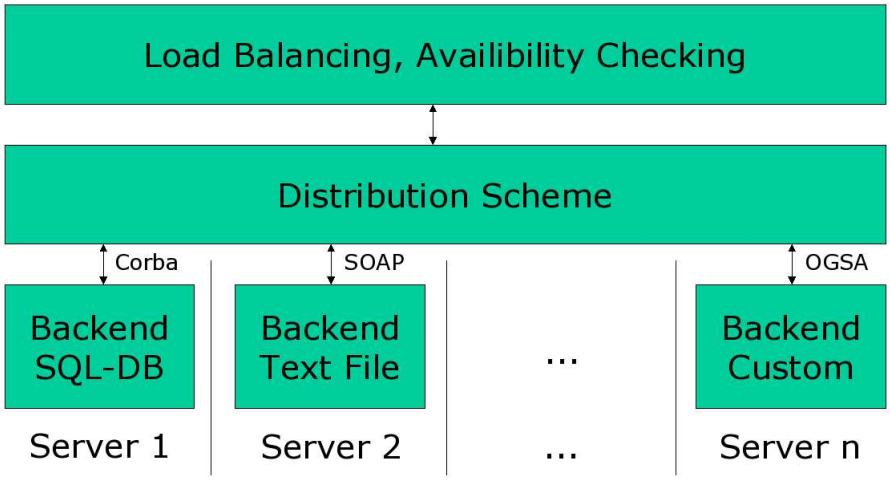


Abbildung B.1: ZIBDMS: Die Abbildung zeigt die Struktur des Replica Location Service von ZIBDMS [53].

Zum Load Balancing kommt der beschriebene Lastbalancierungsalgorithmus zum Einsatz, als Distribution Scheme unter anderem das Distributed Hash Tables System Chord.

Zur Kommunikation zwischen den verschiedenen ZIBDMS Servern wird die Common Object Request Broker Architecture (Corba) [39] eingesetzt.

Wissen gibt es in zweierlei Form:
Wir kennen den Gegenstand oder wir wissen,
wo wir Informationen über ihn erlangen.

– Samuel Johnson (1709-1784)

Anhang C

Literaturverzeichnis

[1] E. Adar und B. A. Huberman.
Free Riding on Gnutella.
Xerox Palo Alto Research Center, August 2000.
`http://www.hpl.hp.com/shl/papers/gnutella/`.

[2] L. O. Alima, S. El-Ansary, P. Brand und S. Haridi.
DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications.
In *Proceedings of The 3rd International Workshop on Global and P2P Computing on Large Scale Distributed Systems (CCGRID 2003)*, Tokyo, Japan, Mai 2003.

[3] A. Andrzejak.
Overview: Challenges in Decentralized and P2P Systems.
Folien, Seminar Information Management in the Web, April 2003.
`http://www.inf.fu-berlin.de/lehre/SS03/seminarDBMobile/`.

[4] P. Arabshahi.
Swarm Intelligence.
Jet Propulsion Laboratory, NASA, Pasadena, USA.
`http://dsp.jpl.nasa.gov/members/payman/swarm/`.

[5] Y. Azar, A. Z. Broder, A. R. Karlin und E. Upfal.
Balanced Allocations.
SIAM Journal on Computing, 29(1):180–200, 2000.

[6] O. Babaoglu, H. Meling und A. Montresor.
Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems.
In *Proceedings of the 22th International Conference on Distributed Computing Systems (ICDCS '02)*, Wien, Österreich, Juli 2002.

[7] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris und I. Stoica.
Looking Up Data in P2P Systems.
Communications of the ACM, 46(2):43–48, 2003.

[8] Bibliographisches Institut & F. A. Brockhaus AG.
Brockhaus in einem Band.
http://www.brockhaus.de/nachschlagen/.

[9] J. Byers, J. Considine und M. Mitzenmacher.
Simple Load Balancing for Distributed Hash Tables.
In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, USA, Februar 2003.

[10] D. D. Clark.
The Design Philosophy of the DARPA Internet Protocols.
In *Proceedings of SIGCOMM*, Seiten 106–114, Stanford, CA, 1988. ACM.

[11] Clip2 Distributed Search Services.
The Gnutella Protocol Specification v0.4.
http://www.stanford.edu/class/cs244b/.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest und C. Stein.
Introduction to Algorithms (2nd Edition).
The MIT press, 2001.

[13] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica und H. Balakrishnan.
Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service.
In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Deutschland, 2001. IEEE Computer Society.

[14] N. K. Daswani, H. Garcia-Molina und B. Yang.
Open Problems in Data-Sharing Peer-to-Peer Systems.
In *Proceedings of the 9th International Conference on Database Theory*, Seina, Italien, Januar 2003.

[15] N. K. Daswani, H. Garcia-Molina und B. Yang.
Open Problems in Data-Sharing Peer-to-Peer Systems.
Folien, 2003.
http://www-db.stanford.edu/peers/.

[16] N. De Bruijn.
A combinatorial problem.
Nederlandse Academie van Wetenschappen, Proc. A 49:758–764, 1946.

[17] S. El-Ansary, L. O. Alima, P. Brand und S. Haridi.
A Framework for Peer-To-Peer Lookup Services based on k-ary search.
Technical Report TR-2002-06, Swedish Institute of Computer Science, 2002.

[18] P. Fraigniaud und P. Gauron.
The Content-Addressable Network D2B.
Technical Report TR-LRI-1349, Laboratoire de Recherche en Informatique, Univ. Paris-Sud, Frankreich, Januar 2003.

[19] C. Gavoille.
Routing in Distributed Networks: Overview and Open Problems.
SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory), 32, 2001.

[20] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker und I. Stoica.
Complex Queries in DHT-based Peer-to-Peer Networks.
In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, USA, März 2002.

[21] G. Himmelein.
RIAA-Strategie scheint aufzugehen.
heise online Newsticker, August 2003.
`http://www.heise.de/newsticker/data/ghi-23.08.03-003/`.

[22] F. Hupfeld.
Hierarchical Structures in Attribute-based Namespaces and their Application to Browsing.
Technical Report ZR-03-06, Zuse Institute Berlin, März 2003.

[23] M. F. Kaashoek und D. R. Karger.
Koorde: A simple degree-optimal distributed hash table.
In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, USA, Februar 2003.

[24] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin und R. Panigrahy.
Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.
In *Proceedings of the ACM Symposium on Theory of Computing*, Seiten 654–663, 1997.

[25] D. Karger und M. Ruhl.
Finding Nearest Neighbors in Growth-restricted Metrics.
In *Proceedings of the 34th ACM Symposium on Theory of Computing*, Seiten 741–750, Montreal, Quebec, Kanada, 2002. ACM Press.

[26] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins und Y. Yerushalmi.
Web Caching with Consistent Hashing.
In *Proceedings of the 8th International WWW Conference*, Toronto, Canada, 1999.

[27] J. Kubiatowicz.
Extracting Guarantees from Chaos.
Communications of the ACM, 46(2):33–38, 2003.

[28] D. Malkhi, M. Naor und D. Ratajczak.
Viceroy: A Scalable and Dynamic Emulation of the Butterfly.
In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, August 2002.

[29] P. Maymounkov und D. Mazieres.
Kademlia: A Peer-to-peer Information System Based on the XOR Metric.
In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, USA, März 2002.

[30] P. Maymounkov und D. Mazieres.
Kademlia: A Peer-to-peer Information System Based on the XOR Metric.
Folien, New York University, USA, 2002.
`http://www.sics.se/~sameh/research/P2P/Kademlia/`.

[31] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins und Z. Xu.
Peer-to-Peer Computing.
Technical Report HPL-2002-57, HP Laboratories, Palo Alto, März 2002.

[32] M. Mitzenmacher, A. W. Richa und R. Sitaraman.
The Power of Two Random Choices: A Survey of Techniques and Results.
In Handbook of Randomized Computing, P. Pardalos, S. Rajasekaran, und J. Rolim, Eds. Kluwer, 2000.
`http://citeseer.nj.nec.com/mitzenmacher00power.html`.

[33] A. Montresor, H. Meling und O. Babaoglu.
Messor: Load-Balancing through a Swarm of Autonomous Agents.
In *Proceedings of 1st Workshop on Agent and Peer-to-Peer Systems*, Bologna, Italien, Juli 2002.

[34] A. Montresor, H. Meling und O. Babaoglu.
Messor: Load-Balancing through a Swarm of Autonomous Agents.
Technical Report UBLCS-2002-11, Dept. of Computer Science, University of Bologna, Italien, 2002.

[35] R. Morris, I. Stoica, D. Karger, M. F. Kaashoek und H. Balakrishnan.
Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.
Folien, SIGCOMM 2001.
`http://www.pdos.lcs.mit.edu/~rtm/slides/sigcomm01.ppt`.

[36] Myricom, Inc..
Myrinet - Open Specifications and Documentation.
`http://www.myri.com/open-specs/`.

[37] M. Naor und U. Wieder.
A Simple Fault Tolerant Distributed Hash Table.
In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, USA, Februar 2003.

[38] Napster.
`http://www.napster.com`.

[39] Object Management Group, Inc..
CORBA - the Common Object Request Broker Architecture.
`http://www.omg.org/gettingstarted/corbafaq.htm`.

[40] J. Olsson.
FastTrack and KaZaA: The number of users decreases.
AxisNova.com, August 2003.
`http://www.axisnova.com/articles/article_109.shtml`.

[41] C. G. Plaxton und R. Rajaraman.
Fast Fault-Tolerant Concurrent Access to Shared Objects.
In *Proceedings of IEEE Symposium on Foundations of Computer Science*, Seiten 570–579, 1996.

[42] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp und I. Stoica.
Load Balancing in Structured P2P Systems.
In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, USA, Februar 2003.

[43] S. Ratnasamy.
A Scalable Content-Addressable Network.
Doktorarbeit, University of California at Berkeley, 2002.

[44] S. Ratnasamy, P. Francis, M. Handley, R. Karp und S. Shenker.
A Scalable Content-Addressable Network.
In *Proceedings of ACM SIGCOMM, San Diego, CA*, 2001.

[45] S. Ratnasamy, S. Shenker und I. Stoica.
Routing Algorithms for DHTs: Some Open Questions.
In *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, USA, 2002.

[46] M. Ripeanu.
Peer-to-Peer Architecture Case Study: Gnutella Network.
Technical Report TR-2001-26, University of Chicago, Juli 2001.

[47] J. Ritter.
Why Gnutella Can't Scale. No, Really., Februar 2001.
`http://www.tch.org/gnutella.html`.

[48] T. G. Robertazzi.
Ten Reasons to Use Divisible Load Theory.
IEEE Computer Society: Computer magazine, 36(5):63–68, Mai 2003.

[49] S. Saroiu, K. P. Gummadi und S. D. Gribble.
Measuring and analyzing the characteristics of Napster and Gnutella hosts.
Multimedia Systems, 9(2):170–184, 2003.
Springer-Verlag.

[50] S. Saroiu, P. K. Gummadi und S. D. Gribble.
A Measurement Study of Peer-to-Peer File Sharing Systems.
Technical Report UW-CSE-01-06-02, Department of Computer Science & Engineering, University of Washington, USA, Juli 2001.

[51] F. Schintke und A. Reinefeld.
On the Cost of Reliability in Large Data Grids.
Technical Report ZR-02-52, Zuse Institute Berlin, 2002.

[52] F. Schintke, T. Schütt und A. Reinefeld.
A Framework for Self-Optimizing Grids Using P2P Components.
In *Proceedings of 14th Intl. Workshop on Database and Expert Systems Applications (DEXA'03)*, Seiten 689–693. IEEE Computer Society, September 2003.

[53] T. Schütt und F. Schintke.
A New Data Management System for the Grid (CSR-DMS).
Folien, Zuse Institute Berlin, 2002.

[54] T. Schütt, F. Schintke und A. Reinefeld.
Efficient Synchronization of Replicated Data in Distributed Systems.
In *Proceedings of International Conference on Computational Science 2003 (ICCS 2003)*, Volume 2657 von *Lecture Notes in Computer Science*, Seiten 271–283, St. Petersburg, Russland, Juni 2003. Springer.

[55] Sharman Networks Ltd.
200m - Hooray!, März 2003.
`http://www.kazaa.com/us/news/201.htm`.

[56] C. Shirky.
What is P2P... and what isn't?
The O'Reilly Network, November 2000.
`http://www.openp2p.com/lpt/a/p2p/2000/11/24/shirky1-whatisp2p.html`.

[57] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek und H. Balakrishnan.
Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.
In *Proceedings of ACM SIGCOMM, San Diego, CA*, 2001.

[58] Sun Microsystems, Inc..
Java Remote Method Invocation (RMI).
`http://java.sun.com/products/jdk/rmi/`.

[59] Sun Microsystems, Inc..
The Source for Java Technology.
`http://java.sun.com`.

[60] Sun Microsystems, Inc..
Java RemoteMethodInvocation Specification, 2002.
`ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf`.
Revision 1.8, Java 2 SDK, Standard Edition, v1.4.

[61] The Anthill Project.
`http://www.cs.unibo.it/projects/anthill/`.

[62] The Chord Project.
`http://www.pdos.lcs.mit.edu/chord/`.

[63] R. Tolksdorf.
Seminar Swarm Intelligence.
Institut für Informatik, Freie Universität Berlin, 2003.
`http://www.inf.fu-berlin.de/inst/ag-nbi/lehre/03/S_SI/`.

[64] M. Zhao.
Peer-to-Peer Online Resource.
Dartmouth College Hanover,USA, April 2003.
http://www.cs.dartmouth.edu/~zhaom/research/marianas/resource.html.

[65] V. Zota.
P2P: eMule will auf Server verzichten.
heise online Newsticker, September 2003.
http://www.heise.de/newsticker/data/vza-18.09.03-000/.

[66] Zuse Institute Berlin.
ElFie @ ZIB.
http://elfie.bcbio.de.