Diplomarbeit

Ein Peer-to-Peer System mit Bereichsabfragen in PlanetLab

Eingereicht:
Christian v. Prollius
19. Februar 2008

Betreuer:
Prof. Dr. Jochen Schiller
Prof. Dr. Alexander Reinefeld

Freie Universität zu Berlin
Institut für Informatik

Copyright (c) 2008 Christian v. Prollius.

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Berlin, 19. Februar 2008 _____
Unterschrift

Danksagung

Mein Dank gilt Univ.-Prof. Dr.-Ing. Jochen Schiller, Vizepräsident der Freien Universität Berlin und Prof. Dr. Alexander Reinefeld, Leiter des Bereichs Computer Science am Konrad-Zuse-Zentrum für Informationstechnik Berlin. Besonders bedanken möchte ich mich bei Thorsten Schütt für die kompetente Betreuung und die vielen Anregungen. Weiterhin bedanke ich mich bei meinen Eltern, die mir mit ihrer großartigen Unterstützung das Studium ermöglicht haben und bei meinem Kommilitonen und guten Freund Tinosch Ganjineh, mit dem ich einen Großteil des Studiums gemeinsam absolviert habe. Zuletzt möchte ich mich bei meinen Freunden und Verwandten bedanken, die mich stets motiviert und aufgebaut haben.

Kurzfassung

Aktuelle Entwicklungen zeigen, dass Peer-to-Peer (P2P) Anwendungen wie Skype oder Bittorrent [1] im Internet immer mehr an Bedeutung gewinnen. In den letzten Jahren hat es einen explosionsartigen Anstieg an Nutzern und Daten in solchen Netzen gegeben. Dabei stellt der eigentliche Dateitransfer zwischen zwei Rechnern kein großes Problem mehr dar und auch der Speicherbedarf für die große Menge an Daten kann durch die Weiterentwicklung der Hardware gut gedeckt werden.

Das eigentliche Problem liegt vielmehr darin, den Rechner zu finden, der die gewünschten Daten hat. Client-Server Architekturen, wie zum Beispiel Napster[1], haben sich als ungünstig herausgestellt. Wenige Server, die eine große Anzahl an Clients bedienen müssen, sind einerseits sehr anfällig gegenüber Angriffen und Ausfällen (*Single Point of Failure*) und kommen auch nicht mit der ständig wachsenden Anzahl an Nutzern zurecht. Verteilte Hashtabellen (DHT) bieten hier einen guten Lösungsansatz, der mit einer großen Anzahl an Nutzern skaliert und ausfallsicher ist.

Andere dezentrale Lösungen, wie zum Beispiel das P2P Netzwerk Gnutella[2] haben zwar das Problem des *Single Point of Failure* gelöst, jedoch haben sie starke Nachteile bei der Suche nach Keys. Bei einer Suche wird ein *Broadcast* verwendet (jeder schickt die Anfrage an jeden weiter) und damit ein enormer Netzwerkverkehr erzeugt. In "Why Gnutella Can't Scale. No, Really" [11] wird erklärt, dass eine Suchanfrage bei Standardeinstellungen in der Clientsoftware einen Netzwerkverkehr von 17MB erzeugt. Deswegen wird zusätzlich eine Lösung benötigt, die Keys und Values geordnet verteilt, damit sie gezielt gesucht werden können.

Aus diesem Grund beschäftigt sich die folgende Arbeit mit einer völlig dezentralen Architektur, die außerdem eine sinnvolle Platzierung der Keys vornimmt. Die dezentrale Architektur hat den Vorteil, dass die Endgeräte den Hauptteil des Dienstes selbst erbringen und damit jeder zusätzliche Teilnehmer seine eigenen Ressourcen beisteuert.

Diese Arbeit präsentiert *Chord#*, eine dezentrale, skalierbare und selbstorganisierende verteilte Hashtabelle. *Chord#* wurde ausgewählt, da in dieser Arbeit auch Wert auf Bereichsabfragen gelegt wurde. Diese sind zum Beispiel bei dem *Chord* Algorithmus nicht möglich, da dieser eine Hashfunktion für die Keys verwendet und somit die Daten zwar gleichmäßig aber unsortiert auf die Teilnehmer verteilt.

Es wird in dieser Arbeit gezeigt, dass mit Hilfe von *Chord#* auch ohne die Hashfunktion gute Ergebnisse erzielt werden. Außerdem können durch den Verzicht auf die Hashfunktion Bereichsabfragen ermöglicht werden. Dafür wird der *Chord#* Algorithmus in Java implementiert (ca. 1500 Zeilen Code) und in dem Forschungsnetz PlanetLab[3] ausführlich auf Laufzeiten, Instandhaltungskosten und Skalierung getestet.

[1] http://www.napster.com, 2007
[2] http://www.gnutella.com, 2007
[3] http://www.planet-lab.org, 2007

Abstract

Recent developments show that peer-to-peer (p2p) applications, such as Skype or Bittorrent [1] have become increasingly important in the internet. Over the last years there has been a rapid growth of both users and data in such networks. However, the actual file transfer between two peers is not really an issue anymore. The same holds true for data storage, since the new hardware grants users enough space to store their data.

The real problem is finding the peers that possess the desired data. Client-server architectures like Napster[4] have proven to be ineffective addressing that problem. One or few servers being responsible for many peers are vulnerable to attacks or failures (*single point of failure*). Additionally, they are unable to cope with the rapidly growing number of peers. Distributed hashtables (DHT) are a good approach to solve these problems, since they scale nicely with large numbers of peers and provide a high tolerance for errors.

Other decentralized solutions like the p2p network Gnutella[5] solved the problem of *Single Point of Failure* but show considerable disadvantages when searching for keys. The peers in Gnutella use a broadcast (sending the message to all peers they know) resulting in massive traffic. According to "Why Gnutella Can't Scale. No, Really." [11], each search using standard client settings yields 17MB traffic. This calls for a different solution, distributing keys and values to peers quickly and efficiently so they can be found fast.

For that reason this thesis focuses on a fully distributed architecture using organized key placement. One major advantage of distributed architecture is the fact, that the peers do most of the work themselves. This way, new peers joining the network add resources to it.

This thesis presents *Chord#*, a scalable, self-organizing and completely decentralized DHT. It has been chosen due to its capability to allow range queries. The regular *Chord* algorithm does not support range queries, because of the hashfunction it uses to evenly distribute the keys among the peers. This results in similar or logical coherent keys most likely not being close together in the network.

This thesis shows *Chord#* achieving same results as *Chord* – regarding performance and maintenance costs – *without* the hashfunction. In dropping the hashfunction this algorithm allows the use of range queries. The *Chord#* algorithm is implemented in Java (about 1500 lines of code) and thoroughly tested in the research network PlanetLab[6]. The results are evaluated regarding performance, maintenance and scalability.

[4] http://www.napster.com, 2007
[5] http://www.gnutella.com, 2007
[6] http://www.planet-lab.org, 2007

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

> Ernst zu nehmende Forschung erkennt man daran, dass plötzlich zwei Probleme existieren, wo es vorher nur eines gegeben hat.
>
> *(Thorstein Bunde Veblen)*

1.1 Thema dieser Arbeit

Chord ist ein Protokoll für eine verteilte Hashtabelle. Es wird dazu verwendet, zu einem gegebenen Key den zuständigen Knoten im Netz zu ermitteln. Die Laufzeit für eine Suche liegt bei n Knoten mit hoher Wahrscheinlichkeit in $O(log(n))$.

Chord basiert auf einer Variante des Consistent Hashing. Es sorgt dafür, dass jeder Knoten in etwa für die gleiche Anzahl an Keys zuständig ist. Durch die Hashfunktion befinden sich jedoch inhaltlich ähnliche Informationen mit hoher Wahrscheinlichkeit auf Knoten, die topologisch weit voneinander entfernt sind.

Es wurde mit Hilfe von Simulationen gezeigt [15], dass mit *Chord#* auf die Hashfunktion verzichtet werden kann, ohne den Vorteil der logarithmischen Laufzeit für die Suche zu verlieren. Vorteile von *Chord#* sind

- bessere Laufzeit bei der Aktualisierung der Fingertabelle
- Laufzeit liegt auch im schlimmsten Fall in $O(log(n))$ (nicht nur mit hoher Wahrscheinlichkeit)
- komplexe Queries (z.B. Bereichsabfragen) werden unterstützt

Die theoretischen Laufzeitanalysen werden mit Messungen an einem realen System bestätigt. Dafür wird der *Chord#* Algorithmus implementiert und auf der PlanetLab Infrastruktur, einem weltweit verteilten Netz von Rechnern, getestet. Hierbei wurde auf folgende Punkte Wert gelegt:

Verifizierung der Theorie
: Die theoretischen Analysen[6] für Bandbreite, Latenz und Instandhaltungskosten sollen verifiziert werden. Ebenso soll die Effizienz der Suche – im Hinblick auf die Anzahl der Hops und der Antwortzeiten – bewertet werden.

Parameterbestimmung
: Es sollen geeignete Parameter gefunden werden für die Anzahl der Nachfolger, die Größe des Aktualisierungsintervalls für die Nachfolger- und Fingertabellen, sowie für die Anzahl der Replikate und die Größe der Zeitintervalle für die Lastverteilung.

Für die Analysen werden Statistiken gesammelt und diese graphisch ausgewertet. Für die Bestimmung der Parameter [5] werden Testreihen durchgeführt und gute Werte gefunden. In den meisten Fällen muss abgewogen werden zwischen weniger Aufwand (Netzlast, Prozessorlast etc) und besser Laufzeit oder Robustheit [7].

1.2 Struktur dieser Arbeit

Zuerst wird in Kapitel 2 das *Chord* Protokoll vorgestellt, welches dem *Chord#* als Grundlage dient. Dabei werden kurz die verwendeten Prozeduren beschrieben, die durchschnittlichen Laufzeiten erklärt und heutige Anwendungsgebiete für das Protokoll genannt.

Kapitel 3 beschreibt verwandte Protokolle und Anwendungen. Dort werden die Protokolle Kademlia, CAN und Tapestry erwähnt und Gemeinsamkeiten bzw. Abweichungen zum *Chord* Protokoll erörtert.

In Kapitel 4 wird ausführlich auf *Chord#*, die Erweiterung des *Chord* Protokolls eingegangen. Die abweichenden Prozeduren werden erklärt und deren Laufzeiten analysiert. Dabei wird gezeigt, dass die Vorteile von *Chord* erhalten bleiben, aber neue Möglichkeiten entstehen, die mit dem einfachen *Chord* Protokoll nicht realisierbar sind.

In Kapitel 5 wird die Testumgebung PlanetLab beschrieben und dabei näher auf die einzelnen Mechanismen eingegangen.

Schließlich werden in Kapitel 6 die für die Testläufe wichtige Parameter genannt und Ergebnisse anhand von Statistiken und Grafiken ausgewertet. Es wird geprüft, ob die erwarteten Werte für Latenzen und Datenmengen (für Instandhaltung und Suche) in einem realen Netz verifiziert werden können beziehungsweise Gründe für Abweichungen erörtert.

2 *Chord*

2.1 Struktur und Prozeduren

Chord [17] ist ein Protokoll für eine verteilte Hashtabelle für P2P-Netzwerke. Es löst das Problem des effizienten Auffindens eines Knotens, der bestimmte Daten hat. Da P2P-Systeme weder zentral gesteuert werden, noch hierarchisch aufgebaut sind, sondern auf jedem Knoten die gleiche Software läuft, wird eine effiziente Abbildung von Keys auf Knoten benötigt. *Chord* findet solche Knoten auch in einem sich ständig verändernden Netz, in dem neue Knoten hinzukommen und alte Knoten das Netz verlassen oder ausfallen. Jeder Knoten hat eine eindeutige Kennung, die *Node-ID*. Sowohl die Keys als auch die *Node-IDs* sind aus dem Keyspace, der lexikographisch sortierbar ist. Durch diese *Node-ID* wird sowohl die Position des Knotens bestimmt, wie auch der Bereich an Keys, für den er zuständig ist.

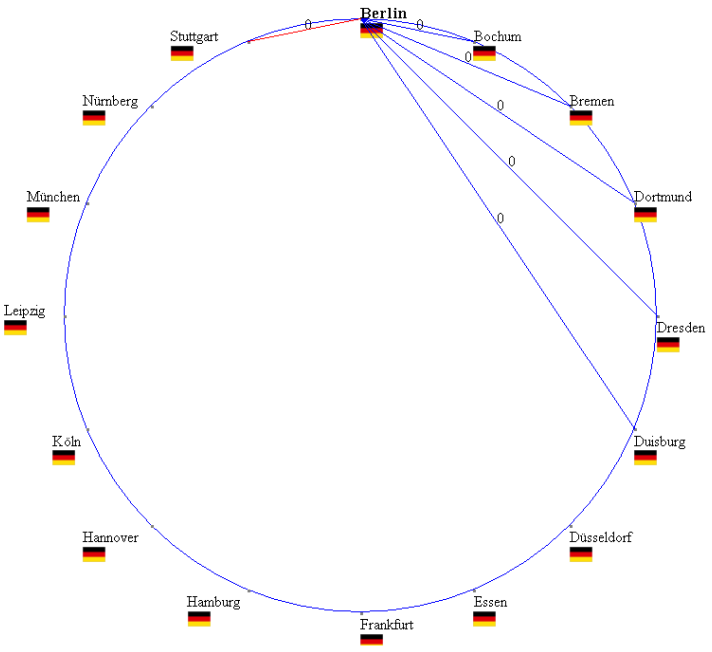

Abbildung 2.1: Chord: Ringstruktur mit Nachfolger und Vorgänger

Abbildung 2.1 zeigt die typische Ringstruktur, die von *Chord* aufgebaut wird. Zur Vereinfachung werden die Knoten hier nach deutschen Städten benannt. Diese *Node-IDs*

sind aber nicht frei wählbar, weil sie den Key-Bereich bestimmen, für den ein Knoten zuständig ist. In diesem (ideal stabilisierten) Ring hat jeder Knoten einen eindeutigen Nachfolger und Vorgänger. Der Knoten *Berlin* hat beispielsweise den Nachfolger *Bochum* und den Vorgänger *Stuttgart*. Für den Fall, dass ein Knoten ausfällt, werden mehrere Nachfolger gespeichert (in Abbildung 2.1 sind dies fünf, gekennzeichnet durch die blauen Linien), damit die Knoten die Verbindung zum Ring nicht verlieren.

Für die Suche benötigen die Knoten außerdem eine Fingertabelle. Dies ist eine möglichst kleine Liste von Knoten, die so ausgewählt sind, dass ein effizientes Routing möglich ist.

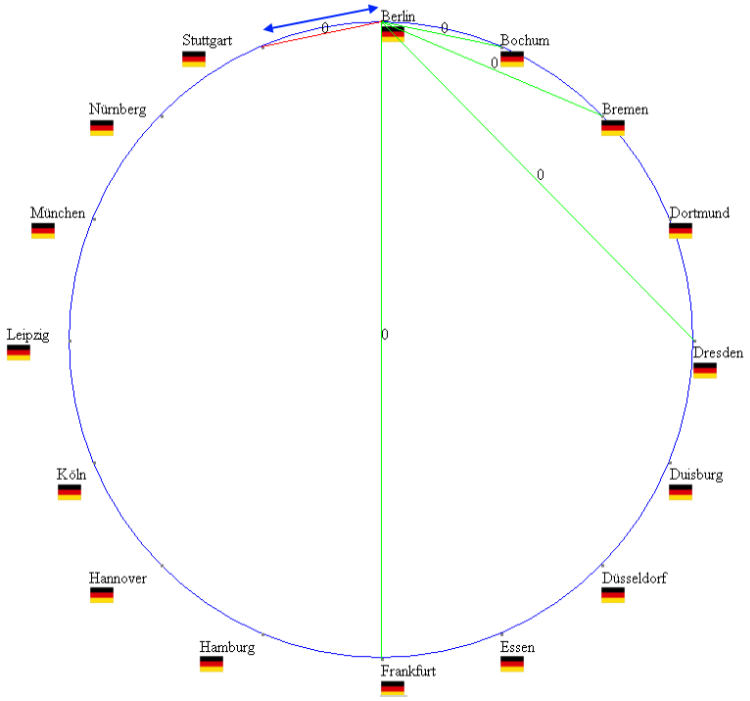

Abbildung 2.2: Chord: Ringstruktur mit Finger und Zuständigkeitsbereich

Abbildung 2.2 zeigt die gleiche Ringstruktur, nur dass diesmal die Finger des Knoten *Berlin* hervorgehoben sind. Diese ermöglichen das effiziente Auffinden von Knoten, da der Abstand zwischen zwei Fingern im Ring exponentiell zunimmt. Zum Beispiel hat der Knoten *Frankfurt* eine Entfernung von acht und ist damit doppelt so weit entfernt wie der vorherige Finger *Dresden*, der vier Knoten entfernt liegt. Die Fingertabelle wird solange vergrößert, bis der Ring einmal umrundet wurde. Würde also ein Finger den Knoten selbst überspringen, so wird er nicht mehr aufgenommen. Der letzte Finger liegt also mindestens soweit entfernt, dass die Entfernung der Hälfte des Rings entspricht. Damit kann bei einer Suche in jedem Schritt mindestens die Hälfte des Weges zum gesuchten Knoten zurückgelegt werden.

2.1.1 Overlay-Netzwerk und Underlay-Netzwerk

Das Routing in *Chord#* bezieht sich auf das Overlay-Netzwerk, also die von *Chord#* erzeugte Ringtopologie. Dabei wird nicht die tatsächliche Entfernung der Knoten voneinander in dem Underlay-Netzwerk berücksichtigt. Ein Hop im Ring kann unter Umständen sehr viele Hops im Internet bedeuten. Dies wird mit dem Begriff *Overlaystretch* verdeutlicht.

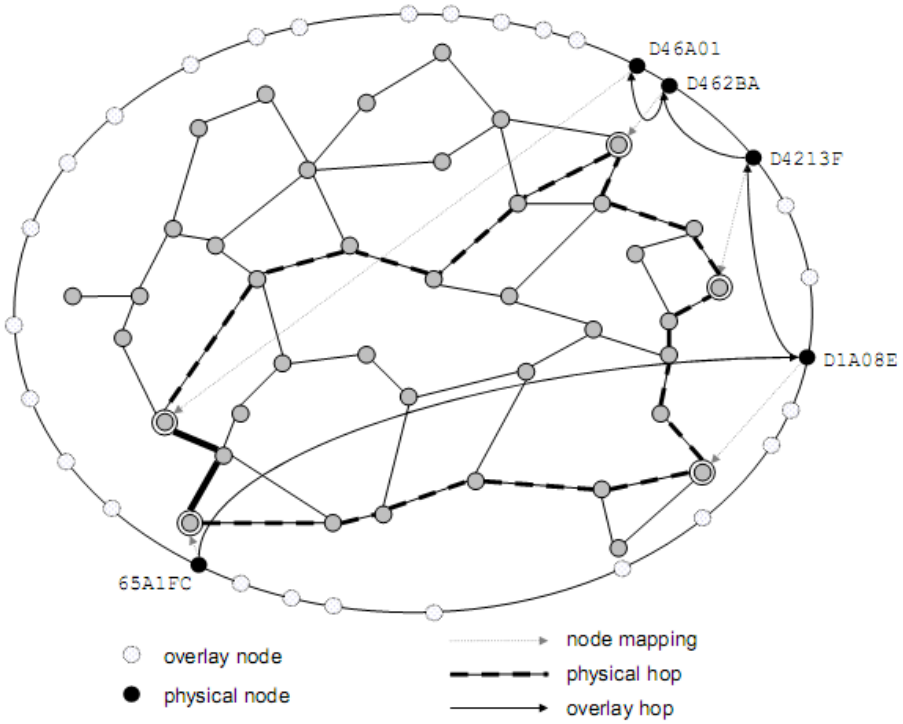

Abbildung 2.3: Chord#: Overlaystretch

Abbildung aus "Structured Peer-to-Peer Services for Mobile Ad Hoc Networks" [19]

In Abbildung 2.3 wird der Overlaystretch veranschaulicht. Das Overlay-Netzwerk ist hier ein Ring (weiße Kreise), das Underlay-Netzwerk ist durch dunkle Kreise gekennzeichnet. Die grauen Pfeile beschreiben die Zuordnung von Knoten im Ring zu ihrer tatsächlichen Position im Netzwerk. Der Knoten *65A1FC* wird nun auf kürzestem Weg im Overlay-Netzwerk zu seinem Ziel *D46A01* geroutet. Dies entspricht im Overlay-Netzwerk 4 Hops (*D1A08E*, *D4213F*, *D462BA* und *D46A01*), im Underlay-Netzwerk aber 17 Hops. Da hier nun aber beide Knoten zufällig im Underlay-Netzwerk dicht beieinander liegen, hätte *65A1FC* sein Ziel mit 2 Hops erreichen können. Der Overlaystretch ist also $\frac{17}{2}$.

Ein Lösungsansatz für dieses Problem ist das *Proximity Routing*. Hier werden die Finger geschickt ersetzt, sodass die Laufzeit für die Suche nicht verschlechtert wird, aber die Beschaffenheit des Underlay-Netzwerks berücksichtigt wird. Dafür wird die Nachfolger-

Liste von jedem Finger geprüft, ob einer der Knoten besser geeignet ist als der Finger selbst. Das Kriterium ist hierbei variabel, zum Beispiel die Antwortzeit in Millisekunden, die Verfügbarkeit oder die Kapazität. Die Nachfolger liegen ungefähr an derselben Stelle im Ring, wodurch das Routing nicht wesentlich beeinflusst wird. Die Wahrscheinlichkeit ist jedoch groß, dass einer der 5 Nachfolger auf das Underlay-Netzwerk bezogen wesentlich näher liegt und damit das Routing beschleunigt wird.

2.2 Verwendung von *Chord*

Da *Chord* nur ein Protokollsystem ist, werden hier ein paar konkrete Implementierungen erläutert.

2.2.1 Cooperative File System

Das Cooperative File System (CFS) [2] ist ein neues P2P read-only storage system, welches Garantien über Effizienz, Robustheit, Lastverteilung und Verfügbarkeit von Daten liefert. Dafür wird die dezentrale Architektur von *Chord* verwendet. Die read-only Eigenschaft bezieht sich hier nur auf die Clients. Ein File-System kann durch seinen Publisher über gesicherte Mechanismen verändert werden.

Die Blöcke der Hashtabelle werden als Dateisystem interpretiert und fein granular gespeichert, um die Last gleichmäßig zu verteilen. Für die Robustheit wird Replikation verwendet und um dabei die Latenz gering zu halten, wird von den Replikaten der am besten geeignete Server ausgewählt. Durch die Verwendung von *Chord* werden die Blöcke in logarithmischer Zeit in Abhängigkeit zur Anzahl der Server gefunden.

CFS läuft unter Linux, OpenBSD und FreeBSD. Tests haben gezeigt, dass CFS seinen Clients Daten ebenso schnell senden kann wie FTP. Auch die Skalierbarkeit wurde gezeigt: Bei einem Netz aus 4096 Servern hat eine durchschnittliche Suche nur 7 Server kontaktiert. Dieses entspricht genau dem erwarteten Wert, da $\log_2(4096) = 14$, also jeder Knoten in 14 Hops erreicht werden kann, wenn die Finger richtig gesetzt sind. Die erwartete durchschnittliche Suche hat halb so viele Hops. Auch die Robustheit des Systems konnte gezeigt werden, da selbst bei einem Ausfall der Server die Leistung nicht beeinträchtigt wurde.

Layer	Responsibility
FS	Interprets blocks as files; presents a file system interface to applications.
DHash	Stores unstructured data blocks reliably.
Chord	Maintains routing tables used to find blocks.

Abbildung 2.4: Cooperative File System: Aufbau [2]

Das CFS besteht aus drei logischen Schichten, der FS-Schicht, der DHash-Schicht und der *Chord*-Schicht. Die FS-Schicht interpretiert die Blöcke als Dateien und stellt der Anwendungsschicht ein File System Interface zur Verfügung. Für den Anwender ist die verteilte Struktur transparent. Die darunter liegende DHash-Schicht ist dafür zuständig, die Blöcke zur Verfügung zu stellen. Diese werden auf den Servern verteilt und Replikate

angelegt, um das System robust gegenüber Ausfällen zu machen. DHash benutzt wiederum die *Chord*-Schicht, um den für einen Block zuständigen Server zu finden. Hier kommt die gute Skalierung von *Chord* zum Tragen, da auf jedem Server nur logarithmisch viele Informationen gespeichert werden müssen, um ein effizientes Routing zu ermöglichen.

2.2.2 UsenetDHT

Das Usenet ist ein weltweites Netzwerk, das so genannte Newsgroups zur Verfügung stellt. Teilnehmer aus aller Welt können dort mit entsprechender Client Software (Newsreadern) Artikel lesen und schreiben. Das Usenet wächst stark an, täglich werden etwa 3 TB an Artikeln erstellt. Einzelne Server können solche Mengen nur schwer bewältigen.

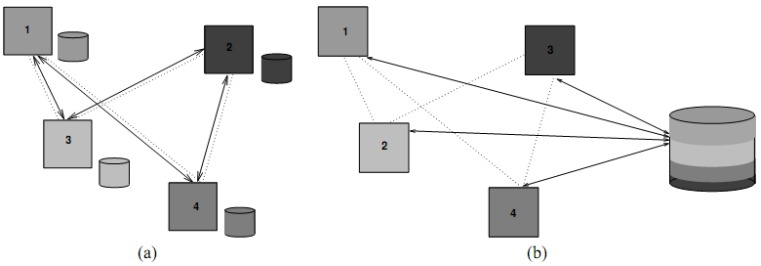

Abbildung 2.5: Usenet DHT: Vergleich zu Usenet [16]

Vergleich zwischen a) dem normalen Usenet und b) Usenet DHT. Die Server tauschen sowohl Artikeldaten (durchgezogene Linien), sowie Metadaten über die Artikel (gestrichelte Linien) aus. Die Server in Usenet DHT speichern ihre Daten in einem großen, gemeinsamen DHT.

UsenetDHT [16] wird dazu verwendet die benötigten Ressourcen in Bezug auf Bandbreite und Speicher für einen Usenet Server zu minimieren. Dafür wird die Last mit Hilfe einer DHT, wie zum Beispiel *Chord*, unter den Teilnehmern verteilt. Dabei ist die Menge an Daten, die auf einem Knoten gespeichert wird, antiproportional zur Anzahl der Knoten. Die benötigte Bandbreite für einen Knoten ist proportional zu den *gelesenen* Artikeln.

2.2.3 Overcite

CiteSeer ist ein bekanntes Online Archiv für wissenschaftliche Paper und Artikel. Doch durch die zentralisierte Realisierung werden viele Ressourcen auf einem Server benötigt. Dieses Problem könnte behoben werden, indem die Benutzer jeweils einen Teil ihrer Hardware und Bandbreite zur Verfügung stellen. Dies lässt die Architektur von CiteSeer jedoch nicht zu. OverCite [18] ist ein Vorschlag für eine neue Architektur für ein verteilte und kooperative Forschungsbibliothek, die auf einer DHT basiert. Hier könnten also mit *Chord* die Artikel auf mehreren Rechnern verteilt und effizient gesucht werden. Diese Architektur ermöglicht es durch die Benutzer zur Verfügung gestellten Ressourcen zu

benutzen, um mit großen Datenmengen zu skalieren. Auch sind dadurch zum Beispiel *document alerts* möglich.

3 *Chord* im Vergleich zu anderen P2P-Netzwerken

Zum Vergleich werden hier die bekanntesten Vertreter anderer Protokolle vorgestellt, die auf DHT basieren.

3.1 CAN

Das Content Addressable Network (CAN) [10] ist, wie andere DHT Implementierungen, eine dezentrale P2P Infrastruktur. CAN skaliert mit einer großen Anzahl von Knoten, ist fehlertolerant und selbstorganisierend. Im Gegensatz zu *Chord*, welches als Topologie einen Ring hat, also auf eine Dimension beschränkt, baut CAN auf einem d-dimensionalen kartesischen Raum auf, also auf einem Multitorus. Dieser d-dimensionale Raum ist vollständig logisch. Er ist zyklisch in jeder Richtung und die Knoten haben in jeder Dimension eindeutige Nachbarn. Der Raum wird dynamisch auf die n Knoten aufgeteilt, sodass jeder seine eigene eindeutige Zone hat. Jeder Knoten besitzt eine Routingtabelle, in der er die IP Adressen und die virtuellen Zonen seiner Nachbarkoordinaten pflegt. CAN bildet, genau wie *Chord*, Keys auf Values ab.

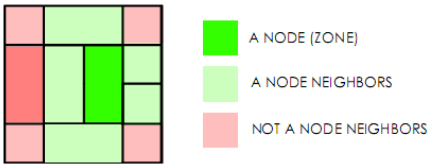

Abbildung 3.1: CAN: Darstellung des Routing in einem 2d-Torus [10]

CAN verfolgt dieselben Ziele wie andere DHT Algorithmen (Skalierbarkeit, gleichmäßige Verteilung der Daten auf viele Knoten, Effizienz). Der Routingalgorithmus ist jedoch anders. Ein Beispiel eines zweidimensionalen Torus wird in Abbildung 3.1 gezeigt. Der betrachtete Knoten ist hier grün gekennzeichnet, seine Nachbarn in den zwei Dimensionen mit hellgrün. Rote Knoten sind keine Nachbarn. Für das Routing wird ein Greedy Algorithmus verwendet, jeder Knoten sendet eine Anfrage an denjenigen Nachbarn weiter, der dem Ziel am nächsten ist.

Die Laufzeit der Suche hängt bei CAN von der Dimension d des Torus ab. Sie ist $O(d * \sqrt[d]{n})$. Abbildung 3.2 zeigt wieder einen zweidimensionalen Torus. Die Zahlen beziffern die Anzahl der Schritte, die der Routingalgorithmus benötigt, ausgehend von dem roten Knoten in der Mitte. Im schlechtesten Fall werden also acht Schritte benötigt. Für solche

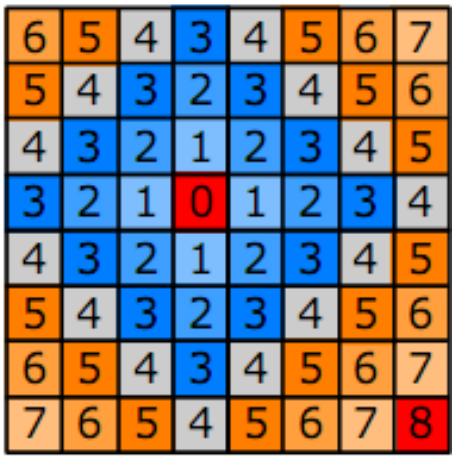

Abbildung 3.2: CAN: Routing in einem 2d-Torus [10]

Die Pfadlänge für das Routing in einem zweidimensionalen Torus. Die Zahlen beziffern die Anzahl der Schritte zu dem roten Knoten in der Mitte. Gleichfarbige Knoten haben den gleichen Abstand zu dem Mittelknoten.

Beispiele, in denen keine fehlerhaften Knoten existieren und keine überlappenden Zonen, beträgt die durchschnittliche Pfadlänge $O(\frac{d}{4} * \sqrt[d]{n})$.

3.2 Kademlia

Kademlia [8] ist ebenfalls eine DHT für ein dezentrales P2P Netzwerk und wurde entworfen von Petar Maymounkov und David Mazières. Es definiert eine Struktur für das zugrunde liegende Netzwerk und wie der Austausch von Informationen mit Hilfe von Suche stattfindet. Die Knoten in einem Kademlia-Netzwerk kommunizieren über das User Datagram Protocol (UDP), ein verbindungsloses ungesichertes, aber gegenüber dem Transmission Control Protocol (TCP) effizienteres Protokoll. Dabei bilden sie ein Overlay-Netzwerk und setzen auf einem vorhandenen Netzwerk auf, zum Beispiel dem Internet. Jeder Knoten wird, ähnlich wie bei *Chord*, durch eine eindeutige *Node-ID* identifiziert. Diese dient auch dazu die Values zu finden. Die Keys werden mit einer Hashfunktion abgebildet und somit den Knoten über deren *Node-IDs* zugeordnet. Das ist ein entscheidender Unterschied zu dem *Chord#* Protokoll, bei dem keine Hashfunktion mehr verwendet wird. Darauf wird noch näher in Kapitel 4 eingegangen.

Die Informationen, die ein Knoten über das Netzwerk hat, nehmen mit dem Abstand ab. Es gibt in Kademlia ein Konzept von Entfernung, sodass ein Knoten A stets für zwei beliebige Knoten B und C entscheiden kann, welcher weiter entfernt ist. Dieser Abstand wird über die XOR-Metrik realisiert. Die Entfernung von zwei Knoten A und B ist demnach das bitweise XOR ihrer beiden *Node-IDs*. Da *Node-IDs* eindeutig sind, haben

zwei Knoten B und C auch immer einen unterschiedlichen Abstand zu Knoten A. Da die XOR-Funktion symmetrisch ist, ist ebenfalls der Abstand zwischen A und B der gleiche wie zwischen B und A. Dies ist wichtig, da auch die Keys über die Entfernungsfunktion auf die Knoten verteilt werden und hier ein Konsens zwischen den beteiligten Knoten herrschen muss, welchem Knoten der Key zugeordnet werden muss. Es gilt ebenfalls die Dreiecksungleichung, sprich die Entfernung von A über B zu C ist länger als der direkte Weg von A nach C.

Ein Knoten A hat detaillierte Information über seine benachbarten beziehungsweise in der Nähe liegenden Knoten. Je weiter die Knoten entfernt sind, desto weniger *Node-IDs* kennt A in dem jeweiligen Teil des Netzes. Dies ist vergleichbar zu den Fingern in *Chord* und *Chord#*. Auch hier kennt ein Knoten viele Finger, die nahe gelegen sind (zum Beispiel die Knoten mit Abstand 1, 2 und 4). Auch hier nimmt das Wissen über Finger, die weiter entfernt sind, ab. So kennt ein Knoten A in *Chord* und *Chord#* in dem Entfernungsbereich 100 bis 200 nur einen Knoten (bei 128). Dies bezieht sich auf den idealen Zustand eines Chordalen Rings und kann leicht abweichen.

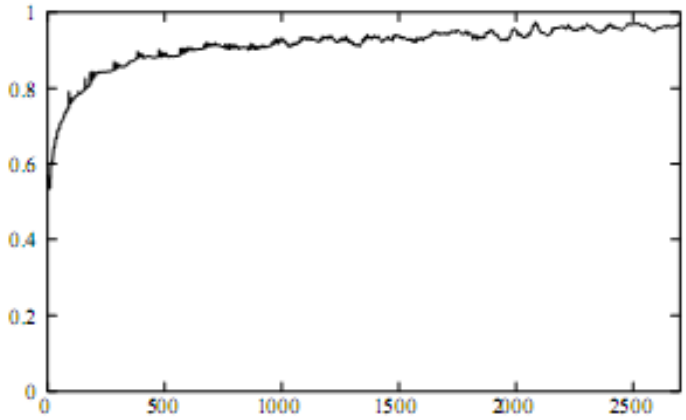

Abbildung 3.3: Kademlia: Betriebszeit der Knoten [8]

X-Achse: Zeit in Minuten, Y-Achse: Wahrscheinlichkeit, dass ein Knoten bei einer Betriebszeit von x Minuten noch eine weitere Stunde online bleibt

Die Ziele von Kademlia sind denen von *Chord* und *Chord#* sehr ähnlich. Auch hier wird eine Abbildung von Keys auf Values realisiert, um den zuständigen Knoten für bestimmte Daten zu finden (hier meistens Dateien). Für die Suche hat auch Kademlia logarithmische Laufzeit. Ein großer Unterschied existiert jedoch bei der Wartung der Finger. Bei Kademlia werden die Informationen über Finger bei jeder Art von Nachricht aktualisiert, also beispielsweise auch bei Suchanfragen. Bei *Chord* und *Chord#* werden Suchanfragen und Ringstabilisierung getrennt, es gibt spezielle Nachrichten für die Ringstabilisierung und Aktualisierung der Finger. Bei Kademlia werden Finger jedoch nur

dann durch einen neuen ersetzt, wenn der alte nicht mehr zu erreichen ist. Diese Entscheidung beruht auf der Beobachtung, dass Knoten, die schon lange online sind, die größte Wahrscheinlichkeit haben, auch weiterhin online zu bleiben. Abbildung 3.3 zeigt Knoten aus einem Gnutella-Netzwerk. Die X-Achse stellt die Betriebszeit der Knoten in Minuten dar, die Y-Achse die Wahrscheinlichkeit, dass ein solcher Knoten eine weitere Stunde online bleibt (genauer: den Anteil an Knoten, der x Minuten online war und auch x + 60 Minuten online geblieben ist).

3.3 Pastry

Pastry [12] ist ein weiteres Overlay- und Routing-Netzwerk für die Implementierung einer verteilten Hashtabelle ähnlich zu *Chord* und *Chord#*.

Das Besondere an Pastry ist, dass es bei dem Routing zu dem zuständigen Knoten die Strukturen und Gegebenheiten des zugrunde liegenden Underlay-Netzwerks (zum Beispiel das Internet) berücksichtigt und nicht nur das Overlay-Netzwerk, dass durch Pastry gebildet wird. Letzteres hat, wie die meisten verteilten Hashtabellen, das Problem, dass im Overlay-Netzwerk benachbarte Knoten meist tatsächlich weit voneinander entfernt liegen und so zum Beispiele viele Hops auf der IP-Schicht zurückgelegt werden müssen. Dies kommt dadurch, dass sich Knoten eine zufällige ID geben.

NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	**1**	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	**2**	10-3-23302
102-0-0230	102-1-1302	102-2-2302	**3**
1023-0-322	1023-1-000	1023-2-121	**3**
10233-0-01	**1**	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Abbildung 3.4: Pastry: Routingtabelle [12]

Pastry Routinginformationen aus Sicht von einem Knoten mit der *Node-ID* 10233102. Alle *Node-IDs* sind Zahlen zur Basis 4. Die hervorgehobenen Zellen zeigen die jeweils zur eigenen *Node-ID* passende Ziffer.

Bei Pastry wird ebenfalls ein Hashwert von den Keys berechnet. Die k zu diesem Hashwert dichtesten Knoten speichern das zugehörige Key-Value-Paar. Damit ist nicht nur

die Replikation realisiert und das Fortbestehen der Daten auch bei Ausfällen von Knoten gesichert, sondern es kann auch beim Routing derjenige von den k Knoten ausgewählt werden, der dem Initiator der Suche auf Internetebene am dichtesten liegt.

Um dies zu realisieren, hat jeder Knoten einige Routinginformationen gespeichert (siehe Abbildung 3.4). Für nähere Ausführungen über den Aufbau und die Instandhaltung der Routinginformationen, sowie über den Routingalgorithmus wird auf das Paper [12] verwiesen. Entscheidend ist hier nur, dass aus der Menge von Knoten, die für ein Präfix zuständig sind, jeweils derjenige in die Routingtabelle eingetragen wird, der am nächsten liegt. Das Kriterium hierfür kann variiert werden, zum Beispiel Antwortzeit oder Anzahl Hops im Underlay-Netzwerk.

3.4 Tapestry

Auch Tapestry [20] ist eine DHT, die eine dezentrale Infrastruktur mit Objektlokalisierung, Routing und Multicasts anbietet. Ähnlich wie Pastry bezieht auch Tapestry das Underlay-Netzwerk mit ein. Es benutzt auch wie Pastry ein Präfix- und Suffix-Routing, womit auch die gleichen Laufzeiten für das Einfügen und Löschen erreicht werden, sowie ähnlich viel Instandhaltungskosten entstehen. Der Unterschied zu Pastry besteht in dem Replizierungsalgorithmus. Tapestry platziert Replikate entlang des Suchweges, um die Wahrscheinlichkeit zu erhöhen, dass eine neue Suche früher beantwortet werden kann.

Der Aufwand für Joins und Failures ist aber aufgrund der Komplexität wesentlich größer als bei *Chord* und *Chord#*. Deswegen skaliert Tapestry schlechter in Netzen, in denen Knoten häufig kommen und gehen.

4 Chord#

> Der Mensch von heute hat nur
> ein einziges wirklich neues Laster
> erfunden: die Geschwindigkeit.
>
> *(Aldous Huxley)*

Chord# ist eine Erweiterung des *Chord*-Protokolls. Es ermöglicht ohne Laufzeiteinbußen auf die Hashfunktion zu verzichten. Durch den Wegfall der Hashfunktion ergeben sich der Vorteil, das Bereichsabfragen (Range Queries) realisierbar sind. Dafür muss jedoch die Lastverteilung implementiert werden, sie findet jetzt nicht mehr implizit durch die Verteilung der Hashfunktion statt, denn im Gegensatz zu *Chord* sind die Knoten *nicht* gleichmäßig über den Keyspace verteilt. Das muss bei der Fingertabelle beachtet werden. Wurde diese bei *Chord* noch aufgebaut, indem ausgerechnet wurde, wie weit der i-te Finger im Keyspace entfernt sein muss, so wird bei *Chord#* die Fingertabelle rekursiv mit Hilfe der Fingertabellen der anderen Knoten aufgebaut.

4.1 Prozeduren

Hier sollen die entscheidenden Prozeduren näher erläutert werden, diese basieren auf Chord [3], jedoch wurden sie an die geänderten Fingertabellen und die nicht-gehashten Keys angepasst. [17]

4.1.1 Join und Punktsuche

```
1  procedure node.join(knownNode)
2      node.successor = knownNode.findSuccessor(node)
3      fixSuccessor()
4      fixFingers()
```

Listing 4.1: Chord#-Prozedur: Join

Die Prozedur *join* (Listing 4.1) wird aufgerufen, um den Knoten *node* dem Ring hinzuzufügen. Dafür wird ein bekannter Knoten *knownNode* benötigt. Zunächst wird die Prozedur *findSuccessor* aufgerufen, um den Nachfolger für den Knoten *node* zu bestimmen. Anschließend wird *fixSuccessor* aufgerufen, um dem Nachfolger mitzuteilen, dass

er einen neuen Vorgänger hat. Schließlich wird noch *fixFingers* ausgeführt, damit der Knoten *node* seine Fingertabelle initialisiert und zur effizienten Suche beitragen kann.

```
1  procedure node.findSuccessor(key)
2      if key ∈ (node.predecessor, node)
3          return node
4      cpf = closestPrecedingFinger(key)
5      return cpf.findSuccessor(key)
6
7  procedure node.closestPrecedingFinger(key)
8      for i = fingerTable.size downto 1
9          if finger_i ∈ (node, key)
10             return finger_i
```

Listing 4.2: Chord#-Prozedur: Find Successor

Die Prozedur *findSuccessor* (Listing 4.2) findet den Nachfolger für einen gegebenen Key *key*. Dafür wird überprüft, ob *key* in dem Zuständigkeitsbereich von *node* liegt, also zwischen dem Vorgänger und *node* selbst. Ist dies der Fall, ist der Knoten *node* zuständig und liefert sich selbst als Ergebnis. Ansonsten muss die Suche an einen möglichst gut geeigneten Knoten weitergegeben werden. Dafür wird die Unterprozedur *closestPrecedingFinger* aufgerufen. Diese bestimmt den Knoten aus der Fingertabelle von *node*, der im Ring möglichst weit von *node* entfernt liegt, ohne jedoch den Knoten, der für *key* zuständig ist, zu überspringen.

Abbildung 4.1 zeigt ein Beispiel einer Suche mit vier Hops. Dabei sucht der Knoten *Stuttgart* nach dem zuständigen Knoten für den Key *Nauen*. Um zu dem gesuchten Knoten *Nürnberg* zu gelangen, wird die Suche in diesem Fall (entlang der roten Pfeile) über die Knoten *Essen, Köln, München, Nürnberg* geleitet. Dabei wird immer mindestens die Hälfte des Weges zum gesuchten Knoten zurückgelegt und damit die logarithmische Laufzeit erreicht. Die grauen Pfeile kennzeichnen Finger, die nicht benutzt werden, da es einen günstigeren Finger gibt, der noch näher an das Ziel herankommt. In einem stabilisierten Ring mit 16 Knoten benötigt die Suche also im schlechtesten Fall 4 Hops ($\log_2(16) = 4$).

4.1.2 Stabilisierung

Die Prozedur 4.3 für die Stabilisierung wird von jedem Knoten *node* periodisch aufgerufen. Dabei wird geprüft, ob die eigene Position im Ring richtig ist, indem *node* den Vorgänger des Nachfolgers erfragt. Erwartungsgemäß ist dies wieder der Knoten *node* selbst. Ist dies nicht der Fall, bestimmt *node* entweder einen neuen Nachfolger (Zeile 4) und ruft danach bei seinem (eventuell aktualisierten) Nachfolger *iThinkIAmYourPred* auf, um ihm mitzuteilen, dass *node* wahrscheinlich sein Vorgänger ist. Dies muss jedoch nicht zwangsläufig stimmen, da zum Beispiel in der Zwischenzeit ein neuer Knoten hinzugekommen sein kann, von dem *node* nichts weiß, der aber ein besseren Vorgänger für

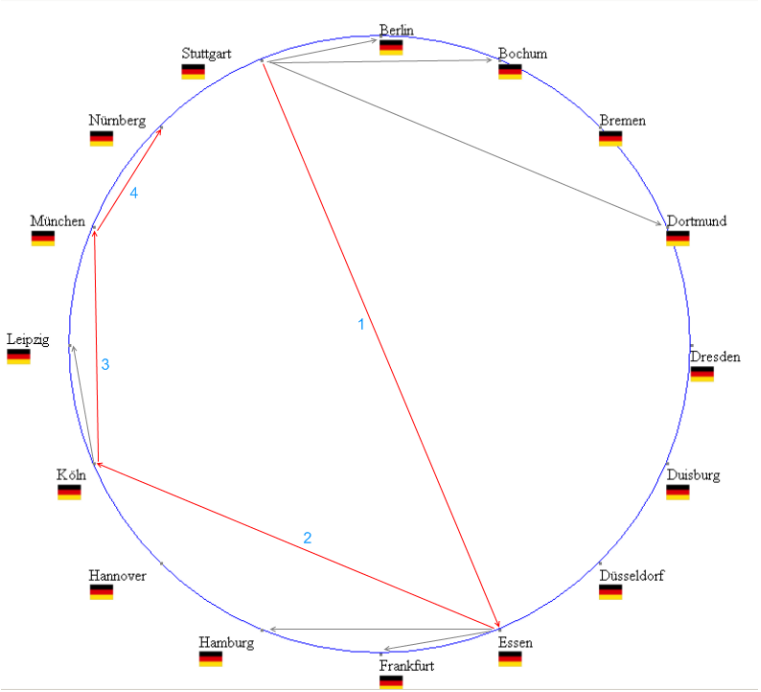

Abbildung 4.1: Chord#: Suche

```
1  procedure node.fixSuccessor()
2      preOfSucc = node.successor.predecessor
3      if preOfSucc ∈ (node, node.successor)
4          node.successor = preOfSucc
5      node.successor.iThinkIAmYourPred(node)
6
7  procedure node.iThinkIAmYourPred(pre)
8      if pre ∈ (node.predecessor, node]
9          node.predecessor = pre
```

Listing 4.3: Chord#-Prozedur: Fix Successor

den Nachfolger ist. Deswegen wird bei der Prozedur *iThinkIAmYourPred* noch einmal geprüft, ob der Knoten *pre* tatsächlich in dem richtigen Intervall liegt (Zeile 8). Nur wenn dies tatsächlich zutrifft, wird der Vorgänger neu gesetzt (Zeile 9).

Das regelmäßige Korrigieren der Finger (Listing 4.4) sorgt dafür, dass die Laufzeit für Suche logarithmisch ist, hat aber keine Auswirkung auf die Stabilität des Rings. Die hier entscheidende Zeile 4 sorgt dafür, dass die Entfernung zu den Fingern exponentiell zunimmt. Anders ausgedrückt überschreiten die Finger $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16} ... \frac{1}{2^m}$ der *Knoten* (im Gegensatz zu *Chord*, dort bezieht sich die Aussage auf *Keys*).

Das Abbruchkriterium für die Schleife ist in Zeile 5 realisiert. Nur für den Fall, dass

```
1  procedure node.fixFingers()
2      finger_1 = node.successor
3          for i = 2 to N
4              newFinger = finger_i-1.finger_i-1
5              if newFinger ∈ (finger_i-1, node)
6                  finger_i = finger_i-1.finger_i-1
7              else
8                  end procedure
```

Listing 4.4: Chord#-Prozedur: Fix Fingers

der neue Finger zwischen dem vorigen Finger und *node* selbst liegt, wird er in die Fingertabelle aufgenommen. Wird der Knoten *node* übersprungen, endet die Prozedur.

Die Entfernung ist hier auf das Overlay-Netzwerk (die Ringtopologie) bezogen und nicht auf das zugrunde liegende Underlay-Netzwerk (in den meisten Fällen das Internet). Sie bezieht sich also nur auf die Anzahl der Knoten, die in der Ringtopologie dazwischen liegen und nicht etwa auf die Anzahl der zurückzulegenden Hops.

4.1.3 Bereichsabfragen

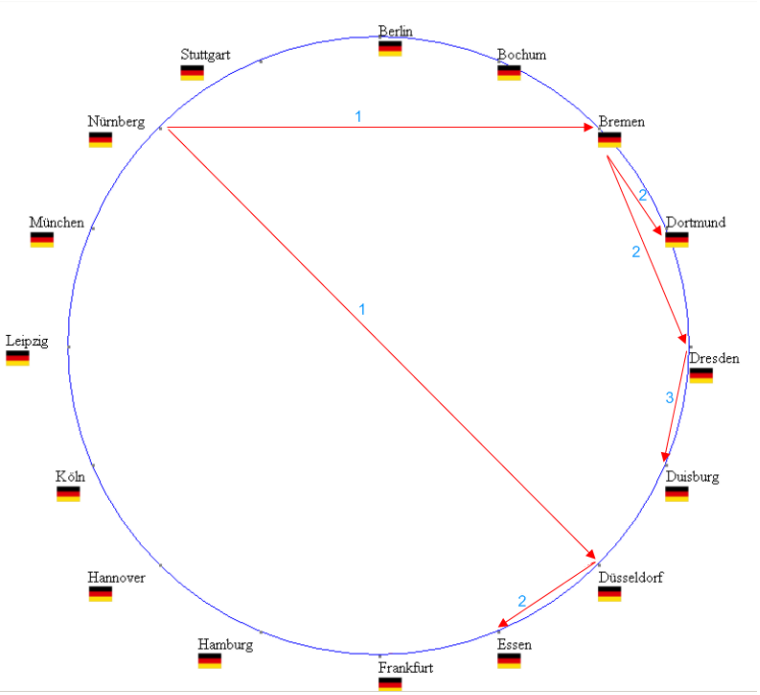

Abbildung 4.2: Chord#: Bereichsabfrage (*Nürnberg* sucht Bereich: C–E)

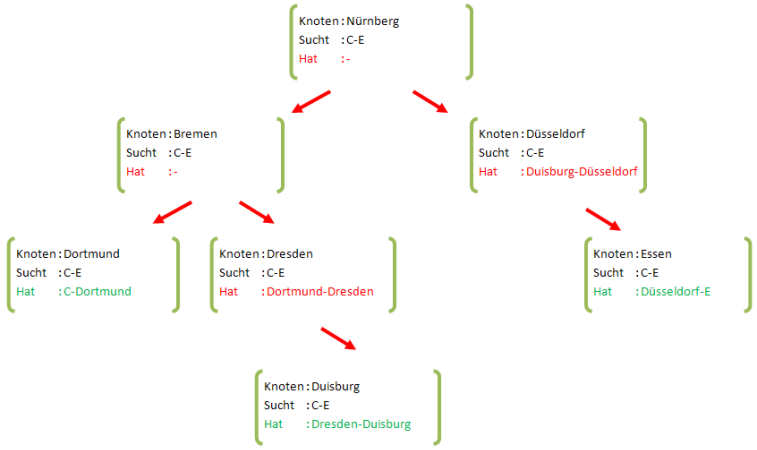

Abbildung 4.3: Chord#: Baum für die Bereichsabfrage (*Nürnberg* sucht Bereich: C–E)

Suchanfrage muss weitergeleitet werden
Suchanfrage musst nicht weitergeleitet werden, Teilbereich wird abgedeckt

Abbildung 4.2 zeigt eine Beispiel für eine Bereichsabfrage auf den Bereich C–E (Da die obere Grenze exklusiv ist werden alle Keys erreicht die den Anfangsbuchstaben C oder D haben). Der aufgespannte Suchbaum wird in Abbildung 4.3 verdeutlicht.

Die naive Prozedur für die Bereichsabfrage verläuft in zwei Schritten. Zunächst wird die Anfrage zu dem Knoten geroutet, der für den Anfang des Bereichs zuständig ist. In diesem Beispiel ist das der Knoten *Dortmund*, da er für den Key C zuständig ist. Jetzt wird die Anfrage solange an den Nachfolger weitergeben, bis das Ende des Bereichs erreicht ist. Hier also der Knoten *Essen*, da er für den Key E zuständig ist. Die Anzahl der Nachrichten für dieses Verfahren liegt in $O(n) + m$, wobei n die Anzahl der Knoten im Ring ist und m die Anzahl der Knoten, die zu dem angefragten Bereich dazugehören. In diesem Beispiel ist n = 16 und m = 5.

Die hier verbesserte Prozedur benötigt zwar nicht weniger Nachrichten, jedoch können durch die Baumstruktur viele Nachrichten parallel verschickt werden. Dies macht sich besonders bemerkbar, wenn der Bereich sehr viele Knoten umfasst. Die Laufzeit kann so auf $O(log_2(n)) + O(log_2(m))$ reduziert werden, da die m Knoten nicht mehr linear abgearbeitet werden. Die Laufzeit ist dann die Höhe des Baums, die Anzahl der Nachrichten bleibt gleich.

Besonders für die effizientere Variante ist es notwendig, dass Bereichsabfragen eine eindeutige Bezeichnung haben, damit doppelte Anfragen, die einen Knoten erreichen, ignoriert werden können. Dies kann in einem dynamischen Netzwerk leicht passieren. Außerdem kann es sinnvoll sein, die Bereichsabfrage über mehrere Wege an die zuständigen Knoten zu liefern, um eine möglichst große Robustheit gegen Ausfälle zu erzielen. In Abbildung 4.3 wird deutlich, dass sonst ein einziger Ausfall (Knoten *Bremen*) dazu führt, dass der ganze Bereich von C bis Düsseldorf nicht erreicht wird, obwohl die Knoten

Dortmund, *Dresden* und *Duisburg* nicht ausgefallen sind.

Für die Bereichsabfragen gibt es mehrere Anwendungsmöglichkeiten. Auf den Knoten kann eine beliebige Prozedur aufgerufen werden. Eine einfache Nutzung dafür wäre zum Beispiel die Beantwortung einer Suchanfrage. So würden alle Key-Value-Paare aus dem Bereich C–E an den Initiator der Suche geschickt werden. Falls nur die Anzahl an Keys in diesem Bereich von Interesse ist und nicht die eigentlichen Values, kann man die Knoten ihre Keys in dem Bereich zählen lassen und nur einen Zahlenwert als Auswertung erhalten. Es sind auch komplexere beliebige Anwendungen möglich, wie zum Beispiel diesem Bereich einen höheren Replikationsfaktor zuzuweisen, da man ihn für besonders wichtig erachtet. Bei der Verwendung des *Chord#* Algorithmus für eine Newsgroups-Anwendung , könnte man sich mit einer solchen Bereichsabfrage für eine Menge an Newsgroups (als Wildcard ausgedrückt beispielsweise alt.comp.hardware.*) als Abonnent eintragen.

```
1  procedure node.rangeQuery(queryID, min, max, operation())
2      if queryID is new
3          firstNode = node.findSuccessor(min)
4          firstNode.rangeQueryTree(min, max, operation())
5
6  procedure node.rangeQueryNaive(min, max, operation())
7      foreach key in node.data ∈ [min, max]
8          key.operation()
9
10     if node.successor ∈ [min, max]
11         successor.rangeQueryNaive(min, max)
12
13
14 procedure node.rangeQueryTree(min, max, operation())
15     foreach key in node.data ∈ [min, max]
16         key.operation()
17
18     for i = 1 to node.fingerTable.size
19         if finger_i ∈ [min, max]
20             finger_i.rangeQueryTree(min, max)
```

Listing 4.5: Chord#-Prozedur: RangeQuery

Das Listing (4.5) zeigt beide Varianten der Bereichsabfrage, die sich im Wesentlichen dadurch entscheiden, an wen die Bereichsabfrage weitergegeben wird. Beide Varianten erhalten den Bereich (*min* und *max*) und eine Prozedur (*operation*), die auf den Keys ausgeführt werden soll.

Bei beiden wird zunächst anhand der *queryID* geprüft (Zeile 2), ob die Bereichsabfrage zuvor schon bearbeitet wurde. Dies kann durch einen Cache realisiert werden, der nach einer gewissen Zeit die *queryIDs* wieder freigibt, der Mechanismus ist an dieser Stelle nicht entscheidend. Dann wird in Zeile 3 der erste Knoten des Bereichs bestimmt und in Zeile 4 das Abarbeiten gestartet (hier wurde die Variante mit dem Baum gewählt).

Die Knoten, auf denen dann *rangeQueryNaive* oder *rangeQueryTree* aufgerufen wird, führen die übergebene Prozedur auf jedem ihrer Keys aus und übergeben die Bereichsabfrage weiter. Die naive Prozedur übergibt nur an den Nachfolger (Zeile 11), die erweiterte Prozedur an alle Finger, die noch in dem Bereich liegen (Zeile 20).

4.1.4 Lastverteilung

Da bei *Chord#* auf die Hashfunktion verzichtet wird, kann man nicht mehr davon ausgehen, dass die Keys gleichmäßig über den Ring verteilt sind. Deswegen ist es notwendig, dass die Knoten in regelmäßigen Abständen eine *Lastverteilung* [4] [14] ausführen um sicherzustellen, dass jeder Knoten in etwa die gleiche Anzahl an Keys hat. Die *Lastverteilung* hat im Vergleich zur Suche von Keys und den Updates der Fingertabelle eine eher geringe Priorität, da eine schlechte Lastverteilung die Suche nicht wesentlich beeinflusst. Es soll aber sichergestellt werden, dass nicht dauerhaft auf sehr wenigen Knoten viele Keys gespeichert sind und diese damit überlastet sind.

Im Wesentlichen besteht die Lastverteilung aus einer geschickten Umbenennung eines Knotens. Damit ändert er seine Position so im Ring, dass er einen anderen Knoten entlastet und etwa die Hälfte seiner Keys übernimmt. Es muss dafür jedoch geprüft werden, ob eine Umbenennung den Aufwand rechtfertigt, den das Verschieben der Keys mit sich bringt.

Jeder Knoten n findet zunächst von den Knoten die ihm bekannt sind, denjenigen mit der höchsten Auslastung. Dafür fragt er alle Knoten aus seiner Fingertabelle nach ihrer aktuellen Auslastung (also Anzahl gespeicherter Keys) und nimmt das Maximum der Antworten. Diesen Wert vergleicht n nun mit der Auslastung, die sein Nachfolger hätte, falls n den Ring an dieser Stelle verlassen würde.

Stellt n nun fest, dass ein Knoten x aus seiner Fingertabelle deutlich mehr Auslastung hat, wird die eigentliche Lastverteilung initiiert. Dafür gibt n zunächst seine Keys an den Nachfolger ab. Dieser Schritt ist nur dazu da, die Wahrscheinlichkeit für den Verlust von Keys im Ring gering zu halten. Auch ohne diesen Schritt würde der Nachfolger früher oder später die Keys in Form von Replikaten erhalten (siehe 4.1.5). Als nächstes lässt sich n den Median *med* der Keys von x geben. Intuitiv also der mittlere Key, der den Bereich, für den x zuständig ist, in zwei gleich große Teile aufteilt. Der Knoten n verlässt nun den Ring und kommt als neuer Knoten mit *Node-ID med* wieder in den Ring hinein. Seinen Nachfolger x kennt er schon, weswegen keine Suche bei diesem Join notwendig ist. Nun bekommt er automatisch von x die Hälfte der Keys zugewiesen.

Um sicherzustellen, dass durch Nebenläufigkeit nicht zu viele Knoten gleichzeitig zu den überlasteten Knoten tendieren, ist es sinnvoll, zufällige Wartezeiten zwischen der Lastverteilung einzuführen. Außerdem wird ein überlasteter Knoten, der nach seinem Median gefragt wird, diesen nur einmal nennen und weiteren anfragenden Knoten für eine gewisse Zeit mitteilen, dass schon eine Lastverteilung stattfindet.

Abbildung 4.4 zeigt ein Beispiel für eine Lastverteilung. Der Knoten *Duisburg* liegt lexikographisch sehr nahe an *Düsseldorf* und hat daher einen relativ kleinen Bereich an Keys, den er bearbeiten muss. Der Knoten *Stuttgart* hat im Gegensatz dazu einen sehr großen lexikographischen Bereich (Stuttgart-Berlin). Sobald dies festgestellt wird,

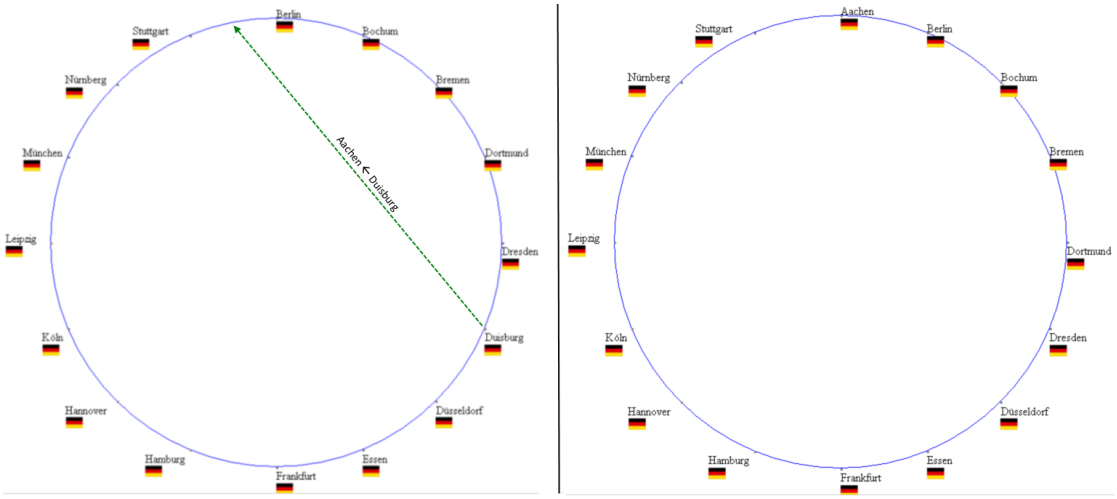

Abbildung 4.4: Chord#: Lastverteilung

links: Vor der Lastverteilung
rechts: Nach der Lastverteilung

benennt sich *Duisburg* in *Aachen* um, um dem Knoten *Stuttgart* einen großen Teil der Arbeit abzunehmen. Der Bereich, für den *Duisburg* vorher zuständig war, fällt nun in den Bereich von *Dresden*.

Die Prozedur zur Lastverteilung (Listing 4.6) lässt sich in zwei Bereiche unterteilen. Zunächst wird in den Zeilen 2–5 bestimmt, welcher der Finger die größte Last hat. Der zweite Abschnitt entscheidet dann darüber, ob eine Lastverteilung stattfinden muss und leitet diese gegebenenfalls ein. Dazu wird in Zeile 7 berechnet, wieviel Last der eigene Nachfolger hätte, würde der Knoten *node* den Ring tatsächlich an dieser Stelle verlassen (dies ist die Summe aus der Last von *node* und *node.successor*. Zeile 8 trifft dann die eigentliche Entscheidung, ob sich die Lastverteilung lohnt. Dafür muss die Last von dem Finger deutlich größer sein, als das berechnete x. Hier wurde als Faktor 1.5 verwendet, um einen guten Kompromiss zu finden zwischen guter Lastverteilung und hohem Aufwand durch den Positionswechsel von Knoten *node*. Sollte die Last des Fingers höher sein, wird die eigentliche Lastverteilung initiiert. Dafür wird in den Zeilen 9–10 (beziehungsweise 17 und 20) jeweils der Nachfolger und Vorgänger von *node* informiert, die dann jeweils den Vorgänger und Nachfolger von *node* übernehmen. Dies wäre eigentlich nicht notwendig, da *node* auch einfach ausfallen könnte und sich der Ring dennoch stabilisieren würde. Es trägt aber zur zügigen Stabilisierung bei. Anschließend werden in Zeile 11 die Daten von *node* an den Nachfolger übergeben und schließlich in Zeile 12 die *Node-ID* von *node* so geändert, dass er die Hälfte der Keys von *hardestWorker* übernimmt. Die neue *Node-ID* ist der Median der Keys von dem Finger, also der Key der den Bereich in zwei gleich große Teile teilt. Zuletzt wird die Prozedur *cleanUp* aufgerufen, um die alten Keys und Finger zu löschen.

```
1  procedure node.loadBalance()
2      hardestWorker = node
3      for i = 1 to node.fingerTable.size
4          if finger_i.load > hardestWorker.load
5              hardestWorker = finger_i
6
7      x = node.load + node.successor.load
8      if hardestWorker.load > 1.5 * x
9          node.successor.preIsLeaving()
10         node.predecessor.succIsLeaving()
11         node.successor.data += node.data
12         node.ID = hardestWorker.median
13         node.successor = hardestWorker
14         node.cleanUp()
15
16 procedure node.preIsLeaving()
17     node.predecessor = node.predecessor.predecessor
18
19 procedure node.succIsLeaving()
20     node.successor = node.successor.successor
21
22 procedure node.cleanUp()
23     node.data = ∅
24     node.fingerTable = ∅
```

Listing 4.6: Chord#-Prozedur: Lastverteilung

4.1.5 Symmetrische Replikation

Eine wichtige Funktion, um die Verfügbarkeit von Keys und deren Values zu garantieren, ist das Erstellen von *Replikaten* . Dafür wird der Bereich, für den ein Knoten zuständig ist, erweitert. Wird jeder Key im Ring nur einmal gespeichert (also findet gar keine Replikation statt), so ist der Bereich eines Knotens genau der Wertebereich zwischen seiner *Node-ID* und der des Vorgängers. Wie schon in Abschnitt 4.1.4 beschrieben, sind diese Bereiche unterschiedlich groß und hängen von der Ballung der Keys ab. Für die Replikate werden zusätzliche Bereiche berechnet, die von dem Keyspace N und dem Replikationsfaktor f abhängen.

Für die Berechnung des i-ten Replikats für den Key x gilt:

$$r(i,x) = i \oplus (x-1)\frac{N}{f} \tag{4.1}$$

Als Modulo für die Klassenzugehörigkeit wird $m = \frac{N}{f}$ gewählt.

Es entstehen Äquivalenzklassen von Knoten. Dies wird an Abbildung 4.5 deutlich. In diesem Beispiel ist der Replikationsfaktor f=2, der Keyspace N=16 und der Modulo $m = \frac{16}{2} = 8$.

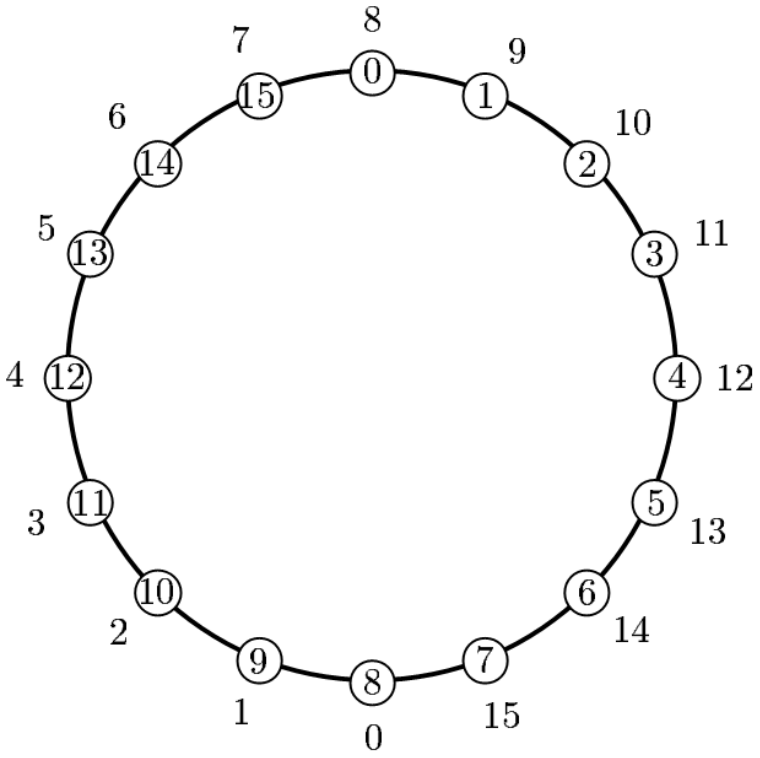

Abbildung 4.5: Beispiel für Symmetrische Replikation

Der Keyspace ist hier N=16, der Replikationsfaktor f=2. Die Zahlen in den Kreisen bezeichnen das erste Replikat, die Zahlen außerhalb der Kreise das zweite Replikat.

So errechnen sich die Beiden Replikate für die 1 durch:

$$r(1,1) = 1 \oplus (1-1)\frac{16}{2} = 1 \tag{4.2}$$

$$r(1,2) = 1 \oplus (2-1)\frac{16}{2} = 9 \tag{4.3}$$

Diese beiden Keys bilden eine der Äquivalenzklassen da $1 \equiv 9 \ (mod\ 8)$. Analog dazu sind {2,10},{3,11} usw. ebenfalls Äquivalenzklassen. Für diese gilt, dass jeder Knoten der für *einen* Key aus einer Klasse verantwortlich ist, auch gleichzeitig für *alle* Keys aus dieser Klasse verantwortlich ist.

4.1.6 TimeChecks

Die timeCheck Prozedur (4.7) wird periodisch ausgeführt. Jede Prozedur hat seinen eigenen Timer, durch Tests wurden günstige Werte für die einzelnen Parameter bestimmt. Bei den Tests auf PlanetLab (siehe Kapitel 5 und 6) haben sich folgende Größenordnungen für die einzelnen Zeiten und Parameter als günstig erwiesen. Entscheidend sind

```
1  procedure node.timeCheck()
2      if timer.successorsExpired   (1 Minute)
3          node.fixSuccessor()
4      if timer.fingersExpired      (2 Minuten)
5          node.fixFingers()
6      if timer.idsExpired          (5 Minuten)
7          node.refreshIds()
8      if timer.dataExpired         (10 Minuten)
9          node.refreshData()
10     if timer.loadExpired         (20 Minuten)
11         node.loadBalance()
12
13     if timer.queryExpired        (1 Minuten)
14         node.sendRandomQuery()
15     if timer.statsExpired        (20 Minuten)
16         node.sendStats()
```

Listing 4.7: Chord#-Prozedur: Timechecks

hier mehr die Verhältnisse zueinander, nicht die absoluten Werte. Man könnte also alle Werte beispielsweise verdoppeln und würde immer noch eine sinnvollen, aber verlangsamten Ablauf des Algorithmus beobachten. Eine Ausnahme bildet jedoch hier die Prozedur *fixSuccessor*. Diese *muss* in kurzen Abständen ausgeführt werden, weil der Ring sonst auseinander fällt.

Ändert man die Verhältnisse zum Beispiel so, dass die Lastverteilung häufiger als das Stabilisieren des Rings stattfindet, stellt man einen großen Zuwachs an (unnötigem) Datenverkehr fest.

Gewählt wurden folgende Werte für die Tests in PlanetLab, da sie sich als guter Kompromiss zwischen Netzlast und Stabilität beziehungsweise Verfügbarkeit herausgestellt haben. Mit diesen Werten sind auch die Auswertungen in Kapitel 6 entstanden.

fixSuccessor ist für die Ringstabilisierung zuständig und damit die wichtigste und am häufigsten aufgerufene Prozedur. Um den Ausfall beziehungsweise das Verlassen von Knoten im Ring zu bemerken, muss eine kleine Zahl von Nachfolgern aktuell gehalten werden. Diese Zahl ist konstant, also unabhängig von der Ringgröße und hängt davon ab, wieviele Knoten erwartungsgemäß gleichzeitig ausfallen. In den hier vorgenommenen Tests (oft über Tage oder Wochen) war es ausreichend fünf Nachfolger zu speichern, um den Ring auch bei Ausfällen von bis zu 50% der Knoten zusammenzuhalten. Die Wahrscheinlichkeit, dass selbst bei 50% Ausfall *alle* Nachfolger ausfallen ist nur etwa 3%.

$$p = 0.5^5 = 0.03125 \tag{4.4}$$

Für ein Auseinanderfallen des Rings müssten die 50% aber alle innerhalb einer *fixSuccessor*-Periode ausfallen, weil sonst der Ausfall erkannt wird und neue Nachfolger in die Liste aufgenommen werden. Abbildung 4.6 zeigt einen simulierten Ausfall von zufälligen

50% der Knoten und den stabilisierten Ring danach.

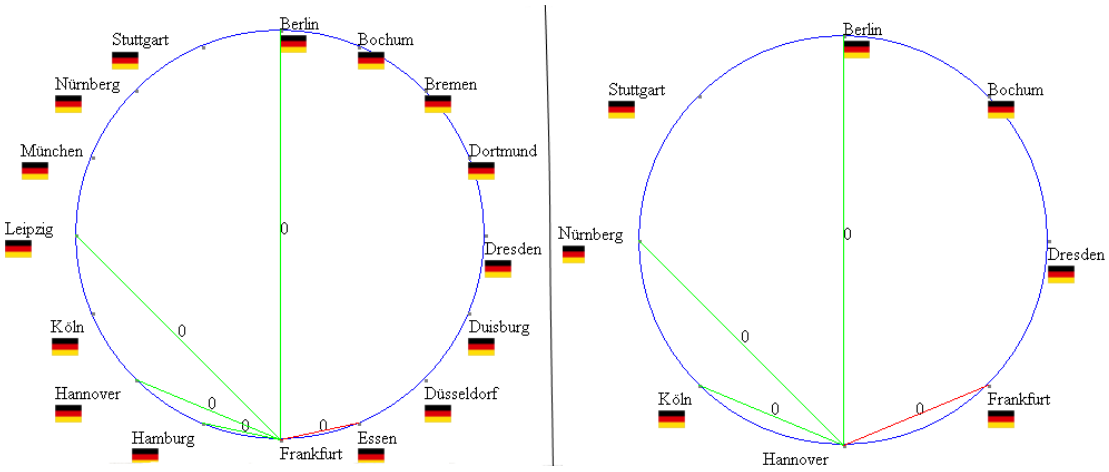

Abbildung 4.6: Chord#: Beispiel für Ausfall von 50% der Knoten

links: stabilisierter Ring mit 16 Knoten.
rechts: nach Ausfall von 8 Knoten erneut stabilisierter Ring.

Die Prozedur *fixFingers* muss etwas weniger oft ausgeführt werden. Sie dient zwar dazu die Suche in logarithmischer Laufzeit zu ermöglichen, indem durch die aktuellen Finger sichergestellt wird, dass ein großer Teil des Wegs in einem Schritt abgedeckt wird, ist jedoch nicht für die Stabilisierung zuständig.

Wegen der Lastverteilung (4.1.4) kann es vorkommen, dass Knoten ihre *Node-ID* ändern. Für das Routen der Suchanfragen ist es erforderlich, dass Knoten die aktuelle *Node-ID* ihrer Finger kennen, jedoch ist es unnötig diese ständig (bei jedem Prozeduraufruf) zu erfragen. Deswegen verwenden die Knoten einen Cache für die *Node-IDs*, der alle 5 Minuten aktualisiert wird. Zu häufige Refreshs würden die Netzlast erhöhen, zu seltene würden die Laufzeit für die Suche verschlechtern, in Extremfällen sogar zu falschen Ergebnissen führen. Die Auswirkungen des Cache auf die Netzlast werden in den Abbildungen 4.7 und 4.8 deutlich.

Die ersten Auswertungen haben gezeigt, dass der Großteil an versendeten Bytes für das Erfragen der *Node-ID* verwendet wird. Durch die Einführung des Caches konnte der durchschnittliche Netzwerkverkehr eines Knotens von ca. 1000 Bytes/Sekunde auf etwa 250 Bytes/Sekunde reduziert werden.

Die Prozedur *refreshData* kümmert sich um die eigentlichen Keys und Values, die ein Knoten lokal in einer Datenbank hat. In (relativ großen) Zeitabständen wird geprüft, ob der Knoten noch für alle Daten in seiner Datenbank zuständig ist, da sich durch Joins, Leaves und Lastverteilung die Bereiche verschieben. Außerdem wird in dieser Prozedur die Replikation der Daten realisiert. Ein Knoten errechnet die Bereiche, für die er zuständig ist und fragt diese Daten von den anderen Knoten an, sofern er sie nicht schon hat.

Die Lastverteilung kann in noch größeren Abständen ausgeführt werden, da sich typi-

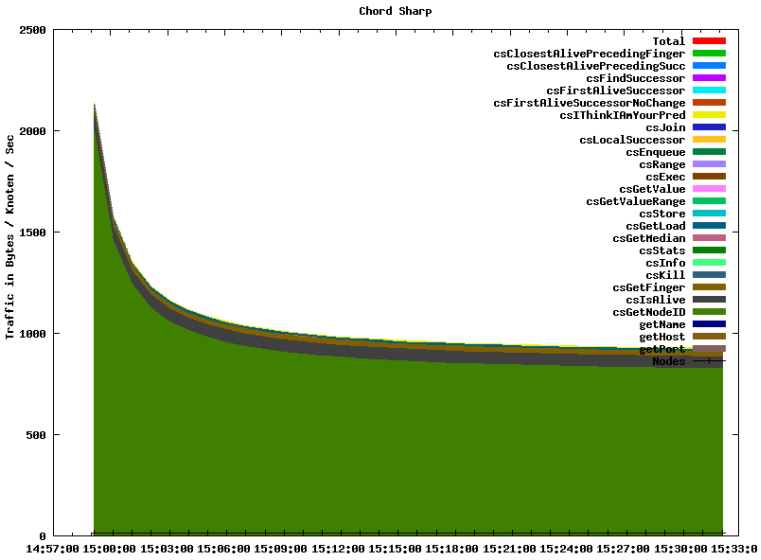

Abbildung 4.7: Chord#: Ohne Cache für die *Node-IDs*

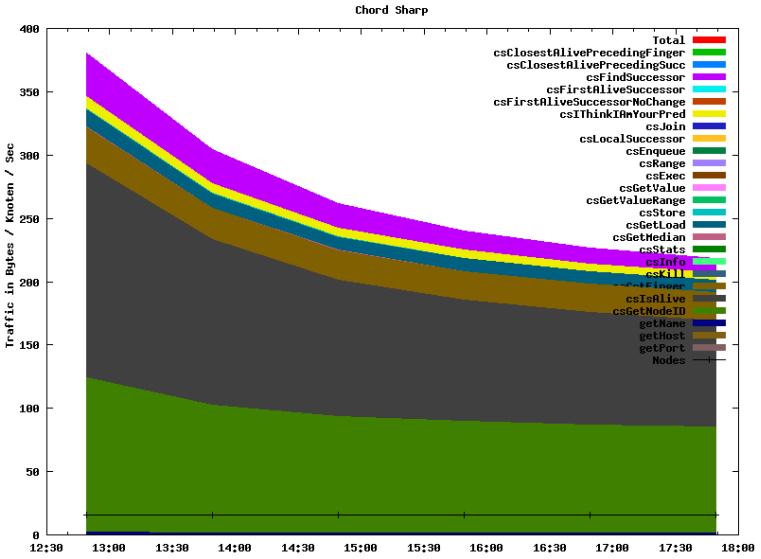

Abbildung 4.8: Chord#: Mit Cache für die *Node-IDs*

scherweise der Bestand an Keys nicht so schnell ändert, dass einzelne Knoten in kurzer Zeit überlastet sind. Die Lastverteilung wurde in Abschnitt 4.1.4 genau beschrieben.

Die letzten beiden Prozeduren sind für den eigentlichen Algorithmus nicht erforderlich,

sondern wurden für das Sammeln von Statistiken implementiert. Die Prozedur *sendRandomQuery* simulierte einen Benutzer, der eine Suchanfrage stellt. Dabei wurde gemessen, wie lange es gedauert hat, bis die Antwort kam und wieviele Nachrichten dabei verschickt wurden. Die Auswertungen werden in Abschnitt 6.2.2 erläutert.

Ebenfalls für die Auswertungen mussten die Knoten sämtliche Statistiken sammeln und in regelmäßigen Abständen an einen Statistikserver übermitteln. Unter anderem wurde vermerkt, wie oft welche Methode aufgerufen wurde, wie lange ein Knoten schon online ist, wie gut seine Verbindung zu seinen Nachbarn ist, sowie detaillierte Statistiken über Suche, Lastverteilung und Replikation.

4.2 Unterschiede zu *Chord*

4.2.1 Fingertabelle – *Chord*

In Chord berechnet ein Knoten n seine Finger f durch:

$$f_i = successor(n + 2^{i-1}) \ \ mod \ \ 2^m, 1 \leq i \leq m \tag{4.5}$$

Der Parameter m bestimmt die Anzahl der Finger und hat eine Größenordnung von *log(N)*. Die Platzierung der Finger richtet sich also nach dem Keyspace. Dabei wird von einer gleichmäßigen Verteilung der Knoten über den Keyspace ausgegangen, abhängig von der Güte der Hashfunktion. Deswegen liegt die Laufzeit für die Suche nur mit großer Wahrscheinlichkeit (und nicht garantiert) in *O(log(n))*.

4.2.2 Fingertabelle – *Chord#*

In Chord# berechnet ein Knoten n seine Finger f durch:

$$f_i = \begin{cases} successor & : i = 0 \\ f_{i-1}.f_{i-1} & : i \neq 0 \end{cases} \tag{4.6}$$

Einer der großen Unterschiede von *Chord#* und *Chord* ist die Handhabung der Fingertabelle. Da bei *Chord* die Knoten gleichmäßig über den Keyspace verteilt sind, wird dort nur die *findSuccessor* Prozedur mit einem berechneten Key aufgerufen, sodass der Ergebnisknoten die gewünschte Entfernung im Ring hat. Bei *Chord#* ist der i-te Finger der *(i-1)*-te Finger des *(i-1)*-ten Fingers. Auch dadurch nimmt die Entfernung im Ring der Finger exponentiell zu. Im idealen Fall liegen also zwischen einem Knoten und seinem i-ten Finger genau $(2^i - 1)$ Knoten. Durch die exponentiell wachsende Entfernung der Knoten in der Fingertabelle, wird also bei einer Suche bei jedem Hop mindestens die Hälfte des Weges zum Ziel zurückgelegt. Dadurch ergibt sich die logarithmische Laufzeit.

4.2.3 Äquivalenz der Fingertabellen von *Chord* und *Chord#*

In den folgenden zwei Abbildungen werden die Fingertabellen von *Chord* und *Chord#* verglichen. In beiden Fällen ist der Keyspace der Bereich von [0..255]. Der Knoten, der

die Fingertabelle berechnet, ist jeweils der Knoten 0. Die Entfernung ist hier auf die dazwischenliegenden Knoten bezogen.

Der erste Ring (Abbildung 4.9) hat 52 Knoten mit *Node-IDs* (0,5,10,15..255). Der zweite Ring (Abbildung 4.10) hat 32 Knoten mit *Node-IDs* (0,8,16,32..248).

Der entscheidende Unterschied ist hierbei, dass 32 eine Potenz der 2 ist, und daher die Fingertabellen (als Menge betrachtet) äquivalent sind, während die Fingertabellen bei 52 Knoten voneinander abweichen.

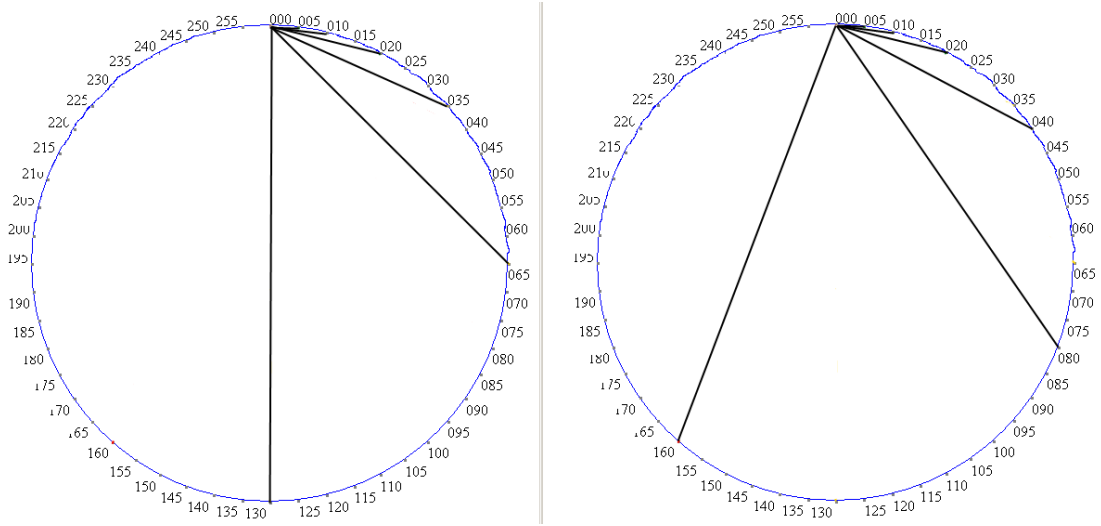

Abbildung 4.9: Vergleich der Fingertabellen bei 52 Knoten

Chord Chord#

	0	1	2	3	4	5	6	7
$p + 2^i$	1	2	4	8	16	32	64	128
$succ(p + 2^i)$	5	5	5	10	20	35	65	130
Chord Entfernung	1	1	1	2	4	7	13	26
Chord# Entfernung	1	2	4	8	16	32		

4.3 Byzantinische Knoten

Byzantinische Knoten, also Knoten, die nicht nur ausfallen, sondern gezielt versuchen dem System zu schaden, wurden bei den Testläufen und in der hier vorliegenden Implementierung nicht berücksichtigt. Sie sollen jedoch erwähnt werden, da dieses Problem für eine produktive Anwendung beachtet werden muss.

Knoten könnten zum Beispiel durch geschicktes Ändern ihrer *Node-ID* einen Bereich des Rings übernehmen und die dortigen Keys blockieren, indem sie die Suchanfragen falsch beantworten. Einfaches Ignorieren der Suchanfragen würde nicht ausreichen, weil dies als Ausfall des Knoten gedeutet würde und die Replikation dafür sorgt, dass wieder

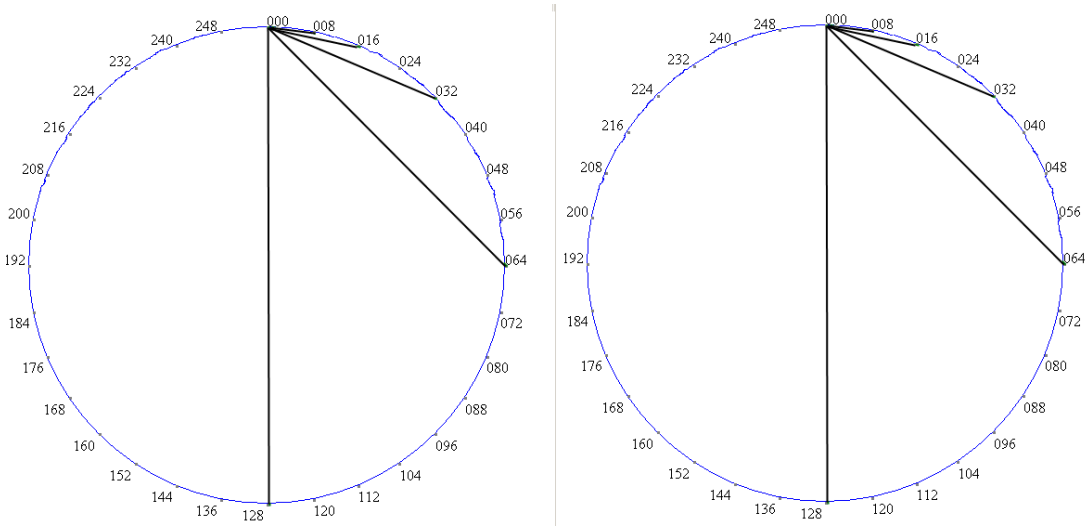

Abbildung 4.10: Vergleich der Fingertabellen bei 32 Knoten.

Chord Chord#

	0	1	2	3	4	5	6	7
$p + 2^i$	1	2	4	8	16	32	64	128
$succ(p + 2^i)$	8	8	8	8	16	32	64	128
Chord Entfernung	1	1	1	1	2	4	8	16
Chord# Entfernung	1	2	4	8	16			

andere Knoten den Bereich zugeteilt bekommen. Durch Ändern der Values könnte gezielt Zensur betrieben werden.

Für *Chord* wurden abgeänderte Algorithmen vorgestellt [13], um den Algorithmus robust gegen byzantinische Knoten zu machen. In diesem Algorithmus sind dann nicht mehr Knoten die kleinste Einheit, sondern *Schwärme* von Knoten. Es wurde gezeigt, dass bei einem Netz, in dem weniger als $\frac{1}{4}$ aller Knoten byzantinisch sind, der Algorithmus korrekt funktioniert.

Bei *Chord#* könnten byzantinische Knoten sogar einen ganzen Bereich blockieren, was bei *Chord* wegen der Hashfunktion nicht so einfach möglich ist.

Bei beiden Algorithmen ist es außerdem möglich einen einzelnen Knoten n zu blockieren. Dafür benennt sich der Angreifer so, dass er der Nachfolger von n wird. Fragt n das nächste Mal den Angreifer nach seinem Nachfolger oder einem Finger, so gibt dieser als Antwort wieder n. Dadurch hat n den Eindruck, dass der Ring nur aus 2 Knoten besteht und wird alle seine Suchanfragen an den Angreifer leiten.

5 Planet-Lab

5.1 Entstehung

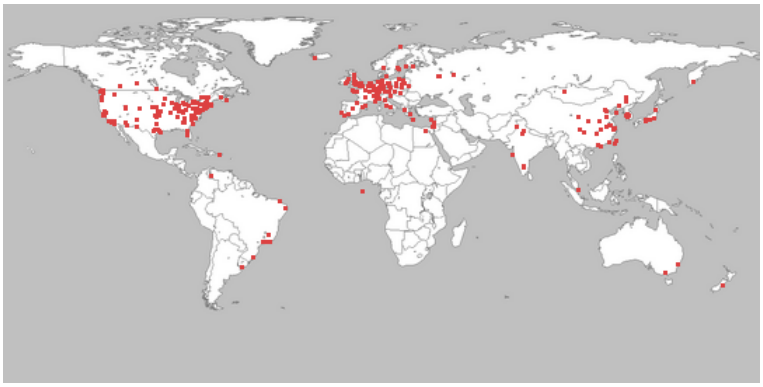

Abbildung 5.1: Verteilung der 840 Knoten (PlanetLab)

(Stand: Dezember 2007)

Der Vorschlag für das PlanetLab-Forschungsnetz wurde im März 2002 eingereicht und von Intel Research finanziert. Im Juni 2002 ging dann der erste PlanetLab-Knoten an das Netz. Etwa ein Jahr später umfasste das Netzwerk bereits 200 Knoten. Das Netz wuchs kontinuierlich an bis zu seiner heutigen Größe von 840 Knoten auf 411 Sites.

5.2 Begriffsklärung

PlanetLab ist ein weltweites Forschungsnetz, das der Entwicklung neuer Netzwerkdienste dient. Beispiele sind verteilte Datenhaltung (distributed storage), Netzwerkabbildungen, P2P Systeme, verteilte Hashtabellen (wie Chord#) und query processing.

5.2.1 Sites

Eine *site* ist z.B. ein Forschungsinstitut wie das Zuse-Institut Berlin, das PlanetLab Rechner als *Knoten* zur Verfügung stellt.

5.2.2 Nodes

Ein *Knoten* ist ein Rechner in PlanetLab, der von allen Benutzern aus PlanetLab genutzt werden darf. Jeder Knoten hat mehrere *Slivers* also ein Scheibchen des Knoten, der für eine *Slice* eingeteilt wurde. Es ist nicht garantiert, dass ein Knoten immer erreichbar ist. Genauso kann ein Rechner jederzeit neu gestartet werden oder auch seine Daten gelöscht werden. Die PlanetLab-Anwender müssen also darauf achten, dass die Algorithmen robust gegen Ausfälle programmiert sind und keine wichtigen Daten auf den Knoten gespeichert werden.

5.2.3 Slices

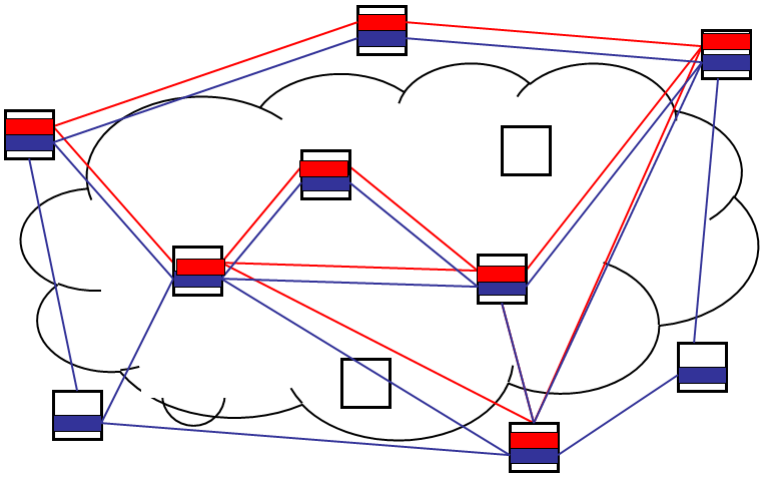

Abbildung 5.2: Beispiel für Slices [9] (PlanetLab)

Zwei Slices die auf zwei (sich teilweise überschneidenden) Mengen von Knoten laufen

Eine Slice entspricht in etwa einem Experiment, also einem Service oder Algorithmus der auf einer Anzahl von Nodes auf PlanetLab läuft.

Jeder Nutzer kann *Slices* erstellen, also eine Menge von Knoten, auf denen er eine Umgebung mit Ressourcen zugeteilt bekommt. Dass ein Knoten an mehreren Slices teilnimmt, bleibt dem Anwender verborgen.

5.2.4 Sliver

Slivers sind virtuelle Server, die auf einem Knoten laufen. Ein User bekommt für seine Slice auf jedem der Knoten eine Sliver zugeteilt. Dort hat er Rootrechte und wird nicht von anderen Usern gestört.

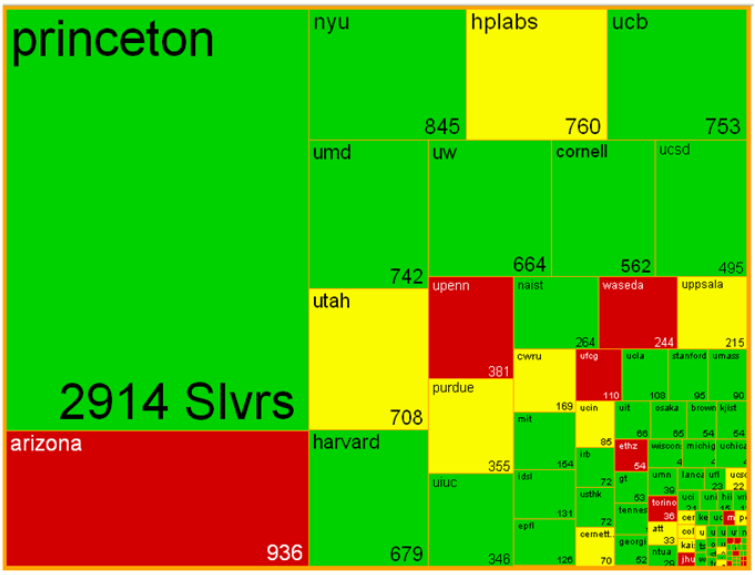

Abbildung 5.3: Verteilung der Slivers (PlanetLab)

(Stand: März 2006)

5.3 Services

Es werden auf PlanetLab eine Vielzahl von unterschiedlichen Services getestet. Es laufen dort Experimente zu Content Distribution (CoDeeN, Cora, Cobweb), Storage bzw. Transfer großer Dateien (LoCI, CoBlitz), Routing (DHARMA, VINI), DNS (CoDNS) und Internetmessungen allgemein.

5.4 Stats

Im März 2006 gab es über 2500 User, die mit 600 Slices einen Netzwerkverkehr von 4 TB verursachten. Die zwei Abbildungen (5.3 und 5.4) zeigen, welche Sites die meisten Slivers und Bandbreite in Anspruch genommen haben.

5.4.1 Vergleich zu Grid5000

Die für diese Arbeit entscheidenden Experimente sind unter realen Bedingungen wie zum Beispiel Rechnerausfällen entstanden. Die Anzahl der Knoten in PlanetLab, die als stabil bezeichnet wurden (also länger als 30 Tage online waren), lag zwischen 15% und 40% (siehe Abbildung 5.5).

Dies war jedoch beabsichtigt. Es gibt auch zum Beispiel Forschungsnetze wie das Grid5000[1], die eine wesentlich höhere Zuverlässigkeit bieten. Dies ist dadurch möglich,

[1] www.grid5000.fr

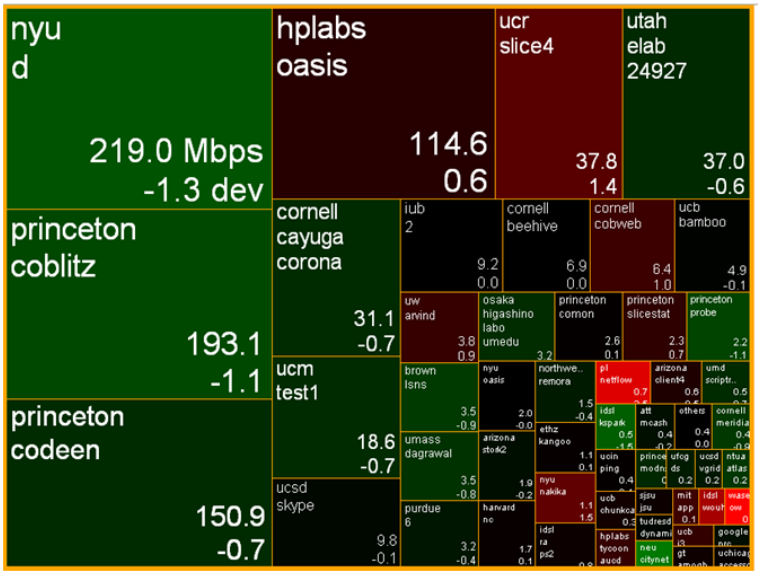

Abbildung 5.4: Verteilung der Bandbreite (PlanetLab)

(Stand: März 2006)

dass die 5000 Prozessoren auf nur 9 Sites verteilt sind, die alle in Frankreich liegen. Jedoch sind dort die Antwortzeiten und die Ausfallzeiten unrealistisch gut. Die Messergebnisse würden erwartungsgemäß den Simulationen auf einem Desktop-Rechner entsprechen. Hier sollte aber der Algorithmus in einer möglichst realen Umgebung getestet und seine Anpassungsfähigkeit analysiert werden.

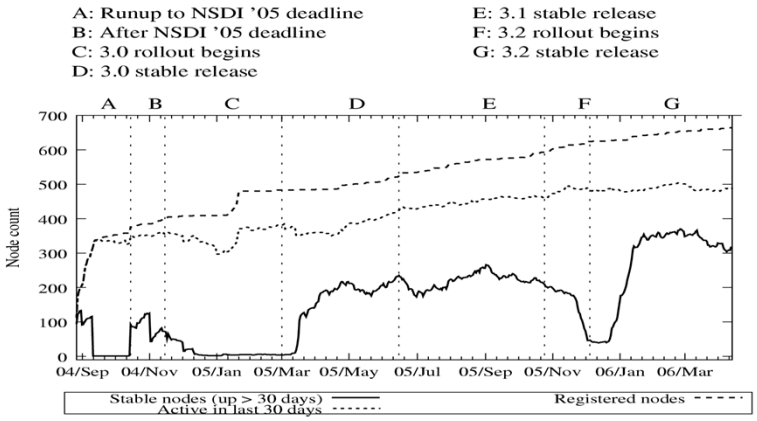

Abbildung 5.5: Verfügbarkeit der Knoten [9] (PlanetLab)

6 Experimente und Auswertung

> Aber man verlangt vom Forscher, daß er Beweise liefert, wenn es sich zum Beispiel um die Entdeckung eines großen Berges handelt, verlangt man, daß er große Steine mitbringt.
>
> *(Antoine de Saint-Exupéry)*

6.1 Realisierung

Allgemein ist das PlanetLab kein zuverlässiges Netz. Es wird nicht garantiert, dass Knoten über einen längeren Zeitraum erreichbar sind oder zuverlässig arbeiten. Knoten können jederzeit ausfallen, neugestartet werden oder auf Dauer unerreichbar sein. Eine zufällige Auswahl an Knoten führt zu einer Zuverlässigkeit von ca. 50%. Deswegen wurden gezielt Knoten ausgewählt, die in den letzten Tagen keinen Ausfall hatten und damit auch wahrscheinlich in den nächsten Stunden keinen Ausfall haben werden. Die Zeitmessungen wurden teilweise nur über Minuten gemacht, zum Beispiel um festzustellen , wie lange es dauert, bis alle Knoten die richtigen Nachfolger und Finger haben. Andere Tests, wie Tests zur Lastverteilung oder Suchanfragen, gingen über Stunden beziehungsweise Tage.

Außerdem wurden zwei unterschiedliche Netze getestet. Es wurden ein europäisches Netz, bestehend aus Knoten hauptsächlich aus Deutschland sowie einigen Nachbarländer (Frankreich, Schweiz, Österreich) und ein weltweites Netz verglichen. Hierbei kam es zu großen Unterschieden, auf die noch näher eingegangen wird.

Besonders in dem weltweiten Netz, aber auch in dem europäischen Netz kam es hin und wieder zu Ausreißern, also Messergebnissen, die sehr stark von den anderen abweichen. Für die folgenden Bilder wurden solche Messergebnisse nicht berücksichtigt, um die Grafiken übersichtlich zu erhalten. Es wurden für eine Grafik jedoch maximal 1% (Perzentil) der Messungen verworfen, somit bleiben die Grafiken auch aussagekräftig, da die verworfenen Messungen die allgemeine Aussage nicht beeinflussen würden.

Die Abbildungen 6.1 und 6.2 zeigen beide Werte aus derselben Messreihe. Abbildung 6.1 enthält die kompletten Messwerte, während Abbildung 6.2 nur die 99% kleinsten Werte darstellt. Damit werden die Ausreißer vernachlässigt, um die Aussagekraft zu erhöhen. So kann man zum Beispiel bei der zweiten Abbildung anhand der Antwortzeiten im Bereich zwischen 20 und 40 Millisekunden erkennen, wieviele Hops die jeweilige Suche benötigt hat (1–3), was bei der ersten Abbildung nicht möglich ist.

Ein Grund für solche Ausreißer ist zum Beispiel, dass die benutzen Knoten in PlanetLab für eine kurze Zeitspanne sehr ausgelastet sind und deswegen gar nicht oder sehr

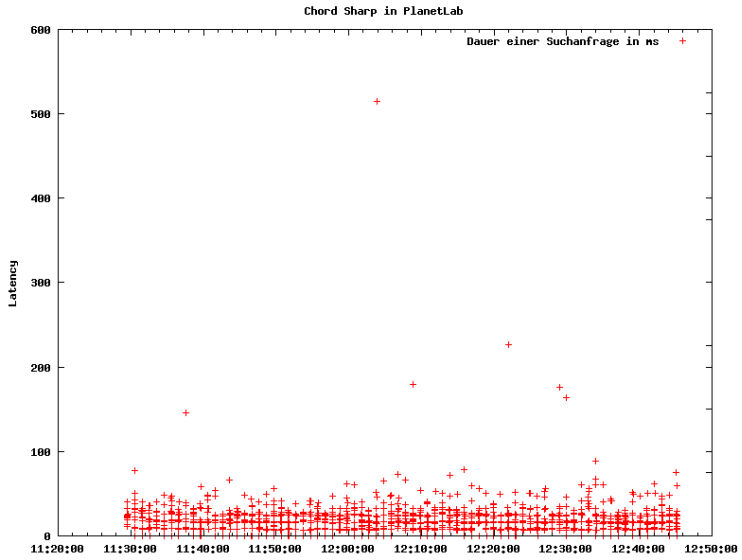

Abbildung 6.1: Grafik mit allen Daten, inklusive Ausreissern

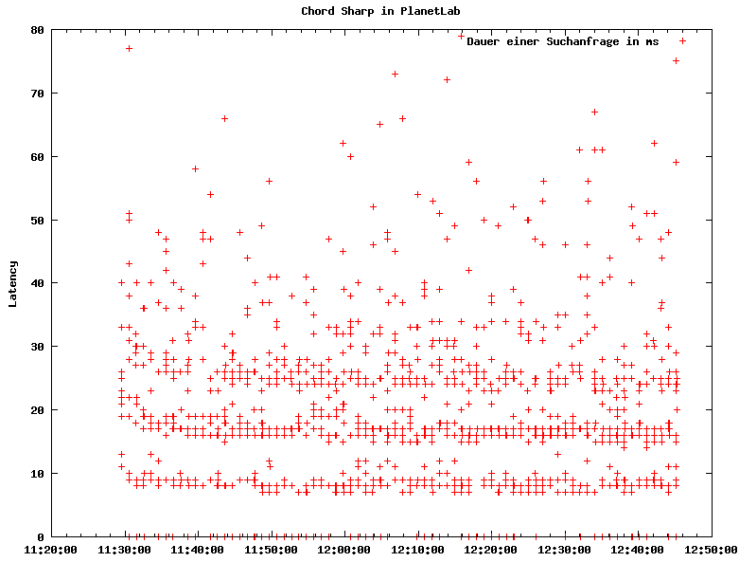

Abbildung 6.2: Grafik ohne das am stärksten abweichende Perzentil

verzögert antworten, jedoch ansonsten einwandfrei funktionieren und daher nicht aus den Fingertabellen entfernt werden. In solchen Fällen kommt es vor, dass Antworten plötzlich statt wenigen Millisekunden mehrere Sekunden benötigen.

Daten für Statistiken und Auswertungen zu sammeln, besonders für die globale Sicht auf den Ring, widerspricht dem Prinzip des verteilten Algorithmus, bei dem die Peers gleichberechtigt sind. Es gibt gerade *keine* zentrale Instanz, die das gesamte Netz kennt. Für die Auswertungen musste eine solche Instanz künstlich erschaffen werden. Dafür lief während der Experimente auf einem Rechner im Zuse-Institut Berlin ein Statistikserver, der für das Sammeln der Daten zuständig war.

Die Knoten erhielten zusätzlich eine Prozedur, sodass sie alle 30 Minuten ihre lokalen Daten zu dem Statistikserver transferierten. Darin enthalten waren die Nachfolger und Finger, der Vorgänger, sowie detaillierte Informationen über Antwortzeiten und Netzlast bei Suche, Stabilisierung, Lastverteilung, Range Queries und Replikation.

Diese Daten wurden wiederum von dem Statistikserver aufbereitet und in Logfiles geschrieben. Die Aufbereitung bestand zum Beispiel darin, die Werte für bestimmte Vorgänge wie die Suche über alle Knoten aufzusummieren und einen Mittelwert zu bilden. Die Logfiles wurden dann mit Gnuplot[1] visualisiert.

Der Statistikserver realisierte noch eine weitere Funktion. Jeder Knoten, der den Ring betreten möchte, benötigt einen schon vorhandenen Knoten als Einstiegspunkt. In der Theorie wird dies vorausgesetzt und nicht näher beschrieben, da es sich um ein triviales Problem mit mehreren Lösungsansätzen handelt. Zum Beispiel könnten Serverlisten im Internet heruntergeladen werden, zusätzlich können sich Knoten in einer lokalen Datenbank merken, zu welchen Knoten sie jemals verbunden waren (oder eine geeignete Auswahl, falls es zuviele waren) und diese wieder als Einstiegspunkte benutzen.

Der Statistikserver war dieser Einstiegspunkt. Er war jedem Knoten bekannt und teilte den neuen jeweils einen zufälligen – schon im Ring vorhandenen – Knoten zu, denn in einer realen Umgebung ist anzunehmen, dass jeder Knoten einen eigenen Einstiegspunkt wählt. Wenige vereinzelte Einstiegspunkte könnten ausfallen oder angegriffen werden.

6.2 Netz Europa

6.2.1 Aufrechterhaltung der Ringstruktur

Abbildung 6.3 zeigt die detaillierte Aufteilung des Netzwerkverkehrs. Hier kann man beobachten, dass die Kosten für die Stabilisierung relativ gleichbleibend sind und nur leicht anwachsen, als der Ring seine Größe von 3 auf 33 ändert. Da die Länge der Nachfolgerliste konstant bleibt, werden diese Kosten auch bei weiterem Wachstum des Rings nicht größer werden.

Die *csStoreData* Prozedur, die nur in der ersten Hälfte des Tests zu sehen ist, wird durch die Lastverteilung ausgelöst. Anfangs befinden sich alle Keys auf einem einzigen Knoten, der diese dann auf die anderen verteilt. Nachdem die Last verteilt ist, findet nur noch selten eine Umverteilung der Keys statt, die hier nicht ins Gewicht fällt.

Um zu zeigen, dass im Vergleich zu den Benutzeranfragen die Stabilisierung einen geringfügigen Teil des Netzwerkverkehrs ausmacht, wurde eine Suche nach den im Ring vorhandenen 2000 Keys initiiert. Alle Knoten haben in diesem Zeitraum ihre Keys an den

[1] www.gnuplot.info

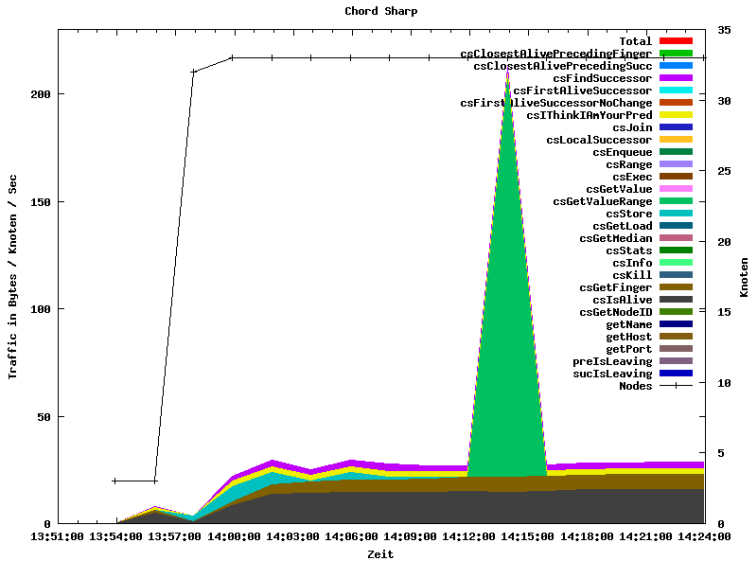

Abbildung 6.3: Netzwerkverkehr für Instandhaltung der Ringstruktur (Europa)

Zeigt den Netzwerkverkehr den ein Knoten pro Sekunde verursacht für alle Prozeduren (die RMI benutzen) einzeln. Hauptsächliche Verursacher sind *csIsAlive* (Feststellen, ob ein Finger oder einer der Nachfolger noch erreichbar ist), *csFindSuccessor* und *csIThinkIAmYourPred* für die Ringstabiliserung, sowie *csGetFinger* für die Finger. Die große Spitze *csGetValueRange* ist ein vom Benutzer initiierter Vorgang, der alle Keys anfragt und damit jeden Knoten erreicht.

Initiator der Suche geschickt, was insgesamt über einige Minuten einen Netzwerkverkehr von einigen Kilobytes verursachte. Die Keys und Values waren in diesem Fall kurze Zeichenketten.

Abbildung 6.4 zeigt nochmal die Aufteilung des Netzwerkverkehrs. Dabei wurden die einzelnen Prozeduren zu logischen Gruppen zusammengefasst, um zu verdeutlichen welche Vorgänge (Lastverteilung, Benutzeranfragen, Ringstabilisierung) den größten Netzwerkverkehr verursachen.

6.2.2 Suchanfragen

Abbildung 6.5 zeigt die Auswertung von PlanetLab-Daten, die über den Zeitraum einiger Stunden gesammelt wurden. Die Anzahl der Knoten (türkis) schwankt, da die Knoten, die auf den PlanetLab Rechnern laufen, von einer Zufallsvariable abhängig kommen und gehen. Das bedeutet, dass sie entweder den Ring verlassen oder einen weiteren Knoten erzeugen, der dem Ring hinzugefügt wird. Zwar können dadurch mehrere Knoten auf einem Rechner laufen (oder auch keiner), dies behindert jedoch die Messungen nicht. Über den Zeitraum wurden insgesamt etwa 150 Knoten in den Ring aufgenommen (dunkelblau).

Die Hops (grün) sind auf der rechten y-Achse abzulesen und schwanken zwischen 0 und

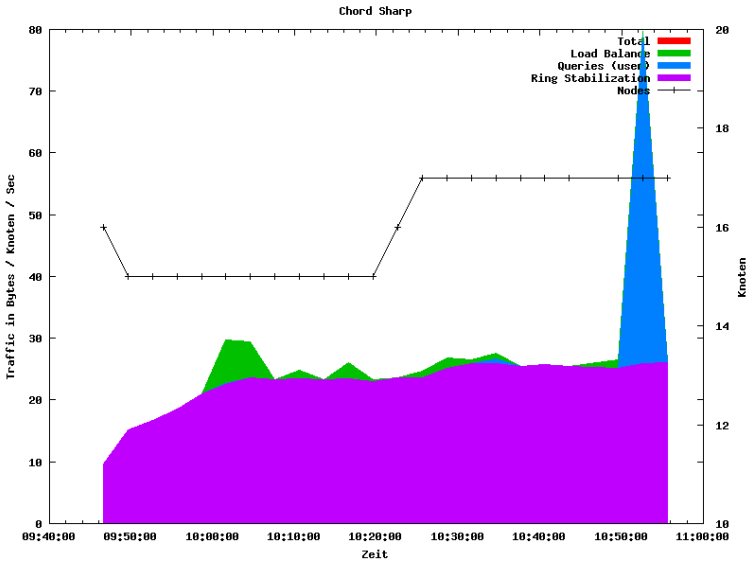

Abbildung 6.4: Netzwerkverkehr (gruppiert) für Instandhaltung der Ringstruktur (Europa)

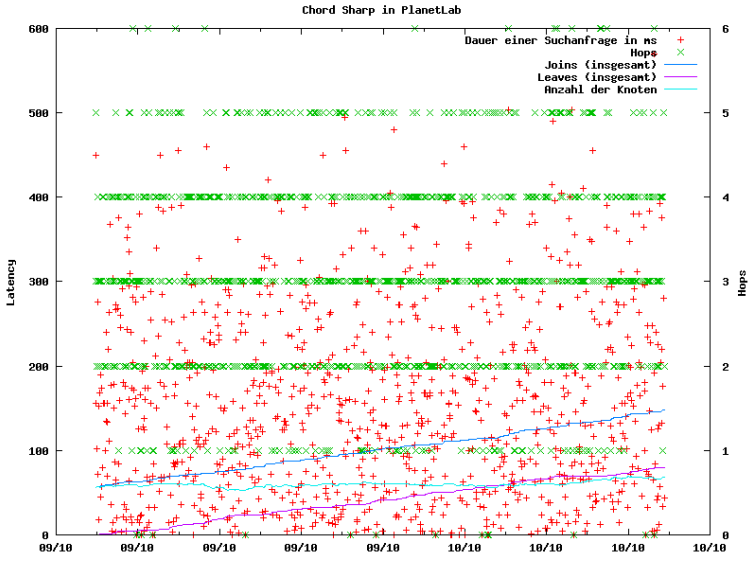

Abbildung 6.5: Antwortzeiten und Hops (europaweit)

6. Bei etwa 64 Knoten sollte der längste Weg nicht mehr als 6 Hops benötigen, da die Finger in Chord# so gewählt sind, das bei jedem Hop mindestens die Hälfte des Weges zurückgelegt wird und $log_2(64) = 6$. 7 Hops wäre ebenfalls möglich gewesen, für den Fall

dass ein neuer Knoten noch keine guten Finger hat und deswegen die Suche nicht ideal weiterleitet. Dieser Fall trat aber hier nicht auf oder verschwand in dem 1% Perzentil, das nicht ausgewertet wurde.

Man sieht, dass die durchschnittliche Anzahl an Hops (3) am häufigsten auftrat, während Extremwerte eher selten vorkamen. 0 Hops bedeutet, der suchende Knoten hat den Key selbst. 6 Hops kann zum Beispiel vorkommen, wenn der Vorgänger des suchenden Knoten den Key hat und deswegen die Suche über den ganzen Ring gehen muss. Durch die zufällig gewählten Keys bei der Suche ergibt sich für die Anzahl der Hops eine Gauß-Verteilung.

Die Antwortzeiten für eine Suche (rot) sind ebenfalls stärker im mittleren Bereich (100–200 Millisekunden) vertreten. Die besonders kleinen Werte sind Suchanfragen die direkt von dem Knoten selbst beantwortet werden konnten und daher kein entfernter Prozeduraufruf nötig war.

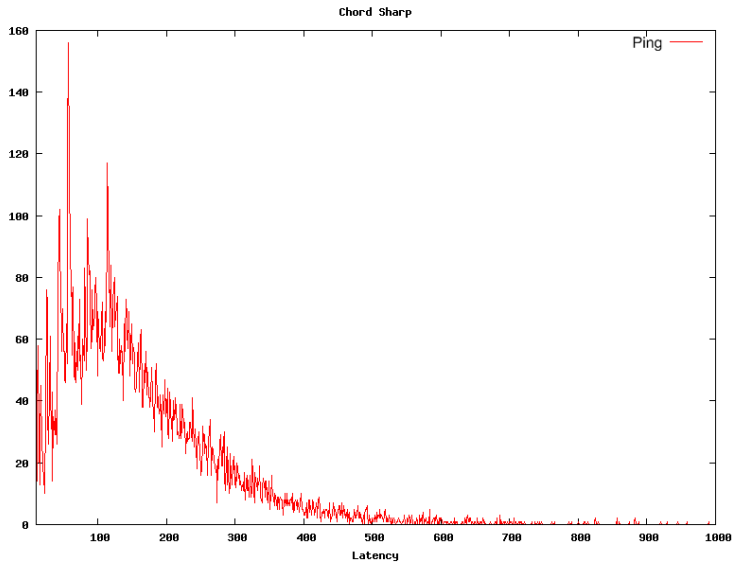

Abbildung 6.6: Verteilung der Antwortzeiten (absolut)

16.000 Suchanfragen und deren Verteilung mit jeweiliger Anzahl (y-Achse) und dazugehöriger Antwortzeit (x-Achse)

Abbildung 6.6 zeigt die Verteilung der Antwortzeiten für die einzelnen Suchanfragen. Insgesamt gab es in dem Experiment etwa 16.000 Suchanfragen. Hier ist die Antwortzeit aufgetragen gegen die Anzahl der Suchanfragen, die diese Zeit benötigt haben. Man kann erkennen, dass die meisten Suchanfragen zwischen 0 und 200 Millisekunden benötigt haben und ab etwa 700 Millisekunden nur noch sehr wenige Suchanfragen auftauchen. Eine andere Ansicht auf diese Daten zeigt Abbildung 6.7. Dort kann man ablesen wieviel Prozent der Suchanfragen innerhalb der Zeit beantwortet wurden. Zum Beispiel wurden 50% der Suchanfragen in weniger als 120 Millisekunden beantwortet. 90% konnten in 300 Millisekunden beantwortet werden und nur etwa 1% der Suchanfragen benötigte länger

als 1 Sekunde.

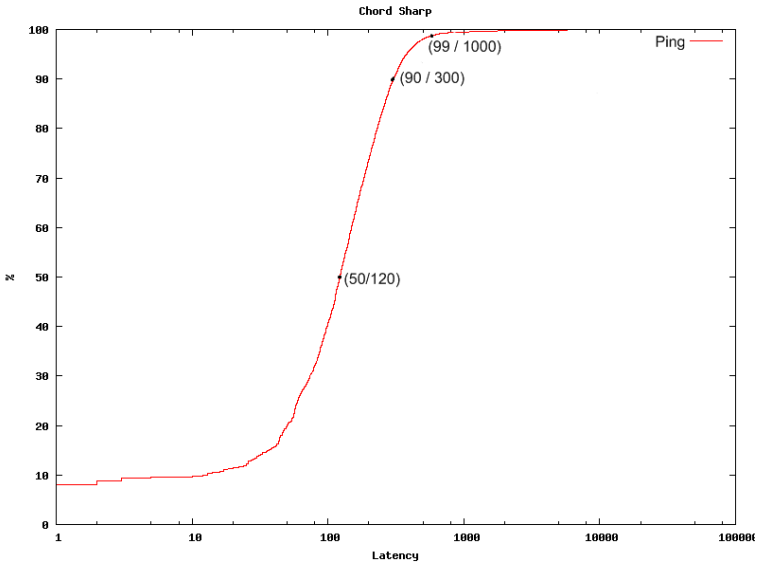

Abbildung 6.7: Verteilung der Antwortzeiten

16.000 Suchanfragen und deren prozentuale Aufteilung auf die Antwortzeiten. mit jeweiliger Anzahl (y-Achse) mit dazugehöriger Antwortzeit (x-Achse)

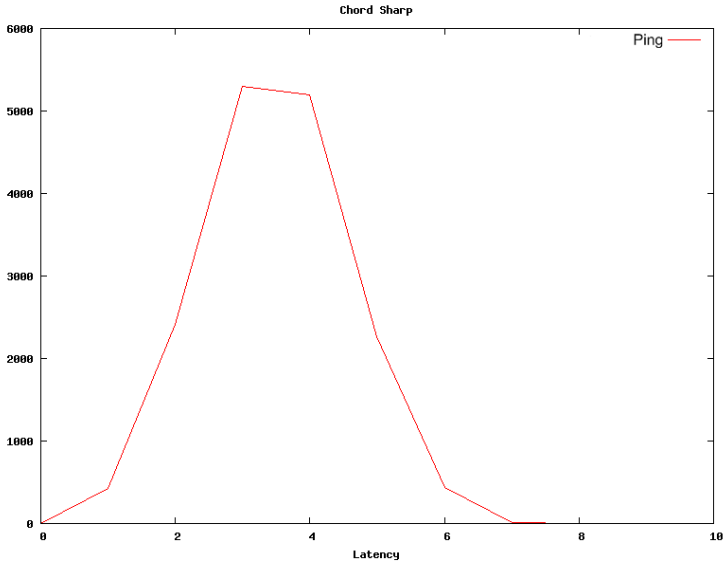

Abbildung 6.8: Verteilung der Hops

Analog zu den Antwortzeiten zeigt Abbildung 6.8 eine Verteilung der benötigten Hops. Hier sieht man nochmal deutlich die Gauß-Verteilung um den Erwartungswert.

6.2.3 Lastverteilung

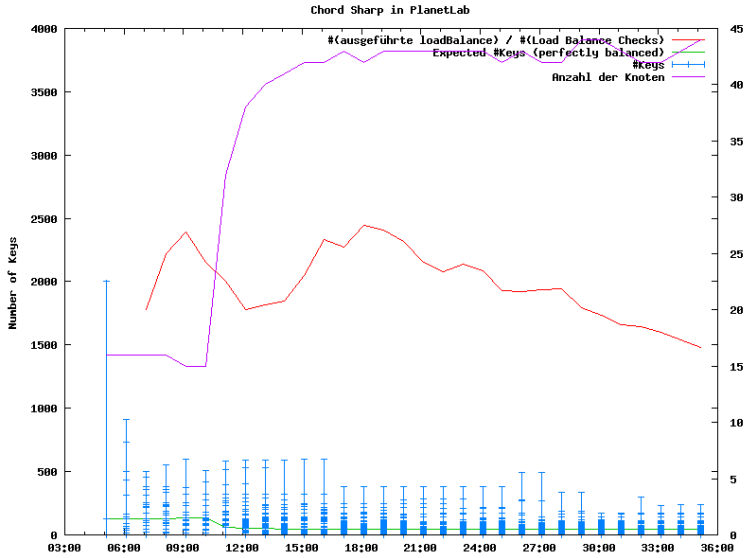

Abbildung 6.9: Lastverteilung über die Zeit (europaweit)

Abbildung 6.9 zeigt die Auswertung der Lastverteilung über einen längeren Zeitraum. Zum Anfang des Experiments waren alle 2000 Keys auf einem einzelnen Knoten, alle anderen Knoten haben 0 Keys (blaue Linie). Über die Zeit verteilten sich die Keys auf die Knoten (lila). Die grüne Linie ist der für eine perfekte Lastverteilung angestrebte Mittelwert, also der Quotient aus Knoten und Keys. Dieser wird in der Praxis nie erreicht, da die Knoten die Lastverteilung erst ab einem gewissen Schwellwert initiieren.

Die rote Linie gibt an, wie oft eine Lastverteilung stattfinden musste. Jeder Knoten prüft in regelmäßigen Abständen, ob eine Lastverteilung nötig ist und initiiert diese gegebenenfalls. Die rote Linie ist der Quotient und die Tendenz ist fallend, da die Keys nach einer Weile verteilt sind und damit keine weitere Lastverteilung mehr nötig ist.

Abbildung 6.10 zeigt ein weiteres Experiment zur Lastverteilung. Diesmal waren die Schwankungen in der Anzahl der Knoten (türkis) wesentlich höher. Durch viele Knoten, die den Ring betreten oder ihn verlassen, wird die Lastverteilung zwar verlangsamt, tendiert jedoch trotzdem zu dem gleichen Ergebnis. Man sieht in dieser Grafik jeweils den Knoten mit den meisten (lila) beziehungsweise wenigsten (blau) Keys und deren Abweichung zum erwarteten Mittelwert. Anfangs befinden sich wieder alle 2000 Keys auf einem Knoten. Durch die Schwankungen kann es zwar passieren, dass der schwerste Knoten kurzfristig wieder zunimmt, die Tendenz geht jedoch trotzdem zu dem angestrebten Mittelwert.

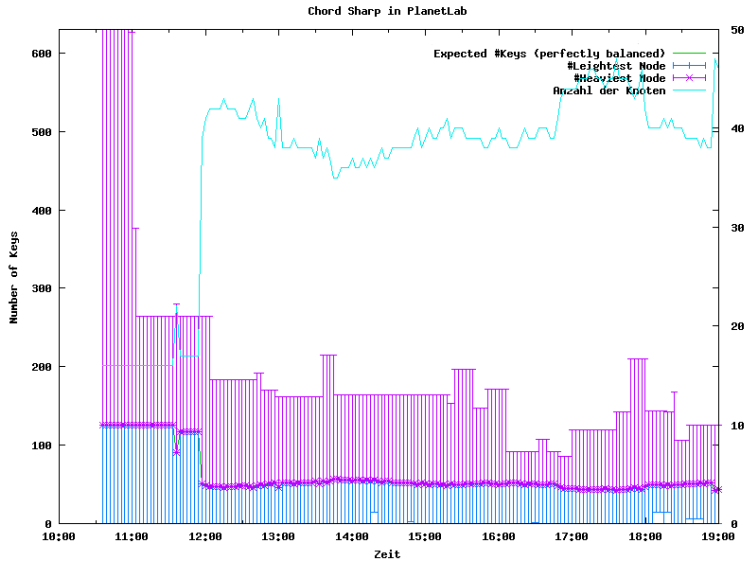

Abbildung 6.10: Lastverteilung bei instabilem Netz (europaweit)

6.3 Netz Welt

6.3.1 Aufrechterhaltung der Ringstruktur

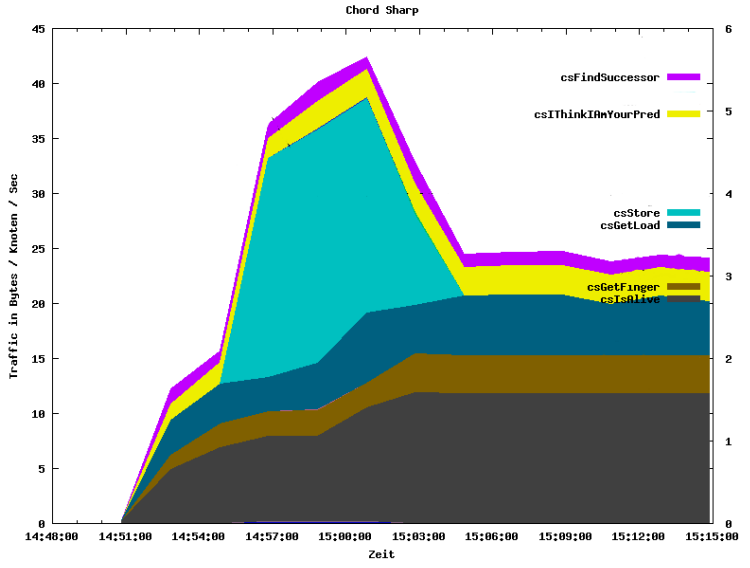

Abbildung 6.11: Netzwerkverkehr für Instandhaltung der Ringstruktur (weltweit)

Bei den Kosten für die Instandhaltung 6.11 verhält es sich im weltweiten Netz ähnlich wie im europäischen Netz. Es werden ebenfalls von jedem Knoten etwa 25 Bytes/Sekunde für die Ringstabilisierung benötigt. Das Ansteigen des Netzwerkverkehrs am Anfang des Tests ist dadurch zu erklären, dass die Knoten zunächst nur ihren Nachfolger kennen und deswegen nicht viel Stabilisierung und keine Aktualisierung der Finger vornehmen. Je mehr sich die Knoten untereinander verbinden, desto mehr steigt der Netzwerkverkehr bis zu seinem Durchschnitt von 25 Bytes/Sekunde an. Zwischendurch werden wegen der Lastverteilung kurzfristig viele Bytes versendet, der Aufwand dafür geht jedoch wieder gegen 0, da dieser Ring nur aus stabilen Knoten besteht und die Last irgendwann verteilt ist.

6.3.2 Suchanfragen

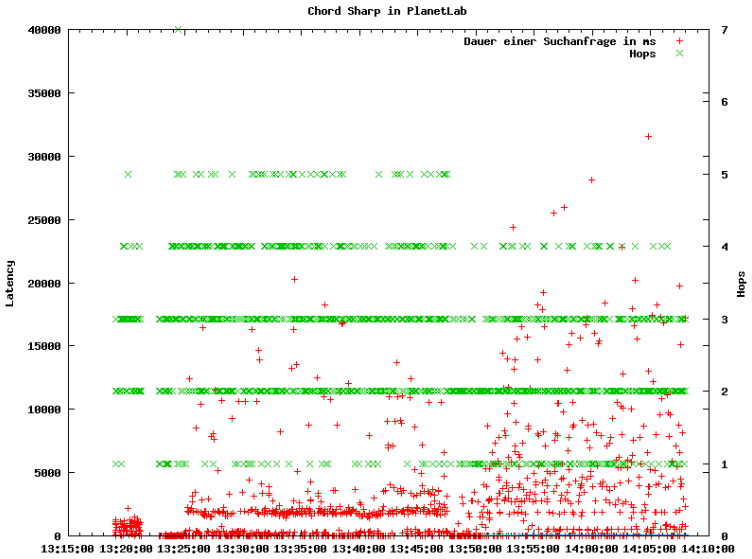

Abbildung 6.12: Antwortzeiten und Hops (weltweit)

Abbildung 6.12 zeigt die Auswertung der Daten zu den Suchanfragen im weltweiten Netz (analog zum europäischen Netz 6.5). Während die Anzahl der Hops in etwa den Erwartungen entspricht (sie liegt ebenfalls in einer Gauß-Verteilung um den Erwartungswert und damit im gleichen Bereich wie bei den Europa-Tests), kommt es bei den Antwortzeiten zu drastischen Abweichungen. Im Vergleich zu dem europäischen Netz liegen hier die Zeiten vieler Suchanfragen etwa um den Faktor 10 höher (3000–4000 Millisekunden im Vergleich zu etwa 300 Millisekunden). Eine denkbare Ursache dafür ist, dass das Underlay-Netzwerk nicht bei der Wahl der Finger beachtet wird. Das hat zur Folge, dass eine Suche mit 6 Hops eventuell mehrmals den Atlantik überqueren muss. Weiterhin ist es einfach wahrscheinlicher, dass Knoten aus Ländern wie Brasilien, Ungarn oder Israel ausfallen oder nicht antworten. In diesem Fall wird erst nach einem Timeout ein anderer

Knoten kontaktiert, was zur massiven Verschlechterung der Ergebnisse führt. Erweiterungen des *Chord#* Algorithmus, die das Underlay-Netzwerk beachten, sind deswegen in einem weltweiten Netz besonders vielversprechend, da dadurch die Knoten aus dem weltweiten Netz ein Teilnetz bilden, was den Ergebnissen her eher denen des europäischen Netzes 6.5 gleicht.

6.3.3 Lastverteilung

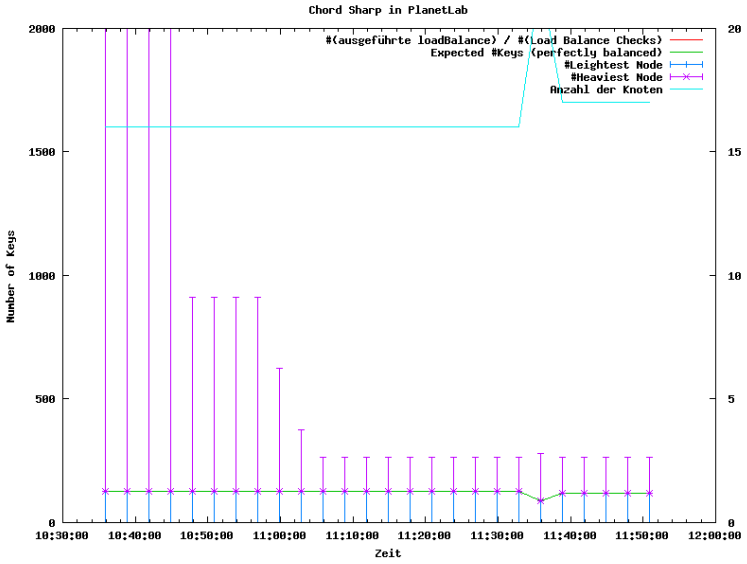

Abbildung 6.13: Lastverteilung über die Zeit (weltweit)

Abbildung 6.13 zeigt die Auswertung von den Daten zur Lastverteilung im weltweiten Netz. Auch im weltweiten Netz funktioniert die Lastverteilung und der schwerste Knoten konvergiert gegen den angestrebten Mittelwert. Die Lastverteilung dauert jedoch insgesamt länger (etwa 35 Minuten statt 12 Minuten im europäischen Netz).

7 Zusammenfassung

> Wir Menschen verwenden unsere höchsten Kräfte zu albernen Resultaten.
>
> *(Conrad Ferdinand Meyer (1825–98), schweizer. Dichter)*

7.1 Realisierung

Für diese Arbeit wurde der Chord# Algorithmus in Java implementiert (ca. 6500 Zeilen Code, 1500 für den Algorithmus und 4000 für RMI, Logging, GUI etc.). Die Kommunikation zwischen den Knoten wurde mit Remote Method Invocation (RMI) realisiert. Die verteilte Struktur stellte sich an vielen Stellen als problematisch heraus. Ein verteilter Algorithmus verhält sich in realen Netzwerken stets anders als simuliert auf einem Desktop-Rechner. Dies liegt an verzögerten Laufzeiten, heterogenen Rechnersystemen, Ausfall von Rechnern und Ausfall von Zwischensystemen. Außerdem ist das Lokalisieren von Fehlern sehr aufwändig. Dafür müssen Logfiles ausgewertet werden und immer wieder neue Tests gemacht werden.

Die Werte für die Grafiken in Kapitel 6 konnten ebenfalls nur durch Logfiles gewonnen werden. Dafür mussten die Daten der einzelnen Knoten wieder zusammengeführt werden, um ein Gesamtbild über den Ring zu erhalten. Dies widerspricht eigentlich der P2P Architektur, in der es keine Server und Clients gibt, sondern nur gleichgestellte Peers.

7.2 Ergebnisse

Es wurde mit Hilfe der Tests in PlanetLab gezeigt, dass *Chord#* – bezogen auf die Suche – genauso gute Ergebnisse erzielt wie *Chord*, dabei jedoch auf die Hashfunktion verzichtet. Die Laufzeit für die Suche liegt bei beiden Algorithmen in $O(log(n))$. Die empirischen Werte aus [15] wurden bestätigt.

Auch die Instandhaltungskosten bleiben im erwarteten Rahmen. Besonders nachdem der Cache eingeführt wurde und geeignete Werte für die einzelnen Parameter (Nachfolgerliste, Timer für die einzelnen Prozeduren) gefunden waren, benötigte ein Knoten nur etwa 25 Bytes/Sekunde für die Instandhaltung des Rings. [5] [7] [6]

Weiterhin wurde gezeigt, dass sich die Lastverteilung nicht sehr stark bemerkbar macht in einer fortgeschrittenen Phase des Algorithmus. Beim Aufbau des Rings wird noch verhältnismäßig oft Lastverteilung betrieben, jedoch ändert sich später nicht mehr so viel an der Menge der Knoten (prozentual gesehen), dass die Lastverteilung stark ins Gewicht fallen würde.

Es wurde festgestellt, dass für ein produktives System noch einige Hürden zu überwinden wären. Zum einen muss noch näher erforscht werden, wie sich die unterschiedlichen Verfahren der Bereichsabfragen auf die Netzlast und Antwortzeiten auswirken. Außerdem muss das Problem mit den byzantinischen (bösartigen) Knoten behandelt werden. Der Lösungsansatz, der für *Chord* existiert, ist für *Chord#* nicht anwendbar, da er darauf basiert, dass die Knoten eine *zufällige Node-ID* erhalten und sich selbst nicht umbenennen können. Gerade dies ist aber für die Lastverteilung notwendig. Also muss ein Verfahren entwickelt werden, um bösartige Knoten zu erkennen und möglichst unschädlich zu machen.

Ein weiterer Aspekt, der im *Chord#*-Algorithmus verbessert werden sollte, ist die Beachtung des Underlay-Netzwerks. Hier liegt ein großes Potential (besonders im weltweiten Netz, siehe Abschnitt 6.3.2) an Verbesserungen von Antwortzeiten. Die Standardverfahren zur Ersetzung der Finger sollten näher geprüft und getestet werden.

Auch muss beachtet werden, dass hier der Einstiegspunkt in den Ring sehr trivial gelöst wurde. Der Statistikserver, der die Einstiegspunkte vergab, wäre in einer produktiven Anwendung ein idealer Angriffspunkt, um den gesamten Ring zu zerstören, denn ohne ihn könnten keine neuen Knoten den Ring betreten. Es ist also wichtig, eine Methode zu implementieren, die dieses Problem nicht hat. So sollten zumindest die Knoten ein Gedächtnis haben, mit welchen Knoten sie in letzter Zeit verbunden waren. Dies wurde zum Beispiel in dem Torrent-Netzwerk realisiert. Auch wenn der *Tracker* (entspricht etwa dem Einstiegspunkt in *Chord#* ausfällt, können Knoten, die schon einmal im Ring waren, diesen – mit Hilfe von gespeicherten Knoten – wieder betreten.

Literaturverzeichnis

[1] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[2] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[3] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden, December 2006.

[4] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43, New York, NY, USA, 2004. ACM Press.

[5] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and M. Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proc. of the 3rd IPTPS*, Feb 2004.

[6] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of dht routing tables. In *Proc. of the 2nd NSDI*, May 2005.

[7] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, and Thomer M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proc. of the 24th Infocom*, March 2005.

[8] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric, 2002.

[9] Larry Peterson. Planetlab: Evolution vs intelligent design in global network infrastructure, 2006.

[10] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[11] Jordan Ritter. Why gnutella can't scale. no, really. Technical report.

[12] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[13] Amos Fiat Jared Saia and Maxwell Young. Making chord robust to byzantine attack.

[14] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Chord$^{\#}$: Structured overlay network for non-uniform load distribution. Technical report, Konrad-Zuse-Zentrum für Informationstechnik Berlin, August 2005.

[15] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. In *Proceedings of the Sixth Workshop on Global and Peer-to-Peer Computing (GP2PC'06)*, May 2006.

[16] E. Sit, F. Dabek, and J. Robertson. Usenetdht: A low overhead usenet server, 2004.

[17] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[18] Jeremy Stribling, Isaac G. Councill, Jinyang Li, M. Frans Kaashoek, David R. Karger, Robert Morris, and Scott Shenker. Overcite: A cooperative digital research library. In *International Workshop on Peer-to-Peer Systems*, 2005.

[19] Thomas Zahn. *Structured Peer-to-Peer Services for Mobile Ad Hoc Networks*. PhD dissertation, Fachbereich Mathematik u. Informatik, Freie Universität Berlin, July 2006.

[20] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

Index