

Analyse und Vergleich von Peer-to-Peer basierten Algorithmen zur Lokalisierung von Ressourcen

Studienarbeit im Fach Informatik

vorgelegt von
Alexander von Gernler

Institut für Informatik, Lehrstuhl für Informatik IV:
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
Dipl.-Inf. Rüdiger Kapitza

Beginn der Arbeit: 01. Oktober 2002
Abgabedatum: 01. Juli 2003

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.

Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Alexander von Gernler

Erlangen, 01. Juli 2003

Kurzzusammenfassung

Der Trend um die Musiktauschbörsen im Internet hat nicht nur bekannte Projekte wie Napster oder Gnutella schnell wachsen lassen. Neben breiter Berichterstattung in den Medien hat er viele Einzelpersonen und Firmen dazu motiviert, ihre eigene Implementation eines für ihren Zweck „besten“ Peer-to-Peer Projektes zu erstellen. Viele unterschiedliche Zielsetzungen und eine Fülle theoretischer Ansätze generierten so im Laufe der Zeit ein auf den ersten Blick kaum überblickbares Feld an verfügbaren Anwendungen. Leider wurde durch die heftige Strapazierung des Begriffes *Peer-to-Peer* auch eine Verzerrung der Bedeutung herbeigeführt, so dass manche Internet-Tauschbörsen sich vollkommen zu Unrecht mit dem Prädikat *Peer-to-Peer* rühmen, obwohl sie für die Funktion ihres Protokolls immer noch einen oder mehrere zentrale Server benötigen.

Die Antwort der Wissenschaft auf ineffiziente Algorithmen und mit der heißen Nadel gestrickte Programme ließ nicht lange auf sich warten: Die letzten fünf Jahre haben das Interesse für Peer-to-Peer auch unter den Forschern hochkochen lassen — praktisch jede Konferenz über verteilte Systeme konnte in dieser Zeit mit mehreren Beiträgen über solche Algorithmen aufwarten.

Kein Wunder, denn die Anforderungen an derartige Verfahren sind hoch, nicht zuletzt, weil die vielen Teilnehmer an solchen Netzen ständig fluktuieren und oft von ihren Einwahlknoten aus dynamische IPs zugeteilt bekommen. Die wissenschaftlichen Ansätze konzentrierten sich auf den Entwurf von verteilten Routing-Schichten, und nicht auf Protokolle und Clients, wie die kommerziellen Entwürfe. Allen solchen Projekten ist jedoch gemeinsam, dass verteilte Ressourcen verwaltet und bei Bedarf schnell gefunden werden müssen, ganz gleich, welcher speziellen Anwendung das jeweilige Programm dient.

Diese Studienarbeit möchte eine Übersicht über wissenschaftlich relevante verteilte Algorithmen zur Peer-to-Peer basierten Lokalisierung von Ressourcen geben, da diese für jede weitere Anwendung grundlegend sind. Hierfür werden nach einer vorausgehenden Begriffsklärung zunächst die allgemeinen Eigenschaften eines dezentralen Peer-to-Peer Algorithmus herausgestellt. Anschließend folgt ein detaillierter Vergleich der fünf ausgewählten Systeme nach vorher definierten, einheitlichen Kriterien mit Bewertung der Algorithmen hinsichtlich ihrer Vor- und Nachteile, sowie ihrer Eignung für bestimmte Anwendungen.

Abstract

Not only projects like Napster or Gnutella have profited from the increasing popularity of Internet music exchange sites. Apart from regular and diversified reports in the media it has been a motivation for many people and companies to realise their own implementations of Peer-to-Peer projects „best“ suited for their individual purpose. Since most of these projects have different aims and are based on different theoretical approaches, too many applications have been created over time for all available possibilities to be taken in at a glance. Unfortunately, the rather over-exhaustive use of the term *Peer-to-Peer* has led to a slight distortion of its meaning, i. e. some of the file sharing websites are attributed with the term *Peer-to-Peer*, although they still need one or more central servers in order for their protocol to function.

Scientists were quick to respond to inefficient algorithms and programs hastily put together: the interest in P2P among researchers was such that during the last five years there was no conference on distributed systems without at least something on the subject of these algorithms.

This comes as hardly any surprise, though, since many demands put on such systems are high, not the least because the number of participants in these kinds of networks is subject to constant fluctuations and the individual participant is often allocated a dynamic IP through his/her dial-in node. The scientific approach, however, concentrates on distributed routing layers and not on protocols and clients, as do the commercial designs. All these programs, however, have in common that they need to be able to manage distributed resources and to quickly regroup those when needed, no matter what specialised application the individual program serves.

This term paper aims to give an overview over scientifically relevant distributed and fully decentralised algorithms for the localisation of resources of the moment. As a first step it gives definitions for all the specialised terms used and lists the general characteristics of a decentralised Peer-to-Peer algorithm. This is followed by a detailed comparison according to pre-defined and standardised criteria of five selected systems, which are judged with regard to general advantages and disadvantages as well as their suitability for certain uses.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Aufgabenstellung	2
1.3	Struktur	2
2	Herangehensweise	3
2.1	Begriffsklärung	3
2.2	Ergebnisdarstellung und -diskussion	7
3	Allgemeine Betrachtung	10
3.1	Orthogonale Dimensionen dezentraler P2P-Algorithmen	10
3.2	Systemunabhängige Gemeinsamkeiten	12
4	Die Algorithmen	18
4.1	Chord und DHash	18
4.2	CAN	31
4.3	PLAXTONs Algorithmus	40
4.4	Pastry	46
4.5	Tapestry	53
4.6	Kademlia	60
5	Fazit	68
5.1	Betrachtung von Peer-to-Peer-Systemen als metrische Räume	68
5.2	Problem des ersten Kontakts ungelöst	68
5.3	Auswirkungen und Anforderungen an die Hashfunktionen	70
5.4	Vorteile redundanter Verbindungen	72
5.5	Replikation, Lastverteilung und Anfrageoptimierung	72
5.6	Die Systeme in der Übersicht	73
6	Zusammenfassung und Ausblick	76
A	Anwendungen	77
A.1	Filesysteme	77
A.2	Verteilte Nameserver	79
A.3	Multicast-Dienste	79
A.4	Dateiarchive	80
A.5	Kooperative Webcaches	80

B Ausser Konkurrenz: Pseudo Peer-to-Peer	81
B.1 Verteiltes Rechnen	81
B.2 Verteilte Datenspeicherung	82
B.3 Clustercomputing	82
C Mathematischer Hintergrund	83
C.1 Hashfunktionen	83
C.2 Public-Key Kryptographie	84
C.3 Metrische Räume	85
C.4 Grundbegriffe der Graphentheorie	86
Literaturverzeichnis	89

Kapitel 1

Einleitung

1.1 Motivation

Seit der Möglichkeit, Musik und Videodaten mit geringem Platzverbrauch auf dem heimischen PC zu speichern, ist auch die Nachfrage nach einem Austausch solcher Daten extrem gestiegen. Bedient wird sie von populären Musiktauschbörsen, deren Namen derzeit durch die Medien gereicht werden: KaZaa, Morpheus, Napster [35], Gnutella [17], Audiogalaxy und andere Begriffe werden da genannt, und alle sollen, so der Volksmund, *Peer-to-Peer* sein. Obwohl das Wort in aller Munde ist, wissen die meisten Menschen nicht, was genau darunter zu verstehen ist. Auch in der Wissenschaft und der IT-Branche herrscht noch kein Konsens über die Definition des Begriffes. So kursieren die verschiedensten Auffassungen über *Peer-to-Peer*, auf deren Basis auch konkrete Software realisiert wird. Eine allgemeine Klärung der Begrifflichkeit *Peer-to-Peer* ist also von großem Nutzen für die weitere Arbeit auf diesem Gebiet.

Außerdem ist im populären Gebrauch des Worts eine Verzerrung der Bedeutung zu erkennen, die das eigentliche Problem verteilter Algorithmen charakterisiert. So ist für die meisten Anwender das *Peer-to-Peer* Programm eine Anwendung auf ihrer lokalen Workstation, die den o. g. Austausch von Dateien, einen Chat mit Freunden oder das verteilte Brechen eines Kryptoschlüssels mit schiefer Rechenleistung ermöglicht. Kaum ein Anwender verdeutlicht sich, dass das Realisieren von verschiedenen Anwendungen wie den oben erwähnten ein Leichtes ist, wenn die verwendeten Ressourcen wie Rechenzeit, verteilte Speicherung von Daten und Kommunikation zwischen Rechnern schon bereit stehen.

Das ungleich schwierigere Problem ist aber, gerade diese Ressourcen in einem verteilten System zu realisieren und zu verwalten. Weil viele verschiedene Sichtweisen auf die *Peer-to-Peer* Problematik existieren, gibt es auch hierfür diverse Ansätze. Hinzu kommt noch, dass die reale Situation im Internet einen hohen Grad an Fluktuation der Teilnehmer mit einschließt. Kommt ein Teilnehmer nach seiner Abwesenheit in das Netz zurück, so hat er auch oft eine andere IP-Adresse¹, da viele Internetprovider diese Adressen dynamisch zuteilen.

Weil es sich hier um harte Probleme handelt, sind *Peer-to-Peer* basierte Algorithmen auch in das Interesse der Forschung gerückt. Von Seiten der Wissenschaft sind in den letzten Jahren eine Vielzahl an Projekten gestartet worden, um effiziente Systeme für die verteilte Lokalisierung von Ressourcen zu entwickeln. Die Programme von Konferenzen, die sich mit verteilten Systemen beschäftigen, dokumentieren diesen Trend. Im Gegensatz zu ihren kommerziellen Gegenstücken, die Routing und Application in einem Stück Software vereinen, konzentrieren sich die wissen-

¹IP für Internet Protocol, definiert in RFC 791 [40].

schaftlichen Ansätze zunächst rein auf das Realisieren einer Routing-Schicht. Anwendungen werden, wenn überhaupt, als sog. *proof of concept* nachgeschoben und modular auf Basis der Routing-Schicht realisiert. Ein Blick in einschlägige Literatursammlungen und Veröffentlichungen zeigt, dass es bisher zwar eine Menge von publizierten verteilten Algorithmen hierzu gibt, aber keine Vergleiche dieser Verfahren.

Die Fragestellung ist, ob die vielen verschiedenen Systeme wirklich große Unterschiede aufweisen, oder ob allen von ihnen generische und wiederverwendbare Konzepte innewohnen.

1.2 Aufgabenstellung

Diese Studienarbeit möchte beiden in der Motivation aufgeführten Gedanken gerecht werden, und sich sowohl um eine formale Klärung des Begriffes *Peer-to-Peer* mit seinen verschiedenen Nuancen kümmern, als auch einen detaillierten Vergleich der wissenschaftlich relevanten Peer-to-Peer basierten Algorithmen zur Lokalisierung von Ressourcen ziehen. Wir konzentrieren uns bei der Analyse also bewußt auf die von verschiedenen Forschungsgruppen vorgestellten Routing-Schichten, und nicht auf die danach vergleichsweise trivial zu realisierenden Peer-to-Peer Protokolle, Schnittstellen und Anwendungen. Auch interessieren in dieser Arbeit nur die echt dezentralen Algorithmen. Aus dieser immer noch großen Menge wurden als Repräsentanten die bekanntesten, weil am öftesten in Arbeiten referenzierten Systeme zur Untersuchung herangezogen.

Um einen Vergleich sinnvoll führen zu können, werden zunächst einheitliche Kriterien aufgestellt und allgemeine Eigenschaften von Peer-to-Peer Algorithmen herausgearbeitet. Anschließend werden fünf konkrete und ein theoretischer Algorithmus anhand dieser Kriterien analysiert und bewertet. Weil jeder Algorithmus in einer eigenen Notation präsentiert wird, wurde im Zuge dieser Studienarbeit eine einheitliche mathematische Notation und uniforme Darstellung der Kernalgorithmen entwickelt, was dem Leser einen Vergleich stark erleichtert.

Zum Abschluss werden die Ergebnisse zusammen mit den Vor- und Nachteilen einzelner Verfahren präsentiert, und auf Ihre Eignung für bestimmte Typen von Anwendungen eingegangen. Eine Zusammenfassung mit Ausblick rundet die Arbeit inhaltlich ab.

1.3 Struktur

Im nächsten Kapitel werden wir den Begriff *Peer-to-Peer* klären und eine für diese Arbeit gültige Definition festlegen. Es folgen Bemerkungen zur verwendeten mathematischen Notation und zur Darstellung der Algorithmen. Den Schluß des Kapitels bilden die in der Arbeit verwendeten, einheitlichen Vergleichskriterien, nach denen die Algorithmen in Kapitel 4 analysiert und bewertet werden. Zuvor werden aber in Kapitel 3 die Gemeinsamkeiten aller Algorithmen und ihre Bedeutung für Peer-to-Peer Systeme behandelt. Diesen Gemeinsamkeiten steht ein allgemeines Modell voran, in das jeder existierende dezentrale Peer-to-Peer Algorithmus eingeordnet werden kann. Dieses Modell wird im Fazit in Kapitel 5 verwendet, um eine Ergebnistabelle als Übersicht über die untersuchten Algorithmen zu erstellen. In diesem Kapitel werden auch bestimmte, bei der Analyse aufgeworfene Gesichtspunkte von Peer-to-Peer Systemen durchdiskutiert.

Zur Abgrenzung der Thematik wird in Anhang A außerdem ein kurzer Überblick über mögliche Anwendungen gegeben. Anhang B beinhaltet häufig angetroffene Begriffe, die zwar im Umfeld der Peer-to-Peer Diskussion oft genannt werden, aber im eigentlichen Sinne keine dezentralen Algorithmen implementieren. Auch die Gründe für eine derartige Verwechslung werden kurz diskutiert.

Kapitel 2

Herangehensweise

Dieses Kapitel setzt die in der Studienarbeit verwendeten Konventionen und geht auf oft verwendete, fundamentale Begriffe ein. Weiterhin werden Kriterien aufgestellt, nach denen die später untersuchten Algorithmen bewertet werden.

2.1 Begriffsklärung

Wie bereits erwähnt, ist eine Festlegung auf die Bedeutung verschiedener Begriffe entscheidend für eine eindeutige Diskussion der Algorithmen. In jedem Abschnitt werden die möglichen Alternativen beleuchtet, die sich bei der Definition geboten haben, und anschließend eine Entscheidung getroffen und begründet.

2.1.1 Peer-to-Peer

In der Sammlung des O'Reilly Peer-to-Peer Projekts [37] wird auch auf die Definition des Begriffs *Peer-to-Peer* eingegangen [20]. Der Verfasser dieser Definition, ROSS LEE GRAHAM, benennt folgende Eigenschaften, die ein Knoten eines Peer-to-Peer Systems besitzen muss:

1. Es muss sich um einen Computer handeln, der in der Lage ist, selbst Dienste anzubieten.
2. Er verfügt über ein System der Adressierung, das unabhängig vom Internet Naming Service DNS¹ ist (vgl. logisches Netzwerk, Kapitel 3.2).
3. Er kommt mit schwankender Qualität der Verbindung zurecht.

Diese Definition ist aus mehreren Gründen ungünstig und für die Arbeit nicht verwertbar. Zum einen wird mit dieser Festlegung nicht ausgeschlossen, dass das gesamte System über zentrale Server verfügt. Zum anderen stört die konkrete Festlegung auf Techniken und Implementationen mit Wörtern wie *DNS* oder *Computer*. Darüberhinaus sind manche Teile der Definition wiederum viel zu vage. Es wird etwa nicht näher konkretisiert, was mit *schwankender Verbindungsqualität* gemeint ist.

Andere Quellen definieren Peer-to-Peer jeweils wieder auf eine andere Art. NATHAN TORKINGTON geht auf dieses Phänomen mit einem Vortrag [55] auf der *Yet Another Perl Conference 19101* [3] im Juni 2001 auf das Phänomen ein. Er sagt, dass nach dem aufkommenden „P2P Hype“ plötzlich alle Anwendungen als Peer-to-Peer bezeichnet wurden, auch wenn es nur Programme

¹Domain Name Service, definiert in RFC 1034 und 1035 [32, 33].

wie z. B. Napster [35], Seti@Home [49] und ICQ ging. Alle diese Programme arbeiten mit zentral orientierten Diensten, lassen aber eine große Zahl von Benutzern vom heimischen PC aus partizipieren. Der Journalist DAVID WEINBERGER berichtet in einem Artikel [56] für das Darwin Online Magazin über die O'Reilly Peer-to-Peer Konferenz im November 2001 [38] und stellt dabei die Frage „Is P2P Anything?“.

Schnell wird klar, dass eine exakte und eindeutige Definition, die auf alle existierenden Systeme anwendbar wäre, nicht zu finden ist. Alle Definitionen haben zwar gemeinsam, dass die Klienten in die Dienstbereitstellung miteinbezogen werden und eine Abkehr vom Konzept des zentralen Servers stattfindet, aber jedes einzelne Projekt realisiert dies wieder auf verschiedene Weise. Auch geben manche Systeme nur vor, ohne zentralen Server auszukommen, arbeiten dann intern aber doch mit einem Client-Server-Modell. Solche *Pseudo Peer-to-Peer Systeme* werden in Anhang B behandelt. Wegen der stark mehrdeutigen Verwendung des Begriffes *Peer-to-Peer* war es unbedingt nötig, eine feste Definition im Rahmen dieser Arbeit zu finden.

Funktionale Einordnung

Zwei Begriffe [20] erschienen bei der Recherche als sehr nützlich und sollen deshalb in dieser Arbeit verwendet werden²:

Hybrides Peer-to-Peer	Homogenes Peer-to-Peer
Nicht jeder Knoten erfüllt alle Aufgaben, die für ein funktionierendes Gesamtsystem nötig sind. Es findet eine Spezialisierung auf Teilaufgaben statt, vergleichbar der Arbeitsteilung in einem Ameisenstaat.	Jeder Knoten erfüllt exakt die selben Aufgaben und ist demnach problemlos durch einen anderen Knoten mit der selben Systemsicht ersetzbar.

Geschichtliche Einordnung

Der Begriff *Peer-to-Peer* hat in der Vergangenheit eine Entwicklung durchgemacht, die von den Anfängen des Internets reicht und durch die Hysterie um die Musiktaschbörsen zuletzt eine Veränderung seiner Bedeutung erfahren hat.

Vor dem Aufkommen von Filetauschbörsen war der Begriff lediglich ein Synonym für die Existenz netzwerkfähiger Computer ohne zentralen Server. In den Anfängen des Internets nahmen noch wenige Rechner an der Vernetzung teil, die fast immer mit festen IP-Adressen angebunden waren. Lange Zeit waren nur große Unix-Server am Netz beteiligt, die auf immer gleichbleibenden Adressen und Ports ihre Dienste (meist RPC-Services) anboten. Später wurde der Computer auch für die Wirtschaft in großem Stil bezahlbar und nützlich. Vor allem für kleinere Firmen empfohlen, weil kostengünstiger, wurde eine Peer-to-Peer Vernetzung und somit die Freigabe bestimmter Ressourcen für Teilnehmer im lokalen Netzwerk ermöglicht. In der Computerspielergemeinde ist das Peer-to-Peer Netzwerk heute noch Grundlage für diverse netzwerkfähige Spiele, bei denen sich die teilnehmenden Rechner ohne zentralen Server koordinieren.

Im heutigen Internet erfreut sich der Netzzugang einer so hohen Popularität, dass den meisten privaten Endkunden wegen des herrschenden Mangels im IPv4-Raum nur noch dynamisch IP-Adressen zugeteilt werden. Die Zahl der teilnehmenden Rechner ist über die letzten Jahrzehnte stark angestiegen, und auch das Nutzerverhalten hat sich weg von einer kontinuierlichen Onlinezeit

²In der ursprünglichen Definition wurde das Wort *echt* statt *homogen* verwendet. Diese Bezeichnung erschien im Vergleich mit den anderen Dimensionen jedoch überbetont, und wurde deshalb im Rahmen dieser Arbeit in *homogen* umbenannt.

zu einer Internetbenutzung auf Bedarf hin entwickelt. Somit muss man als Designer eines Peer-to-Peer Algorithmus mit einer hohen Fluktuation der Teilnehmerknoten rechnen. Heutige Peer-to-Peer Systeme haben sich mit Skalierungsproblemen und Effekten wie Latenz, Paketverlust, zufälligem Ausfall von Knoten und hoher Fluktuation zu beschäftigen.

Kennzeichen für historisches Peer-to-Peer ist also, dass zwar ein zentraler Server fehlt, aber das Routing der Teilnehmer immer noch auf Netzwerkebene stattfindet. Bei modernem Peer-to-Peer ist das Routing auf Anwendungsebene, was durch die Einführung einer Abstraktionsschicht (sog. *overlay network*) möglich wurde und seine Wurzeln in den veränderten Anforderungen an ein stabiles Peer-to-Peer System hat.

Allgemeine Peer-to-Peer Definition

Wir haben versucht, durch Beleuchtung verschiedener Ansätze den Begriff *Peer-to-Peer* einzugrenzen. Aus den erwähnten Beispielen konnte ein Schluß gezogen werden, der als Basis für diese Studienarbeit dient. Passenderweise hat eines der untersuchten Projekte bereits eine sehr prägnante Definition aufgestellt, die wir hier übernehmen. Sie stammt aus dem Arbeitspapier des Chord-Projektes [53, S. 1] (das in Abschnitt 4.1 besprochen wird):

Definition 2.1.1 (Peer-to-Peer Systeme) *Peer-to-Peer Systeme sind verteilte Systeme, die ohne zentrale Kontrolle und hierarchische Organisation auskommen. Desweiteren soll die Software, die auf jedem Knoten läuft, äquivalent in ihrer Funktionalität sein.*

Diese Definition schliesst allerdings nicht die Realisierung von hybridem Peer-to-Peer aus, da die volle Funktionalität der Software auf den Knoten zwar vorhanden sein kann, nicht aber unbedingt überall in gleichem Umfang zum Einsatz kommen muss.

2.1.2 Lokalisierung von Ressourcen

Im Fokus der Arbeit liegen nicht die vielen verschiedenen Peer-to-Peer *Anwendungen*, die heutzutage im Netz zu finden sind, sondern vielmehr die dahinter arbeitenden *Algorithmen*. Ressourcen wie Speicherplatz oder Rechenzeit sind die Grundlage einer jeden Anwendung. Jedoch ist das Auffinden von Ressourcen insbesondere durch die hohe Fluktuation an Rechnern, die am Internet und damit potentiell an einer Peer-to-Peer Anwendung teilnehmen, ein großes Problem. Insofern ist die Frage nach der Art der Lokalisierung von Ressourcen eine viel grundlegendere und ungleich schwierigere Frage als die nach einer Kategorisierung verschiedener Client-Anwendungen (vgl. [43, Kapitel 1]). Sie stellt das zentrale Problem dar, da erst nach erfolgreicher Lokalisierung die Ressourcen von der Anwendungsschicht überhaupt genutzt werden können.

Beispiel 2.1.1 (Dateinamen) *Ein zentrales Problem ist etwa die Verwaltung der Zuständigkeit für bestimmte Dateinamen unter den Knotenrechnern in einem verteilten Dateisystem. Ist diese Zuordnung erst einmal klar, fällt die Arbeit, eine Dateisystemsicht darauf zu implementieren, vergleichsweise leicht.*

Vor diesem Hintergrund fallen viele Begriffe, die man bei Suchen im Internet nach dem Stichwort *Peer to Peer* findet, nicht in den Fokus dieser Arbeit, da sie nur Client-Anwendungen ein und desselben Algorithmus darstellen.

2.1.3 Fehlerarten

Bei der Diskussion werden unterschiedliche Einflüsse von außen auf die Algorithmen betrachtet. Auf manche können die Systeme angemessen reagieren, während andere zu einer Handlungsunfähigkeit führen. Hier werden die zwei wichtigen Kategorien von Fehlern und Störungen in einem Peer-to-Peer System erläutert.

Fail-Stop

Unter Fail-Stop versteht man ein Modell, das das automatische Ausfallen eines Knotens im Fehlerfall vorsieht. Der Knoten wird einfach nicht mehr erreichbar sein, weil er nicht weiter an der Kommunikation teilnimmt. Dadurch kann er auch keine fehlerhafte Information in das System injizieren. Andere Knoten, die mit dem fehlerhaften Knoten kommuniziert haben, als der Fehler aufgetreten ist, können bei Ablauf eines Timeouts den Knoten eindeutig als ausgefallen markieren. Diese Beschreibung des Fail-Stop Modells stammt aus [7] und wird aufgrund seines utopischen Charakters dort auch gleich relativiert: Das Modell sei vereinfachend, weil sehr optimistisch. Dies ist richtig: Man denke nur an den temporären Ausfall der Netzwerkverbindung, der auf einem Knoten den Timeout ablaufen läßt. Sofort ist dieser Knoten der Annahme, sein Gegenüber wäre nach dem Fail-Stop Modell ausgefallen. Trotzdem ist die Fail-Stop Semantik eine wichtige Annahme beim Design aller in dieser Arbeit untersuchten Peer-to-Peer basierten Algorithmen: Ein zeitweiliges Ausbleiben der Antwort auf eine Anfrage wird immer mit dem Ausfall des zuständigen Knotens gleichgesetzt.

Byzantinische Fehler

Unter byzantinischen Fehlern versteht man das geplante und zielgerichtete Injizieren fehlerhafter Information in das System. Problematisch sind byzantinische Angriffe für alle kooperativ arbeitenden Systeme, d. h. für Systeme, in denen von einem korrekten Verhalten jedes Teilnehmers und einer Fail-Stop-Semantik ausgegangen wird.

Ein spezieller Typ des byzantinischen Fehlers soll wegen seiner Praxisrelevanz noch erläutert werden: Peer-to-Peer Algorithmen sind aufgrund ihrer Struktur sehr anfällig für sogenannte *Denial-of-Service*-Angriffe.

Denial-of-Service Dieser mit der Popularität des Internets erst breit bekannt gewordene Begriff übersetzt sich wörtlich mit *Dienstverweigerung*. Gemeint ist nicht, dass ein Knoten selbst anderen Knoten den Dienst verweigert, sondern dass er durch äußere Einflüsse (also z. B. einen byzantinischen Angreifer) dazu gezwungen wird, den Dienst zu verweigern. Dies kann durch jegliche Art der Überlastung des Knotens passieren, denn dann kann der Knoten keine anderen Anfragen mehr bearbeiten. Aber auch ein Ausnutzen von Softwarefehlern durch Anstoßen einer nicht vom Autor vorgesehenen Reaktion des Programms fallen unter *DoS* (so die Abkürzung).

Mögliche *DoS*-Angriffe sind etwa das Fluten des Knotens mit Netzwerkpaketen oder das gleichzeitige Öffnen einer Vielzahl von Verbindungen, aber auch das Ausnutzen fehlerbehafteter Software auf dem Knoten durch entfernte Fehlerinjektion (sog. *exploits*). Derartige Angriffe sind oft Gegenstand von Berichten in der Fachpresse und richten sich in diesen Fällen meist gegen populäre Internetangebote.

2.2 Ergebnisdarstellung und -diskussion

Dieser Abschnitt bereitet den Leser auf die Art der Präsentation der Untersuchungen und Resultate in dieser Studienarbeit vor. Es wurden Konventionen zur Algorithmdarstellung und zur mathematischen Schreibweise getroffen. Außerdem wurden Kriterien aufgestellt, nach denen die Algorithmen analysiert und beurteilt werden.

2.2.1 Einheitliche Notation

Jede untersuchte Arbeit präsentiert ihr Verfahren in einer eigenen Sprache und Darstellung, wenn überhaupt Algorithmen explizit notiert werden. Um dem Leser einen besseren Überblick zu geben, sind in dieser Studienarbeit alle betrachteten Algorithmen in einer einheitlichen Notation wiedergegeben. Bei Projekten, die ihre Konzepte nur in Textform darlegen, musste ein solcher Pseudocode überhaupt erst formuliert werden. Die Zuweisungsoperation „ \leftarrow “ ist aus KNUTHS *The Art of Computer Programming* [26] entliehen. Soweit nichts anderes erwähnt ist, wird davon ausgegangen, dass es sich bei den beschriebenen Funktionen um RPC³s handelt. Dies birgt den Vorteil, dass wir uns bei der semantischen Diskussion keine Gedanken machen müssen, ob der Algorithmus gerade lokal oder auf einem entfernten Knoten aufgerufen wird⁴.

Da alle untersuchten Arbeiten zur Gruppe der strukturierten (vgl. Abschnitt 3.1) Peer-to-Peer Algorithmen gehören, ist bei allen eine eindeutige Knotenidentifizierung vorhanden. Um auch hier dem Leser den Vergleich zu erleichtern, wird in dieser Arbeit folgende Notation verwendet, wenn nicht anders angegeben:

Notation	Beschreibung
$N \in \mathbb{N}$	Maximale Anzahl der im Algorithmus verwaltbaren Knoten
$[x] := \{0, 1, \dots, x-1\}$ $x \in \mathbb{N}$	Menge der Zahlen von 0 bis $x-1$, entliehen aus PLAXTONS Algorithmus (Abschnitt 4.3)
$\#\{a, b, \dots\} = m; m \in \mathbb{N}$	Mächtigkeit m einer Menge
$\Sigma = \{0, 1\}$	Das Binäralphabet mit den möglichen Zuständen, die ein Bit annehmen kann
$n \in \Sigma^m, m \in \mathbb{N}$	Ein Binärstring der Länge m , Interpretation als String
$n \in \mathbb{Z}_{2^m}, m \in \mathbb{N}$	Ebenfalls ein Binärstring der Länge m , Interpretation als Binärzahl
$x \in \Sigma^*$	Ein beliebig langer Binärstring. Auch Länge 0 ist erlaubt.
$n = h(x), h : \Sigma^* \rightarrow \Sigma^m$	Ergebnis n einer Hashfunktion h , angewendet auf den Binärstring x
$G = (V, E)$	Das gesamte Peer-to-Peer Netzwerk G , dargestellt als allgemeiner Graph mit Knotenmenge V und Kantenmenge E . Dieses Systemmodell ist die Oberklasse aller möglichen Systemmodelle.
$\Delta = d(a, b); a, b \in V$ $d : V \times V \rightarrow \mathbb{R}$	logische Entfernung zweier Knoten a und b in einem Systemmodell
$\Gamma = c(a, b); a, b \in V$ $c : V \times V \rightarrow \mathbb{R}$	physikalische Entfernung zweier Knoten a und b in einem Systemmodell, hervorgegangen etwa aus Netzwerklatenz oder Anzahl der Hops
$A \in \mathcal{A}$	Ein Objekt A aus der Menge \mathcal{A} aller Objekte
$id(A); id : \mathcal{A} \rightarrow \Sigma^*$	Die Identifikation des Objekts A als Binärstring

Zum genaueren Verständnis dieser Tabelle kann im Anhang C über mathematische Grundlagen nachgeschlagen werden. Die Interpretation von Binärstrings als Binärzahlen wird in dieser Stu-

³Remote-Procedure-Call, definiert in RFC 1831 [50].

⁴Die konkrete Fehlerbehandlung bei einem lokalen Aufruf läßt bei der Implementation natürlich noch mehr Handlungsmöglichkeiten offen, aber diese Betrachtung ist nicht Thema der Arbeit.

dienarbeit analog und je nach Kontext verwendet. Dies ist aber keine Nachlässigkeit, da sich die Binärstrings und die Binärzahlen gleicher Länge bijektiv aufeinander abbilden lassen. Hierzu muss man sich nur vergegenwärtigen, dass die Binärstrings abzählbar sind und sich auf ihnen folglich eine Totalordnung definieren lässt.

2.2.2 Einheitliche Vergleichskriterien

Um die verschiedenen Systeme überhaupt gegeneinander halten zu können, mussten Gesichtspunkte aufgestellt werden, unter denen die Algorithmen betrachtet werden. Diese Kriterien werden im Folgenden stichpunktartig als Fragen formuliert. Es leuchtet ein, dass nicht alle Kriterien für jedes Projekt sinnvoll anwendbar sind.

Allgemeine Einordnung

Es wird kurz auf Autoren, Herkunftsland, Geschichte und Hintergedanken eingegangen. Die Frage nach der Motivation der Autoren ermöglicht meist auch einen Schluß auf die möglichen Anwendungen des Projekts.

Theoretische Kriterien

Diese Kriterien beziehen sich direkt auf Eigenschaften der im Algorithmus realisierten Routing-schicht und machen den wesentlichen Teil der Analyse aus.

Systemmodell und Lokalisierungsalgorithmus: Welche Sichtweise auf die Organisation der Knoten liegt dem Algorithmus zugrunde? Durch welches Verfahren⁵ werden in dem Systemmodell Ressourcen gefunden? Unterscheidet sich die Suche nach Knoten sehr von der Suche nach Daten? Wie wird der Begriff der Entfernung zweier Knoten ausgedrückt und stellt dieser eine mathematisch echte Metrik dar?

Tolerierte Fehler und Redundanz: Werden Vorkehrungen getroffen, das System redundant und damit fehlerresistent zu machen? Wie viele Fehler nimmt das System hin, ohne eine Beeinträchtigung in der Dienstgüte zu bieten? Handelt es sich hierbei um Fehler, die lediglich nach *Fail-Stop-Semantik* auftreten dürfen, oder werden allgemein auch byzantinische Fehler toleriert?

Zustand und Stabilisierung: Wie und wo wird der Zustand des Gesamtsystems und der teilnehmenden Knoten gespeichert und konsistent gehalten? Welche Informationen hält insbesondere jeder Knoten lokal? Welche Algorithmen werden verwendet, um das System konsistent zu halten? Findet die Stabilisierung implizit oder explizit statt (siehe Erläuterung zu Stabilisierung in Abschnitt 3.2)?

Join/Leave: Mit dem Wissen, dass sich die Begriffe *Join* (Hinzukommen eines Knotens zu einem Peer-to-Peer System) und *Leave* (Verlassen des Systems durch einen Knoten) nur umständlich ins Deutsche übersetzen lassen, und ausserdem Standardbezeichnungen in jeder wissenschaftlichen Arbeit über Peer-to-Peer Algorithmen sind, werden sie in dieser Studienarbeit ebenso als Fachbegriffe verwendet.

⁵Bei den Beschreibungen von Lokalisierungsalgorithmen wird zur Bezeichnung von solchen Verfahren oft auch der Begriff *Zugriffsschema* verwendet.

Fragen, die sich bei diesen beiden Vorgängen stellen, sind unter anderem: Welche Methodik wird verwendet, um hinzukommende Knoten in das Gesamtsystem einzubauen und den Zustand entsprechend anzupassen? Wie wird insbesondere auf das Problem des ersten Kontakts (vgl. Kapitel 3.2) eingegangen? Welche Schritte muss ein Knoten durchführen, um das System wieder zu verlassen? Kann das Gesamtsystem davon profitieren, dass sich ein Knoten explizit abmeldet?

Verwaltungsaufwand (Komplexität): Wie wächst der Aufwand für Lokalisierung von Ressourcen, Aktualisierung von Verwaltungsdaten (Stabilisierung) und Join- und Leave-Operationen in der Zahl der teilnehmenden Klienten? Wie groß müssen die Verwaltungsdaten im Vergleich zur Gesamtzahl der Knoten sein? Wie teuer ist es, ein Objekt im System zur Verfügung zu stellen bzw. wieder daraus zu entfernen? Existieren spezifische Obergrenzen, die etwa das System unter bestimmten Bedingungen praktisch unbenutzbar machen?

Funktioniert der Basisalgorithmus erst einmal, so kann aufbauend darauf die Arbeitsweise optimiert werden. Auch derartige Erweiterungen werden, wenn vorhanden, diskutiert.

Knotenauswahl nach gewichteter Metrik: Gibt es eine Möglichkeit für den Algorithmus, von physikalischer Lokalität der Nachbarknoten zu profitieren (*Low Latency Routing*)? Falls redundante Ressourcen vorliegen: Wie entscheidet der Algorithmus, welcher Ressource aufgrund von Qualitätskriterien der Vorzug zu geben ist? Existiert hierfür eine Metrik, und welche Gedanken flossen bei ihrer Erstellung ein? Wird aufgrund gewisser Kriterien und der Gegebenheit von Redundanz auch Lastverteilung in die Knotenauswahl mit einbezogen?

Gesamteindruck

Für welche Anwendungen ist der Algorithmus als grundlegende Schicht geeignet (z. B. Filesystem, verteilter Nameserver, Multicast-Dienst, Dateiarchiv, anonymes Netzwerk, Instant Messaging, Chat, Newsservice)? Welche Vor- oder Nachteile hat der Algorithmus insgesamt? Wie sieht die Zukunft und die Weiterentwicklung des Projekts aus?

Kapitel 3

Allgemeine Betrachtung

In diesem Kapitel werden Kategorien aufgestellt, nach denen man Peer-to-Peer Algorithmen klassifizieren kann. Leicht auszumachende Gemeinsamkeiten aller untersuchten Algorithmen sind hier ebenso aufgeführt wie allgemeine Peer-to-Peer Konzepte. Speziellere Betrachtungen, die aus der Analyse und dem Vergleich der konkreten Algorithmen resultieren, sowie Fakten, die nicht offensichtlich auffallen, sind zusammen mit einer Gesamtübersicht über die untersuchten Systeme in Kapitel 5 zu finden.

3.1 Orthogonale Dimensionen dezentraler P2P-Algorithmen

Bei der Untersuchung der verschiedenen Algorithmen fiel ein Schema auf, nach dem es möglich ist, jeden dezentralen Peer-to-Peer basierten Algorithmus zu kategorisieren. Hierbei wurden drei voneinander unabhängige Dimensionen (vgl. Abbildung 3.1.1) deutlich, die im Folgenden als Begriffe definiert werden sollen. Alle untersuchten Algorithmen realisieren eine verteilte Hashfunktion. Die-

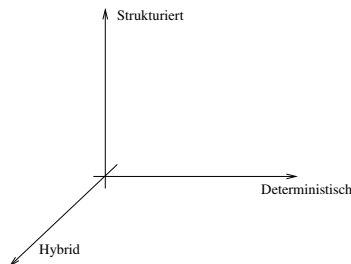


Abbildung 3.1.1: Orthogonalität der Dimensionen eines Peer-to-Peer Algorithmus: **Ordnung** (strukturiert/unstrukturiert), **Berechenbarkeit** (deterministisch/probabilistisch) und **Organisation** (hybrid/homogen)

se liefern für jede Eingabe ein deterministisches und reproduzierbares Ergebnis. Deshalb sind auch Algorithmen, die sich bei Ihrer Knotensuche auf eine solche Funktion stützen, in der Lage, jeden Knoten im System anhand seiner eindeutigen Identifikation zu finden. Wir reden in der weiteren Arbeit von der Dimension der *Ordnung* eines Peer-to-Peer Systems, die die beiden Extremwerte der strukturierten und unstrukturierten Algorithmen beinhaltet.

Definition 3.1.1 (Strukturierte P2P-Algorithmen) *Ist ein Peer-to-Peer Algorithmus strukturiert, so verhalten sich seine Knoten konform zu einem Systemmodell, das 1. jedem Knoten eine eindeu-*

tige Identifikation zuweist, 2. eine Ordnung¹ auf allen Knoten definiert und 3. jedem Knoten die Möglichkeit einräumt, in einem konsistenten Zustand des Peer-to-Peer Systems die Frage nach einer bestimmten Identifikation eindeutig mit Existenz oder Nichtexistenz des zugehörigen Knotens zu beantworten, indem er in Kooperation mit dem Rest des Systems die Lokalisierungsprimitive des gegebenen Systemmodells benutzt. Fehlt einem Peer-to-Peer Algorithmus diese Eigenschaft, so wird er als unstrukturiert bezeichnet.

Insbesondere erfüllen verteilte Hashtabellen die Voraussetzungen, um nach Definition 3.1.1 als strukturierte Peer-to-Peer Algorithmen zu gelten. Vieles deutet darauf hin, dass auch im Allgemeinen das Wissen jedes Knotens um eine vorher vereinbarte Zuordnungsfunktion für die Funktion eines dezentralen strukturierten Netzes unerlässlich ist. PLAXTON zitiert in seiner Arbeit [39] selbst einige andere Werke, die sich mit Spezialfällen des Zugriffs auf verteilte Objekte beschäftigen. Alle verwenden eine Hashfunktion. Liegt ein strukturiertes Netz vor, impliziert dies also auch die Möglichkeit für jeden Knoten, durch Anfragen an das Netz den kompletten Inhalt des Systems abzufragen (potenziell vollständige Sicht). Je nach System können dem Algorithmus unterschiedliche Strukturen als Systemmodelle zugrundeliegen. Einige Beispiele sind etwa der Baum als Spezialfall des allgemeinen Graphen, oder der eindimensionale modulare Ring als Spezialfall des d -Torus.

Eine andere Dimension besteht im Verhalten jedes einzelnen Knotens. Diese *Berechenbarkeit* kann sich entweder in deterministischen oder probabilistischen Algorithmen auszeichnen:

Definition 3.1.2 (deterministische P2P-Algorithmen) *Ein Peer-to-Peer Algorithmus ist deterministisch, wenn das Verhalten jedes Knotens im Gesamtsystem abhängig von seinem lokalen Zustand und der Eingabe von Information vorhersagbar und reproduzierbar ist. Erfüllt ein Algorithmus diese Voraussetzung nicht, so nennt man ihn probabilistisch.*

Während bei deterministischen Algorithmen im gleichen Zustand eine Anfrage immer den selben Weg durch das Netz nehmen wird und deshalb auch immer wieder an den selben fehlerhaften Knoten Probleme haben kann, umgeht ein probabilistischer Algorithmus einen defekten Knoten mit einer gewissen Wahrscheinlichkeit, und ist damit im Fehlerfall toleranter als ein deterministischer. Allerdings gibt man mit der probabilistischen Nutzung von Nachbarknoten auch teilweise den Vorteil einer physikalischen Lokalität auf, wenn man statt eines nahe gelegenen Knotens einen anderen, zufällig gewählten anspricht.

Bereits in der Einleitung wurde die dritte Dimension, die *Organisation* eines Peer-to-Peer Systems, erwähnt. Hiermit wird zwischen hybriden und homogenen Peer-to-Peer Algorithmen unterschieden.

Definition 3.1.3 (Hybride P2P-Algorithmen) *Ein hybrider Peer-to-Peer Algorithmus liegt dann vor, wenn nicht jeder Knoten im System die gleichen Aufgaben wahrnimmt, sondern eine Spezialisierung einzelner Knoten auf eine Teilmenge der Funktionalität stattfindet. Nimmt jeder Knoten im System dagegen exakt die gleichen Aufgaben wahr, spricht man von einem homogenen Peer-to-Peer System.*

¹Wir müssen uns hier auf eine Ordnung beschränken, da die Existenz einer Totalordnung vom Systemmodell abhängig ist: In einem eindimensionalen Ring ist ohne weiteres eine Totalordnung definierbar, dagegen kann es bei einem Baum oder einem d -Torus mehrere Stellen geben, die in der Ordnung als gleichwertig anzusehen sind (etwa die Knoten selber Höhe in den Unterbäumen eines Baumes).

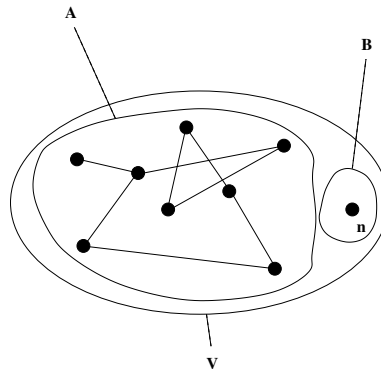


Abbildung 3.2.2: Veranschaulichung des Erstkontakt-Problems als Venn-Diagramm: Der neu hinzukommende Knoten n und die restlichen, schon ins System eingebundenen Knoten in der Menge A wissen gegenseitig nicht von ihrer Existenz.

3.2 Systemunabhängige Gemeinsamkeiten

Bestimmte Konzepte sind den Peer-to-Peer Algorithmen so inhärent, dass eine explizite Erwähnung bei jedem untersuchten Algorithmus redundant wäre. Stattdessen wird in diesem Abschnitt auf die allgemeinen Implikationen eines Peer-to-Peer Algorithmus eingegangen.

Das Problem des ersten Kontakts

Ein systemimmanentes Problem aller echt dezentralen Algorithmen ist der *erste Kontakt*. Damit ist gemeint, dass ein gerade neu gestarteter Knoten, der einem bereits bestehenden Peer-to-Peer-System beitreten will, keine Information darüber besitzt, welche Knoten dem System bereits angehören. Diese Information wird aber für den Beitritt zum verteilten System benötigt. Das Problem kann *prinzipiell nicht* innerhalb des Peer-to-Peer-Systems, d. h. ohne externe Information über andere Knoten gelöst werden. Eine Veranschaulichung ist in Abbildung 3.2.2 gegeben. Das grundlegende Problem ist, dass der Gesamtgraph mit der Knotenmenge V wegen des Hinzukommens des Knotens n ein *nicht zusammenhängender* ist. Das Knüpfen einer Kante, das diesen Zustand behebt, ist gleichwertig mit dem Injizieren externer Information in das System.

Allerdings existieren verschiedene praktische Ansätze, wie man dem hinzukommenden Knoten durch externe Information zu einem ersten Kontakt verhelfen kann (siehe Abschnitt 5.2).

Join und Leave

Ist das Problem des ersten Kontaktes gelöst, hat der neu hinzukommende Knoten eine minimale Sicht auf das System. Nun muss dafür gesorgt werden, dass das System den neuen Knoten als Teilnehmer integriert und mit in den Zustand einbindet. Allgemein sind hierfür folgende Aufgaben zu erledigen, die je nach System variieren:

1. Initialisierung des lokalen Zustands im neu hinzugekommenen Knoten, damit eine Sicht auf das System erlangt wird und eine weitere Teilnahme überhaupt möglich wird.
2. Benachrichtigung aller neuen Nachbarn im System von der eigenen Präsenz, damit der Knoten in das Routing eingebunden werden kann.

3. Optional Benachrichtigung höherer Softwareschichten über eigene Position und Nachbarn im System, weil sich dadurch auf höheren Ebenen Abhängigkeiten und Zuständigkeiten ergeben können (vgl. Chord und DHash in Abschnitt 4.1).

Ein Peer-to-Peer System muss damit rechnen, dass bestimmte Knoten plötzlich nicht mehr verfügbar sind, weil sie das Netzwerk verlassen oder durch andere Umstände getrennt wurden. Dies entspricht der Realität eines Peer-to-Peer Systems im Internet. Die untersuchten Systeme gehen aus diesem Grund von einer *Fail-Stop-Semantik* im Umgang mit anderen Knoten aus. Deshalb würde es prinzipiell ausreichen, das beabsichtigte Verlassen eines solchen Systems als Ausfall zu betrachten und auf die Selbstregeneration des Systems zu warten. In vielen Fällen profitiert das System jedoch davon, wenn das planmäßige Verlassen von einem Knoten explizit bekanntgegeben wird. So kann der aufwändigen Fehlerbehandlung des Systems vorgegriffen werden und die Austragung des Knotens planmäßig und effizient erfolgen, und nicht erst nach Entdeckung durch eine Konsistenzprüfung des Algorithmus.

Hier darf aber nicht vergessen werden, dass die generelle Möglichkeit, Knoten beim System wieder abzumelden, ein Tor für Denial-of-Service Attacken offenhält: Wird nämlich nicht weiter die Authentifikation des Knotens geprüft, der den Abmeldevorgang einleitet, so ist es auch einem böswilligen Angreifer möglich, willkürlich funktionierende Knoten vom System zu trennen und damit die Gesamtfunktion zu beeinträchtigen. Ist es dagegen nicht möglich, sich explizit vom System abzumelden, so kann dies auch nicht byzantinisch ausgenutzt werden.

Zustand des Systems als Summe seiner Teile

Kein Knoten in einem dezentralen Peer-to-Peer System hat zu jedem Zeitpunkt eine exakte und vollständige Gesamtsicht über das System. Weil kein Algorithmus den vollen Gesamtzustand in einem einzelnen Knoten speichert, folgt daraus auch, dass der Zustand des ganzen Systems nur als Vereinigungsmenge der Teilzustände auf den Knoten gesehen werden kann. Dies ist ein wesentlicher Unterschied zu den Pseudo Peer-to-Peer Algorithmen, bei denen durchaus zentrale Knoten existieren, die jederzeit über eine Sicht des Gesamtzustands verfügen.

Beispiel 3.2.1 (Sternförmige Vernetzung in Napster) *In Napster [35] ist jeder teilnehmende Knoten mit dem zentralen Server verbunden. Diesem gegenüber teilt er mit, welche Objekte von ihm lokal bereitgestellt werden. Der Server hat also zu jedem Zeitpunkt eine vollständige und aktuelle Sicht auf das Gesamtsystem, ohne dafür zuerst eine Anfrage im Netz stellen zu müssen.*

Diese Art der zentralen Sicht ist in einem echt verteilten System undenkbar, ohne dass dafür erst eine Lokalisierung im Netz gemacht werden müsste. Selbst wenn dies geschehen ist und eine Sicht auf den Gesamtzustand erlangt wurde, so veraltet diese wieder und müsste durch erneute Anfragen komplett überholt werden.

Konsistenz und Stabilisierung

Ein Peer-to-Peer Algorithmus kann nur dann zuverlässig und korrekt arbeiten, wenn sich das System in einem konsistenten Zustand befindet. Die Konsistenz folgt aber dem Prinzip, dass ein Peer-to-Peer System als Zustand die Summe der Teilzustände besitzt:

Definition 3.2.1 (Konsistenz des Gesamtsystems) *Ein Peer-to-Peer System ist genau dann in einem konsistenten Zustand, wenn sich jeder einzelne teilnehmende Knoten in einem konsistenten Zustand befindet.*

Die permanente Fluktuation von Teilnehmern in einem Peer-to-Peer System und die nicht immer einwandfreie Zuverlässigkeit von Netzanbindung sowie Teilnehmerknoten macht eine ständige Korrektur der Gesamtsicht auf das System nötig, denn fehlerhafte oder ausgefallene Knoten sollen nicht die restlichen Knoten in ihrer Zusammenarbeit stören. Diesen Vorgang der Konsistenzwahrung und -wiederherstellung nennt man *Stabilisierung*. Sie kann entweder implizit mit dem Transport normaler Nachrichten, oder als explizit angestoßener Prozeß im Peer-to-Peer System erfolgen, der dann auch zusätzliche Nachrichten generiert.

Wenn man davon spricht, wieviele Knoten gleichzeitig ausfallen dürfen, ohne dass das System Schaden nimmt, so ist mit *gleichzeitig* nicht unbedingt echte Gleichzeitigkeit gemeint. Im Sinne der Peer-to-Peer Anwendung ist es für eine Fehlersituation vollkommen hinreichend, wenn die Ausfälle in einem Intervall stattfinden, in dem der Knoten keine Stabilisierungsalgorithmen anwendet (vgl. Abbildung 3.2.3). Denn nach einem erfolgreichen Lauf einer Knotenstabilisierung der interne Zustand wieder konsistent, nach dem Lauf aller Knoten auch die Struktur des Netzes. Wichtig ist, dass

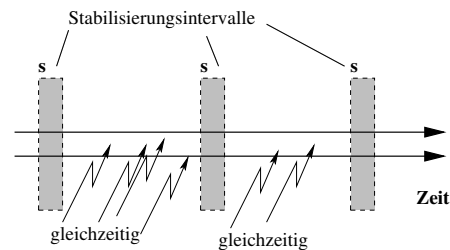


Abbildung 3.2.3: Logische Gleichzeitigkeit von Fehlern (z. B. Knotenausfällen) zwischen Stabilisierungsintervallen aus der Sicht eines einzelnen Knotens

auch bei impliziter Stabilisierung Intervalle existieren, in denen Fehler logisch gleichzeitig auftreten können. Wie wir uns erinnern, wird bei der impliziten Stabilisierung nur dann ein Fortschritt erzielt, wenn überhaupt Nachrichten im Netzwerk verschickt werden. Ist dies nicht der Fall, befindet sich ein Algorithmus mit implizitem Verfahren ebenfalls ausserhalb der Stabilisierung.

Definition 3.2.2 (Gleichzeitigkeit von Fehlern (Fail-Stop Semantik)) *Gleichzeitig im Sinne von Fail-Stop-Fehlern sind für einen einzelnen Knoten alle Ausfälle, die zwischen zwei Stabilisierungsaktionen des zugrundeliegenden Algorithmus eintreten.*

Nun kann man sich die Frage stellen, wann ein System wieder konsistent ist, wenn eine bestimmte Zeit lang überhaupt kein Fehler aufgetreten ist.

Beispiel 3.2.2 (Wiedererlangung der Konsistenz bei periodischer Stabilisierung) *Geht man von einer periodischen Stabilisierung mit der Periode T aus, und nimmt man an, dass ein Stabilisierungsvorgang maximal t_s dauert, so ist der worst case folgender:*

Die Stabilisierung hat gerade begonnen, und es tritt sofort ein Fehler ein. Wir nehmen an, dass wegen begonnener Stabilisierung der Fehler nicht mehr bemerkt wird, und erst Gegenstand der nächsten Stabilisierung sein kann. Die Zeit t , bis der Knoten wieder konsistent ist, beträgt also

$$t \leq 2T + t_s$$

Da wir diesen Fall als schlimmsten Fall angenommen haben, und jeder Knoten autonom, d. h. insbesondere zeitlich parallel, handelt, ist das Gesamtsystem nach einem Zeitintervall von $t > 2T + t_s$ ohne Fehler auf jeden Fall wieder konsistent.

Ein Knoten in einem Peer-to-Peer System kann sich solange komplett vom Verlust von Nachbarknoten erholen, solange er mindestens noch einen Kontakt hat, der voll funktionsfähig am Netz teilnimmt. Über diesen kann er, analog zum Join, alle Information einholen, die er zur Regeneration seines Zustands braucht.

Überlagerung durch logisches Netzwerk

Viele verteilten Algorithmen benutzen nicht die tatsächlich zugrundeliegende Topologie des Netzes (z. B. des Internets), sondern realisieren ihr Verfahren in einer darübergelegten Schicht auf der Basis eines abstrakten Modells, in der Literatur auch als *overlay network* [44] bezeichnet. Daraus folgt, dass das Routing eines Peer-to-Peer Algorithmus auf der Anwendungsebene, und nicht auf der tieferliegenden Netzwerkebene stattfindet. Auf die Frage, warum dies so gelöst wird, fallen sofort zwei Gründe ins Auge: Einerseits wird dadurch eine Unabhängigkeit vom verwendeten Netzwerkprotokoll und der vorhandenen Netzwerktopologie geschaffen, andererseits ist es im Zeitalter der knapper werdenden IPv4-Adressen und des noch nicht produktionsbereiten IPv6-Netzes eine verbreitete Maßnahme von Providern, den Kunden dynamische IP-Adressen statt wertvoller statischer Adressen zuzuteilen. Somit kann eine Identifikation von Teilnehmern ebensowenig auf der Netzwerkebene realisiert werden wie eine Wiederaufnahme einer bereits abgebrochenen Kommunikation durch Wiedererkennung der verwendeten ID².

Die Überlagerung durch ein logisches Netzwerk vereinfacht zwar das Routing, da sich die Knoten am Modell sehr gut orientieren können, verursacht aber auch ineffiziente Arbeit des Algorithmus. Zwei netztopologisch weit entfernte Knoten im Modell können z. B. als unmittelbare Nachbarn gelten. Es leuchtet ein, dass ein kurzer Weg im Modell nicht immer einem kurzen physikalischen Weg entsprechen muss. Allerdings können einige Algorithmen derartige Gegebenheiten berücksichtigen, und bewerten etwa Knoten und Verbindungen im Graphen durch gewichtete Metriken (vgl. Abschnitt 5.1). Solche Optimierungstechniken schließen nicht nur physikalische Gegebenheiten mit ein, sondern bemühen meist auch noch bekannte Konzepte aus der Systemprogrammierung, etwa das Caching von bereits abgefragten Daten entlang des Suchpfades (vgl. Chord) oder das Wissen über den zuletzt kontaktierten Knoten nach der LRU³-Methode.

Gemeinsamer Adreßraum von Knoten und Daten

Der Zweck der behandelten Peer-to-Peer Algorithmen ist die Lokalisierung von Ressourcen im System. Jedes solche Objekt muß vom Suchenden eindeutig identifiziert werden, weshalb auch Objekte Identifikationen tragen. Die strukturierten Algorithmen verwenden gleich geartete IDs für Knoten und lokalisierbare Objekte, so dass diese zusammen in einem Raum im Systemmodell liegen. Diese Tatsache wird dann meistens elegant zur Klärung der Frage genutzt, welcher Knoten für ein bestimmtes Objekt zuständig ist: Der/die Knoten mit der geringsten Entfernung im ID-Raum zum Objekt haben Anfragen nach dem jeweiligen Objekt zu beantworten. Für die Zuordnung der Identifikationen existieren Forderungen wie Determinismus und Gleichverteilung, sowie Nachprüfbarkeit durch andere Knoten. Aus diesem Grund bieten sich spezielle Hashfunktionen (vgl. Anhang C.1 für diese Aufgabe geradezu an. Bei Peer-to-Peer Systemen müssen nicht immer nur Daten als Objekte assoziiert werden. Ein Objekt kann je nach Anwendungszweck ebenso auch eine Indirektion, also ein Zeiger auf einen anderen Ort im Peer-to-Peer Netz sein.

²Im Folgenden soll die Abkürzung *ID* immer als Kurzform für einen Identifikationsstring eines Knotens oder Objekts verwendet werden.

³Last Recently Used

Art der Speicherung von Daten

Um Daten im Peer-to-Peer Netz persistent zu machen, gibt es zwei Herangehensweisen: Entweder, der publizierende Knoten muss die im Netz verteilten Kopien auf irgendeine Art und Weise wieder auffrischen, weil sie sonst nach einiger Zeit im Rahmen der *garbage collection* verfallen. Eine Auffrischung muss nicht nur das erneute Ansprechen des Objekts durch den Knoten, der es ins System gestellt hat, sein. Denkbar ist auch, dass sich die Zeit, die ein Objekt auf einem Knoten gespeichert bleibt, bei jedem Zugriff wieder erhöht. In diesem Fall spricht man von einer *dynamischen* Speicherung. Oder aber eine Kopie eines Objekts muss explizit gelöscht werden, damit sie nicht mehr verfügbar ist. Dann liegt eine *permanente* Speicherung von Objekten vor.

Je nach Verwendungszweck ist die eine oder andere Methode praktischer. Für zensurresistente und anonyme Netzwerke empfiehlt sich offensichtlich die permanente Speicherung, wohingegen die dynamische Variante im Allgemeinen weniger Verwaltungsaufwand fordert (weil nicht mehr benötigte Objekte automatisch einer *garbage collection* anheim fallen) und deshalb größere Flexibilität für den Entwickler bietet. Diese Art der Speicherung ist bedarfsorientiert, da nicht zugegriffene oder aufgefrischte Objekte automatisch entsorgt werden. Die permanente Speicherung hingegen bietet Angriffsfläche für Denial-of-Service: Durch das Speichern unsinniger Daten bleibt weniger Speicherplatz für die wirklichen Anwender. Weil die Daten ja per Definition permanent gespeichert werden, ist ein erheblicher Aufwand zum Löschen der sinnlosen Daten nötig, sofern dies von Systemwegen her überhaupt möglich ist.

Vollindizierung vs. Minimalvernetzung von Knoten

Betrachtet man das Peer-to-Peer System als eine Menge von Knoten, so landet man beim Modell des allgemeinen gerichteten Graphen (hierzu siehe auch die mathematischen Grundlagen in Anhang

Kriterium	voll	minimal
Anzahl der Kanten im Graphen	$\frac{N(N-1)}{2}$	N
Wahrscheinlichkeit, einen bestimmten Knoten als Nachbarn zu haben	1	$\frac{1}{N}$
Anzahl der Einträge (Knoten) im lokalen Zustand	$N - 1$	1
mittlere Weglänge des Routings	1	$\frac{N}{2}$

Tabelle 3.2.1: Eigenschaften der beiden Extremmöglichkeiten in einem zusammenhängenden gerichteten Graphen mit N Knoten

C). Sei nun jede gerichtete Kante ein Hinweis darauf, dass ein Knoten direkt über die Existenz eines anderen Knotens informiert ist. Dann existieren zwei extreme Formen, wie Peer-to-Peer Netzwerke organisiert sein können (siehe Tabelle 3.2.1): Entweder jeder Knoten ist mit jedem anderen und sich selbst verbunden — dann spricht man von einem vollständig vernetzten Graphen. Dies kommt einer Vollindizierung im Peer-to-Peer System gleich, jeder teilnehmende Knoten hält also im lokalen Zustand ein komplettes Verzeichnis aller Knoten des Systems.

Hält hingegen jeder Knoten nur einen Knoten als weiteren Kontakt, führt dies bei einem gerichteten Graphen zu einer Anordnung auf einem Zyklus, der aufgrund der Kantenausrichtung in genau einer Weise traversiert werden kann. Es leuchtet ein, dass mit zunehmendem Verbindungsgrad der Knoten eine höhere Wahrscheinlichkeit einhergeht, einen beliebigen Knoten direkt als Nachbarn zu

haben. Als Folge nimmt die mittlere Weglänge eines Routings ab und die Anzahl der im lokalen Zustand gehaltenen Knoten zu. Weil die Knoten, die im lokalen Zustand gelistet werden, auch aktuell gehalten werden müssen, nimmt auch der Stabilisierungsaufwand zu. Alle untersuchten Peer-to-

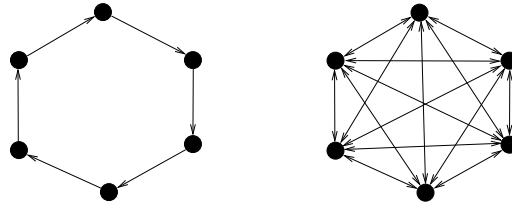


Abbildung 3.2.4: Minimale und volle Vernetzungen in einem zusammenhängenden gerichteten Graphen mit 6 Knoten. Der rechte, vollvernetzte Graph wird auch als K_6 bezeichnet.

Peer Systeme realisieren einen Mittelweg zwischen diesen beiden Polen (siehe Abbildung 3.2.4) und bilden ihre eigene Balance zwischen kurzem Routing und akzeptabler Datenmenge pro Knoten bzw. Stabilisierungsaufwand. Ein generelles Ziel aller Algorithmen ist die Optimierung der mittleren Weglänge, die eine Suchanfrage nach einem bestimmten Objekt im System zurücklegen muss. Hierbei wird ein Aufwand von $\mathcal{O}(\log N)$ Schritten angestrebt, wobei N die Gesamtzahl der Knoten bezeichnet.

Aufwandsbetrachtung

In einem verteilten System ist es nicht teuer, aufwendige lokale Berechnungen auszuführen. Dies rührt daher, dass die Netzwerklatenz im Vergleich zu der Länge von CPU-Zyklen um Größenordnungen höher liegt. Deshalb ist eine aufwendige Berechnung, die aber zu einer Reduzierung des Kommunikationsaufwands führt, in verteilten Systemen generell ein guter Tausch.

Wenn die Kommunikation signifikant für die Performanz eines verteilten Systems ist, so ist auch seine Komplexitätsbetrachtung in der Kommunikation, und nicht in Rechenschritten zu führen. Der Einfachheit halber nehmen wir an, dass lokale Rechenschritte, die der Kommunikation dienen, keine Zeit benötigen und konzentrieren uns auf die Kommunikation, d. h. auf die Anzahl der Hops⁴, verglichen mit der Gesamtzahl aller am System beteiligten Knoten. Zur Darstellung der Kosten in einem Peer-to-Peer-Algorithmus verwenden wir in dieser Arbeit folgende Bezeichnungen:

Variable	Erklärung
C_F	Kosten, einen Knoten zu finden (in Hops)
D_V	Größe der Verwaltungsstrukturen ohne Objekte auf jedem Knoten
C_I	Kosten, ein Objekt ins System einzufügen (in Hops)
C_D	Kosten, ein Objekt aus dem System zu löschen (in Hops)
C_J	Kommunikationskosten, die ein hinzukommender Knoten verursacht
C_L	Kommunikationskosten, die ein wegfallender Knoten verursacht
N_J	Anzahl Knoten, die ihre Verwaltungsstrukturen nach einem Join auffrischen
N_L	Anzahl Knoten, die ihre Verwaltungsstrukturen nach einem Leave auffrischen

Zum Bearbeiten eines Objekts muss auch zuerst der zuständige Knoten gefunden werden. Deshalb sind C_I und C_D im Allgemeinen mindestens genauso groß wie C_F .

⁴Anzahl der Weiterleitungen einer Nachricht zwischen zwei Knoten.

Kapitel 4

Die Algorithmen

Die Auswahl der in diesem Kapitel betrachteten Algorithmen fiel aufgrund der Vielseitigkeit, der wissenschaftlichen Relevanz und der Bekanntheit: Die fünf Algorithmen Chord/DHash, CAN, Pastry, Tapestry und Kademia sind zum einen grundlegend genug, um darauf eine Vielfalt von Anwendungen zu realisieren, zum anderen existieren über alle fünf ausreichend viel wissenschaftliche Arbeiten, die die Systeme theoretisch beschreiben. Darüberhinaus sind diese fünf die mit am meisten referenzierten Systeme in allen betrachteten Papers. Die Algorithmen werden nach den vorher festgelegten Kriterien analysiert und bewertet.

An passender Stelle wird der theoretisch grundlegende Algorithmus von PLAXTON besprochen, da Pastry und Tapestry stark an PLAXTONs Konzept angelehnt sind. Vergleichende Bemerkungen und Ergebnisse sind in Kapitel 5 zu finden.

4.1 Chord und DHash

Das Chord Projekt ist am Laboratory of Computer Science des MIT entstanden. Die existierende Dokumentation [53, 52, 12, 28] besteht aus einer Vielzahl von Arbeiten, die sich sowohl mit der Theorie, als auch mit Anwendungen von Chord beschäftigen. Grund für das Forschungsprojekt war offenbar der Bedarf an einem Algorithmus, der sich durch seine Universalität für viele Anwendungen [11, 13, 10] eignet und darüberhinaus den Anforderungen der Forscher am MIT an Robustheit und echtem Peer-to-Peer gerecht wird. Das Projekt ist auf seiner Homepage [31] mitsamt der wissenschaftlichen Arbeiten zu finden, ebenso wie ein CVS¹-Repository, auf das weltweit anonym Lesezugriff möglich ist. Es ist also kein Problem, den Quellcode kostenfrei herunterzuladen und damit zu experimentieren. Der Code selbst steht unter einer freien, BSD-ähnlichen Lizenz².

Im Laufe dieses Abschnitts werden wir zeigen, dass es sich bei Chord um einen **strukturierten, deterministischen** und **homogenen** Peer-to-Peer Algorithmus handelt.

Systemmodell

Chord stellt selbst lediglich einen Mechanismus zum Routing dar, kann aber selbst keine Objekte speichern. DHash setzt auf Chord auf und realisiert eine Speicherung binärer Datenblöcke als verteilte Hashtabelle, auf der selbst wieder Anwendungen realisiert werden können (vgl. Abbildung 4.1.1). Weil alle anderen untersuchten Algorithmen die Trennung zwischen Routing und Objekt-

¹Concurrent Version System [5].

²Eine Übersicht über die verschiedenen Softwarelizenzen ist unter [51] zu finden.

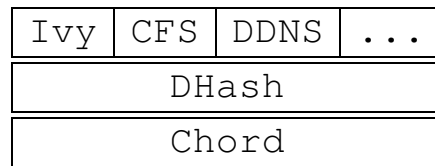


Abbildung 4.1.1: Das Schichtenmodell: Chord als Routing-Schicht, die zuverlässige Speicherung binärer Datenblöcke in DHash und verschiedenste Anwendungen, die darauf aufsetzen.

Speicherung nicht derart explizit in zwei Schichten vornehmen, werden in diesem Abschnitt sowohl Chord als auch DHash erläutert, um einen Vergleich mit den anderen Systemen eingehen zu können.

Wir beginnen mit der Beschreibung von Chord: Das System wird als modularer eindimensionaler Ring der Größe $N \in \mathbb{N}$ gesehen, in dem die Knoten ihrer ID $n \in \Sigma^m$, $m \in \mathbb{N}$ nach geordnet residieren. Die ID, die ein Knoten besitzt, errechnet sich über eine Hashfunktion $h : \Sigma^* \rightarrow \Sigma^m$, $m \in \mathbb{N}$ aus der IP-Adresse des jeweiligen Knotens (siehe Beispiele in Tabelle 4.1.1). Der Bildraum der Hashfunktion spannt genau den Ring auf. Weil Chord die auf 160 bit abbildende SHA-1³ als Hashfunktion einsetzt, ergeben sich folgende konkreten Werte: $N = 2^{160}$, $m = 160$, $\Sigma = \{0, 1\}$.

IPv4-Adresse	Hashwert nach SHA-1 (hexadecimal)
131.188. 40. 91	d52673562bd1a6bc0554e43351aa5e10dd7e6186
131.188. 40. 92	c4408d29305454b125c9874e61b14b2710f50d59
129.128. 5.191	02c9bc895494abbd1eedec506bcd59c00725da16
193. 99.144. 71	ac0769363aa0f83cf3cad8c572d29f0db7aa75ec
64. 90.164. 50	1fd6eb1b90e9aa74b6b4568a7d5583da3465aa53
213. 73. 91. 29	023cd81198bd2fff4bbf08589a3560636543cf6d

Tabelle 4.1.1: Beispiele für Knoten-IDs im Chord-Ring.

Die Tatsache, dass jeder Knoten in Chord die gleichen Aufgaben wahrnimmt und die selbe Funktionalität bereitstellt, zeichnet Chord als *homogenen* Peer-to-Peer Algorithmus aus.

Lokalisierung von Knoten in Chord: Um nun einen bestimmten Knoten in Chord zu finden, bedient sich der anfragende Knoten n verschiedener Funktionen, die bei Chord alle als *RPC-Aufrufe*⁴ realisiert sind. Hierdurch macht es semantisch keinen Unterschied, ob die Prozedur gerade auf dem Knoten selbst, oder auf einem entfernten Knoten aufgerufen wird.

Algorithmus 4.1.1 $n.\text{lookup}(id)$

Benötigt: n ist ein Knoten auf dem Ring

Stellt her: Gibt den für id zuständigen Knoten auf dem Ring zurück.

$t \leftarrow \text{find_predecessor}(id);$

$s \leftarrow t.\text{successor}[1];$

return $s;$

Eine Anfrage nach einer ID wird über die Prozedur `lookup()` (Algorithmus 4.1.1) des eigenen Knotens abgewickelt. Hierzu muss als Parameter eine ID auf dem Ring übergeben werden.

³SHA-1 wird, genau wie alle anderen in dieser Arbeit erwähnten kryptographischen Hashfunktionen, in SCHNEIERS Buch [46] beschrieben.

⁴*Remote Procedure Call*, definiert in RFC 1831 [50].

Die Prozedur liefert dann den jeweils unmittelbar auf die ID folgenden, aktiven Knoten zurück. Intern wird noch Gebrauch von den Prozeduren `find_predecessor()` und `closest_preceding_finger()` (Algorithmen 4.1.2 und 4.1.3) gemacht. Jeder Knoten in Chord ist für Anfra-

Algorithmus 4.1.2 $n.find_predecessor(id)$

Benötigt: n ist ein Knoten auf dem Ring

Stellt her: Gibt den Vorgängerknoten von id auf dem Ring zurück.

```

 $t \leftarrow n;$ 
while  $id \notin [t, t.successor[1]]$  do
     $t \leftarrow t.closest\_preceding\_finger(id);$ 
end while
return  $t;$ 

```

Algorithmus 4.1.3 $n.closest_preceding_finger(id)$

Benötigt: n ist ein Knoten auf dem Ring, $m = \log_2 N$ die Anzahl Einträge in der Fingertabelle.

Stellt her: Gibt den Knoten der Fingertabelle zurück, der am nächsten vor id auf dem Ring liegt.

```

for  $i = m$  downto 1 do
    if  $finger[i].node \in [n, id]$  then
         $f \leftarrow finger[i].node;$ 
        return  $f;$ 
    end if
end for
return  $n;$ 

```

gen in den Abschnitt auf dem logischen Ring zwischen seinem Vorgänger und sich selbst zuständig (vgl. Abbildung 4.1.2). Weil Chord eine verteilte Hashtabelle realisiert, kann der Lookup als Zuordnungsmechanismus einer Schlüssel-Wert-Beziehung gesehen werden (wobei die Schlüssel aus dem Definitionsraum der Hashfunktion einen flachen Namensraum aufspannen und die Werte den Knoten-IDs auf dem Chord-Ring entsprechen). Auch folgt hieraus, dass es sich bei Chord um einen *strukturierten* Peer-to-Peer Algorithmus handelt. Die Chord-Schicht findet zu jedem übergebenen Schlüssel den dafür verantwortlichen Knoten. Dies fassen wir in für Chord grundlegende Definitionen:

Definition 4.1.1 (Zuständigkeit im Chord-Ring) *Für die Auskunft über ein bestimmtes Datum $n \in \Sigma^m$; ($m \in \mathbb{N}$; $\Sigma = \{0, 1\}$) ist immer der nächste auf dem Ring folgende, aktive Knoten zuständig.*

Intern traversiert die `lookup()`-Funktion den Chord-Ring, bis der nächste Knoten gerade größer als die gesuchte ID n ist, und liefert dann die ID dieses Knotens zurück. Hierbei wird jeder traversierte Knoten nach seinem unmittelbaren Nachfolger gefragt, den er im Rahmen seines eigenen Zustands stets aktuell halten muss. Da dieses Vorgehen sehr ungünstig skaliert, wurden einige Mechanismen in Chord eingebaut, die die Laufzeit erheblich verbessern.

Als nächstes ist es wichtig, die zurückgelegte Entfernung bei einem Suchvorgang ausdrücken zu können. Da auf dem Chord-Ring ein Drehsinn existiert, drückt sich die Entfernung auf dem Ring folgendermaßen aus:

Definition 4.1.2 (Entfernungsbegriff in Chord) Die Entfernung zweier Knoten x und y (bezeichnet durch ihre binären Knoten-IDs) ist die zurückgelegte Bogenlänge auf dem Chord-Ring unter Berücksichtigung des positiven Drehsinns. Es gilt also:

$$d(x, y) := y - x \pmod{2^{160}}; \quad x, y \in \mathbb{Z}_{2^{160}}$$

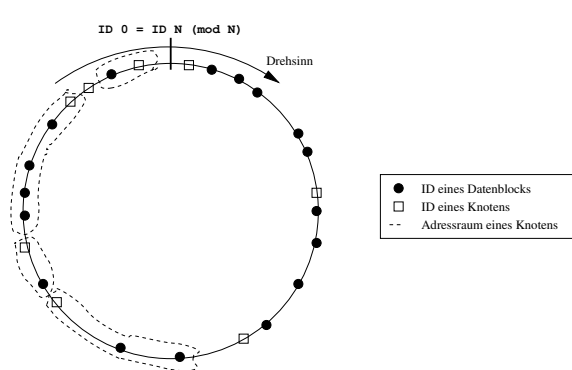


Abbildung 4.1.2: Der Chord-Ring mit Teilnehmerknoten und Daten, für die die Knoten zuständig sind. Nur teilweise gekennzeichnete Adressräume.

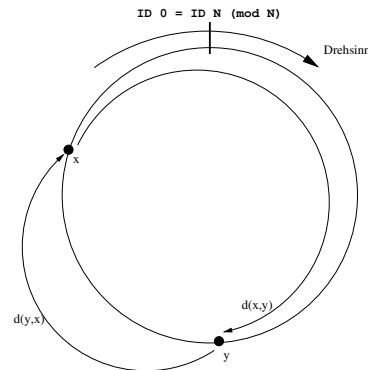


Abbildung 4.1.3: Keine echte Metrik in Chord: Die Entfernung zweier Punkte auf dem Chord-Ring kann unterschiedlich sein, je nachdem, von wo aus man misst.

Bei dem Chord-Ringmodell handelt es sich *nicht* um einen metrischen Raum (siehe Anhang C.3 und Abbildung 4.1.3), da die Entfernung zweier Knoten zueinander wegen des fest definierten Drehsinns im Ring nicht symmetrisch ist. Es gilt also:

$$\exists x, y \in \Sigma^m : d(x, y) \neq d(y, x); \quad m \in \mathbb{N}; \quad \Sigma = \{0, 1\}$$

Die Suche im Ring wird durch Einführung einer sogenannten *Fingertabelle* beschleunigt, die die Komplexität von Anfragen von einem linearen auf ein logarithmisches Niveau senkt. Ermöglicht wird dies durch das Wissen über Knoten, die jeweils unmittelbar nach einer bestimmten, exponentiell wachsenden Entfernung vom aktuellen Knoten auf dem Ring liegen. Die Formel für den Startpunkt der Finger-Intervalle des Knotens n lautet:

$$n.\text{finger}[i].\text{start} := n + 2^{i-1} \pmod{2^m}; \quad 1 \leq i \leq m = \log_2 N \quad (4.1)$$

Je nach Ziel der Anfrage ist es somit möglich, nicht mehr den direkten Nachfolger eines Knotens, sondern denjenigen Knoten aus der Fingertabelle anzuspringen, der noch gerade *vor* der gesuchten ID liegt. Verfährt jeder Knoten auf diese Weise, wird bei jedem Schritt mindestens die Hälfte der Distanz zum Ziel zurückgelegt, was in einer logarithmischen Laufzeit resultiert.

Beschleunigung für die in Algorithmus 4.1.4 dargelegte Stabilisierung wird durch das Halten des Vorgängerzeigers *predecessor* erreicht. Selbstverständlich könnte der direkte Vorgänger des Knotens bei Bedarf auch ad hoc (durch Verwendung von Algorithmus 4.1.2) ermittelt werden. Schneller ist jedoch, den Zeiger aktuell zu halten, und ihn bei Bedarf sofort zugreifen zu können.

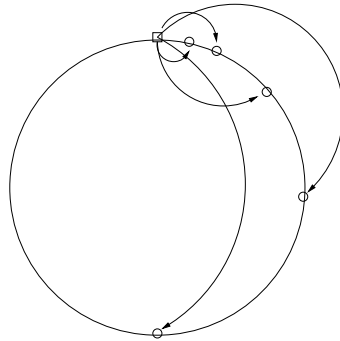


Abbildung 4.1.4: Ein Knoten im Chord-Ring und seine im Abstand von Zweierpotenzen entfernten Sprungziele, deren direkte Nachfolgerknoten in der Fingertable gelistet werden

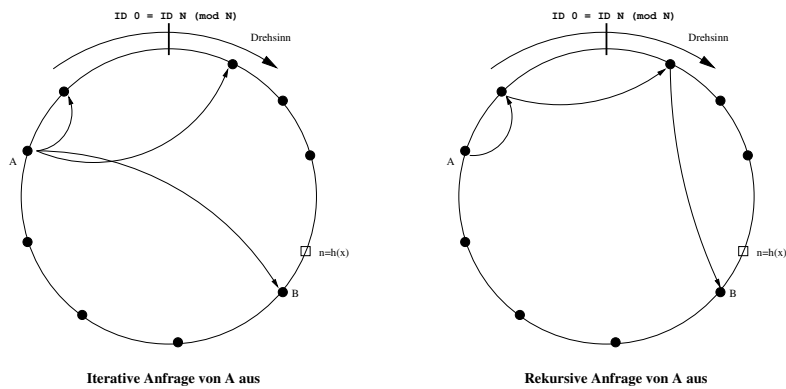


Abbildung 4.1.5: Iteratives und rekursives Lookup in Chord: Gleicher Nachrichtenaufwand wegen RPCs, aber unterschiedliche Fehleranfälligkeit.

Bemerkung: Die Arbeiten über Chord führen die Lokalisierung als rekursiven Algorithmus ein, in der Implementation wird jedoch eine von der Funktionalität äquivalente, aber stabile iterative Prozedur verwendet (siehe Abbildung 4.1.5). Der Nachrichtenaufwand ist bei beiden Varianten gleich, da wegen der Verwendung von *Remote Procedure Calls* jeder Anfrage standardmäßig eine Antwort folgt. Jedoch ist die rekursive Kette von Nachrichten viel leichter unterbrechbar als die sternförmige iterative Variante. Eine Verringerung des Nachrichtenaufwands wäre denkbar, wenn in der rekursiven Variante der Knoten B direkt dem Knoten A antwortet, anstatt die Antwort an der Aufrufkette entlang zurückzugeben. Dies ist aber wie erwähnt mit RPC nicht möglich.

Lokalisierung von binären Datenblöcken in DHash: Aufgabe der DHash-Schicht ist die zuverlässige Speicherung von Datenblöcken. Um ein schnelles und eindeutiges Auffinden zu ermöglichen, profitiert die Schicht vom darunterliegenden Chord und seinem Lokalisierungsprimitiv `lookup()`. Selbst stellt DHash folgende drei Operationen zur Verfügung:

Operation	Beschreibung
<code>put_h(block)</code>	Berechnet den Hashwert $n = h(\text{block})$ des übergebenen Blocks <code>block</code> und speichert ihn auf dem Knoten, der laut Chord (<code>lookup(n)</code>) dafür zuständig ist.
<code>put_s(block, pubkey)</code>	Spezielle <code>put</code> -Funktion, für Blöcke, die kryptographische Signaturen tragen. Der übergebene Block <code>block</code> muss mit demjenigen Private Key signiert sein, der mit dem übergebenen Public Key <code>pubkey</code> korrespondiert. Der Block wird anschließend unter dem Hashwert $n = h(\text{pubkey})$ auf dem Chord-Ring gespeichert.
<code>get(key)</code>	Liefert den zu dem Hashwert <code>key</code> gespeicherten Block aus dem Ring zurück.

Unbedingt nötig für die Schlüssel-Wert-Zuordnung ist die Hashfunktion h , die einen Bitstring $x \in \Sigma^*$ auf den Chord-Ring abbildet. Ein gesuchtes Objekt wird durch seinen Hashwert eindeutig identifiziert. Zusammengefasst kann man also ausgehend vom Objekt A den zuständigen Knoten k auf folgende Weise auf dem Chord-Ring finden:

$$k = \text{lookup}(h(\text{id}(A))) \quad (\text{lookup} : \Sigma^m \rightarrow \Sigma^m; \\ h : \Sigma^* \rightarrow \Sigma^m; \\ \text{id} : \mathcal{A} \rightarrow \Sigma^*)$$

Tolerierte Fehler

In diesem Kapitel sollen nur Fehler nach der *Fail-Stop* Semantik diskutiert werden. Byzantinische Fehler können mit diesen einfachen Vorkehrungen nicht abgefangen werden.

Ausfall von Knoten in Chord — redundante Speicherung der Nachfolger: Eine Lokalisierung in Chord benutzt den jeweiligen Nachfolger eines Knotens auf dem Ring, um die `lookup()`-Funktion auszuführen. Das bedeutet, dass bereits der Ausfall *eines* Knotens genügen würde, um den Ring zu unterbrechen und das System unbrauchbar zu machen. Um dem vorzubeugen, hält jeder Knoten nicht nur *einen* Nachfolgerzeiger, sondern eine Nachfolgerliste der Größe r , wobei der Parameter r prinzipiell frei wählbar ist. In der Literatur zu Chord wird allerdings empfohlen, mit einem $r = 2 \log_2 N$ zu arbeiten, wobei N die maximale mögliche Anzahl der teilnehmenden Knoten

darstellt. Ganz offensichtlich müssen also r Knoten, die *in einer Reihe* auf dem Chord-Ring liegen, *gleichzeitig*⁵ ausfallen, um den Ring unbrauchbar zu machen. Setzt man für die Wahrscheinlichkeit, dass ein Knoten in einem bestimmten Zeitintervall ausfällt, den Wert p an, so ergibt sich für einen bestimmten Knoten k_1 die Wahrscheinlichkeit, dass er und seine r Nachfolgerknoten k_1, \dots, k_r gleichzeitig ausfallen zu

$$P(\text{alle } \{k_1, \dots, k_r\} \text{ fallen aus}) = p^r$$

Beispiel 4.1.1 (Knotenausfall) Wir gehen von einem sehr pessimistischen $p = 0.5$ aus, und nehmen einen Wert von r aus der Praxis an, der sich aus $N = 2^{160}$ wie folgt errechnet: $r = 2 \log_2 2^{160} = 320$. Selbst dann ist $P(\text{alle } \{k_1, \dots, k_r\} \text{ fallen aus}) = 0.5^{320} = 4.68 \cdot 10^{-97}$, was eine verschwindend geringe Wahrscheinlichkeit darstellt.

Wie man im Beispiel sieht, ist ein Brechen des Chord-Rings aufgrund von Knotenausfällen praktisch unmöglich, wenn oft genug Stabilisierungen stattfinden. Allerdings ist hiermit nur ein Ausfall der Knoten nach *Fail-Stop*-Semantik gemeint, nicht byzantinisches Fehlverhalten.

Verlust von DHash-Datenblöcken auf einzelnen Knoten — redundante Datenhaltung: Angenommen, der für einen Datenblock zuständige Knoten ist ausgefallen, oder weist aus irgendwelchen Gründen Datenverlust auf, und kann deshalb nicht den angeforderten Datenblock liefern, den das DHash-Protokoll bei ihm anfordert. Dies würde einen *single point of failure* darstellen, weil der Datenblock nirgendwo anders gespeichert ist. Deshalb wird auch hier Redundanz eingeführt, die den Datenverlust auf einem Knoten tolerierbar macht.

Betrachten wir zunächst den Komplettausfall eines Knotens: Die Zuständigkeit für den Namensraum, den der Knoten bisher bedient hat, würde per Definition automatisch an den *Nachfolger* des Knotens auf dem Ring fallen. Also bietet es sich für eine redundante Speicherung geradezu an, die Duplikate des Datenblocks *ebenfalls* auf den Nachfolgern des zuständigen Knotens zu speichern, da im Falle eines Fehlers nicht einmal mehr ein Kopieren nötig würde. Es leuchtet ein, dass es keinen Sinn macht, mehr als r Instanzen des Datenblocks auf den Nachfolgern zu speichern, da bei einem Ausfall von r Knoten in einer Reihe der $(r + 1)$ te Knoten ohnehin nicht mehr erreicht werden könnte, da zu diesem Zeitpunkt der darunterliegende Chord-Ring bereits defekt ist. k ist also wählbar mit $1 \leq k \leq r$, wodurch die Wahrscheinlichkeit, dass ein Datenblock aufgrund von Datenverlust oder Knotenausfall nicht mehr auffindbar ist, sich analog zu der o. g. Wahrscheinlichkeit, dass der Chord-Ring unbrauchbar wird, berechnen lässt. Auch dieser Fall wird durch hinreichend kleine Stabilisierungsintervalle praktisch unmöglich gemacht.

Zustandsspeicherung

Jeder Knoten hält folgende Datenstrukturen, die seine Sicht auf den Ring wiedergeben:

⁵im Sinne von Abschnitt 3.2: Stabilisierung und Gleichzeitigkeit

Element	Beschreibung
Nachfolgerliste successor[1..r]	Beinhaltet r Zeiger auf die unmittelbaren Nachfolgerknoten auf dem Ring. Obwohl nur <i>ein</i> Nachfolgerzeiger für ein funktionierendes Protokoll nötig wäre, benötigt man für die Fehlertoleranz des Algorithmus eine Liste von Zeigern.
Finger Table finger[1..m]	<i>Optional</i> für das Verfahren, bringt aber einen erheblichen Gewinn an Geschwindigkeit und ermöglicht die später im Text beschriebene logarithmische Anzahl von Schritten zum Finden eines Knotens.
Vorgängerzeiger predecessor	<i>Optional</i> , da der Vorgänger (der z. B. bei jedem <code>lookup()</code> -Aufruf benötigt wird) auch durch eine iterierte Folge von Lookups selbst bestimmt werden könnte (nämlich mit <code>find_predecessor()</code> , Algorithmus 4.1.2). Allerdings ist die Speicherung und periodische Aktualisierung in dieser Variable günstiger für das Verfahren.

Bemerkung: Die Werte der Variablen `successor[1]` und `finger[1].node` sind immer gleich, da nach Formel 4.1 (Seite 21) gilt:

$$\begin{aligned} \text{finger}[1].\text{start} &= n + 2^{1-1} = n + 1 \\ \text{find_successor}(n + 1) &= \text{successor}[1] \end{aligned}$$

Hierbei bezeichnet die Komponente `finger[1].start` die errechnete Start-ID des ersten Finger-Intervalls.

Stabilisierung

Die Stabilisierung in Chord erfolgt explizit. Eine implizite Stabilisierung ist in Chord wegen des festgelegten Drehsinns im Ring nicht möglich, denn sie würde voraussetzen, dass die Empfänger einer Nachricht auch an den Senderknoten Anfragen richten. Meist ist der Sender aber im Drehsinn vor dem Empfänger einer Nachricht angeordnet, und der Sender in keiner der lokalen Datenstrukturen des Empfängers (Nachfolgerliste, Fingertabelle) gelistet. Eine Information, ob der Sender noch online ist, kann also durch implizite Verfahren nicht gewinnbringend verwertet werden und muss für die Stabilisierung verworfen werden. Aufgabe der Stabilisierung ist es, den Chord-Ring konsistent zu halten:

Definition 4.1.3 (Konsistenz in Chord) *Ein Chord-Ring ist genau dann konsistent, wenn jeder Knoten seinen direkten Nachfolger auf dem Ring kennt und der jeweilige Nachfolger korrekt funktioniert.*

Stattdessen erfolgt eine durch den jeweiligen Knoten selbst initiierte, periodische Stabilisierung. Hierzu ruft der Knoten die Funktionen `stabilize()` (Algorithmus 4.1.4) und `fix_fingers()` (Algorithmus 4.1.6) auf. Denkt der Knoten n , er wäre der direkte Vorgänger eines anderen Knotens k , so ruft er dessen Funktion `notify()` (Algorithmus 4.1.5) auf.

Join und Leave

Ein neu hinzukommender Knoten n muss nacheinander folgende Dinge erledigen:

1. Die eigenen Datenstrukturen (`successor` und `predecessor`) initialisieren. Dies geschieht mithilfe der Ergebnisse von Anfragen, die n an seinen *ersten Kontakt* (vgl. Abschnitt 3.2) stellt.

Algorithmus 4.1.4 $n.stabilize()$

Benötigt: Nichts. Dient der Stabilisierung, muss periodisch aufgerufen werden.**Stellt her:** Verbessert den Nachfolgerzeiger, wenn nötig. Macht den Nachfolger auf den lokalen Knoten aufmerksam. $x \leftarrow \text{successor}[1].\text{predecessor};$ **if** $x \in [n, \text{successor}[1]]$ **then** $\text{successor}[1] \leftarrow x;$ **end if** $\text{successor}[1].\text{notify}(n);$

Algorithmus 4.1.5 $k.\text{notify}(n)$

Benötigt: Aufruf, wenn n denkt, es wäre der direkte Vorgänger von k .**Stellt her:** Verbesserung des Vorgängerzeigers, wenn nötig.**if** $(\text{predecessor} = \text{NULL}) \vee (n \in [\text{predecessor}, k])$ **then** $\text{predecessor} \leftarrow n;$ **end if**

2. Die r Vorgängerknoten von n darauf aufmerksam machen, dass n jetzt in den Ring aufgenommen wird. Diese aktualisieren daraufhin ihre Sicht.
3. Die nächsthöhere Softwareschicht (DHash) informieren, weil nun die Daten im Intervall $[\text{predecessor}(n), n]$ von n übernommen werden. Nach Erhalt der Information kann DHash das Kopieren von Blöcken einleiten.

Eigentlich müsste bei jedem Hinzukommen der komplette Datenbestand des Knotens, also auch die gesamte Fingertabelle und die Tabellen anderer Knoten initialisiert werden. Dieses Konzept wird im Konferenzpaper zu Chord [52] als *simple Join* vorgestellt. In der Realität wird allerdings der viel einfachere Join-Algorithmus 4.1.7 verwendet, und der Rest der periodisch aufgerufenen Stabilisierung überlassen. Diese Modifikation wird der hohen Fluktuation von Knoten und dem problematischen Verarbeiten gleichzeitig auftretender Join-Aufrufe gerecht. Die Autoren bezeichnen diese Variante des Algorithmus als *concurrent Join*. Obwohl ein Fail-Stop-Verhalten zum Verlassen des Rings durchaus akzeptabel ist, profitiert das System von folgenden Aktionen, die ein Knoten vor dem planmäßigen Verlassen des Rings ausführen kann. Diese Aktionen werden in [53] beschrieben:

1. Auf DHash-Schicht werden die gespeicherten Blöcke an den Nachfolger weitergegeben. Dieser Schritt ist durch die Redundanz in DHash allerdings nicht unbedingt notwendig. Wichtiger ist, dass der Nachfolger über seine neue Zuständigkeit informiert wird.
2. Das Verlassen des Rings dem Vorgänger und dem Nachfolger mitteilen. Beide können dann ihre Zeiger aktualisieren, und die Lücke im Ring wird nicht erst bei der nächsten Stabilisierung, sondern sofort geschlossen.

Algorithmus 4.1.6 $n.\text{fix_fingers}()$

Benötigt: Muss periodisch aufgerufen werden.**Stellt her:** aktualisiert einen zufällig gewählten Eintrag der Finger Tabelle. $i \leftarrow \text{random}(m);$ $\text{finger}[i].\text{node} \leftarrow \text{find_successor}(\text{finger}[i].\text{start});$

Algorithmus 4.1.7 $n.join(w)$ **Benötigt:** w ist bereits ein Knoten im Chord-Ring**Stellt her:** Initialisiert den lokalen Zustand von n predecessor \leftarrow NULL;successor[1] $\leftarrow w.find_successor(n)$;

Ein anderes, unfallartiges Verlassen des Ringes ist durch die Fail-Stop Semantik eines Knotens gegeben: Antwortet ein Knoten nicht mehr auf Anfragen, so wird sein Nachfolger auf dem Ring herangezogen und der Knoten selbst vom Protokoll ausgeschlossen.

Verwaltungsaufwand

Bei der Komplexitätsbetrachtung wird von einem konsistenten Zustand des Chord-Rings laut Definition 4.1.3 ausgegangen.

Finden eines Knotens in Chord (C_F): Das Lokalisieren eines Knotens in Chord wird komplett über die Finger Tabelle abgewickelt, falls diese konsistent ist. Ansonsten wird auf die weniger Fortschritt pro Hop bringende, aber meist robustere Nachfolgerliste zurückgegriffen. Dieser Fall interessiert hier aber nicht.

Die Finger Tabelle hält — wie bereits erläutert — Knoten, die sich jeweils in Entfernung von Zweierpotenzen zum aktuellen Knoten befinden. Wird nun eine Lokalisierung einer bestimmten ID in Auftrag gegeben, so kann per Finger Tabelle in jedem Schritt mindestens die Hälfte der Entfernung zum Ziel zurückgelegt werden (vgl. Abbildung 4.1.6). Deshalb ist die Lokalisierung eines Knotens insgesamt in einer logarithmischen Anzahl von Hops, gemessen in der Gesamtzahl der Knoten, möglich. Es gilt $C_F = \mathcal{O}(\log N)$.

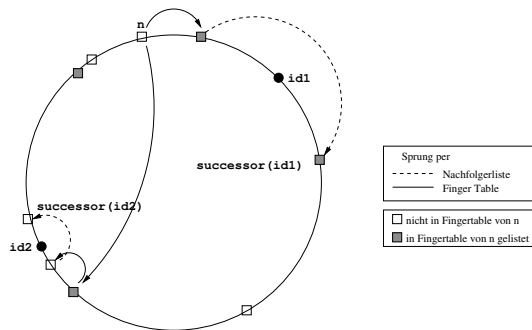


Abbildung 4.1.6: Zwei Beispiele für logarithmische Lookups vom Knoten n aus. Der Nachfolger von $id1$ kann sofort mit einem weiteren Sprung erreicht werden, $id2$ wird lediglich gut approximiert.

Größe der Verwaltungsstrukturen (D_V): Drei Komponenten machen den lokalen Zustand eines Chord-Knotens aus: Die Fingertabelle mit $\log_2 N = m$ Einträgen, der Zeiger auf den unmittelbaren Vorgänger und die Nachfolgerliste mit r Einträgen. Zusammen ergibt dies einen Aufwand von $D_V = 1 + r + \log_2 N$. Mit den in Chord verwendeten Werten ergäbe sich ein numerischer Wert von $D_V = 1 + 320 + 160 = 481$, dieser ist aber für die Komplexitätsbetrachtung nicht von Interesse, da aus diesem Wert keine Abhängigkeit von der Gesamtzahl N der Knoten mehr abzulesen ist.

Kosten, ein Objekt zu lesen (C_R): Zum Lesen eines Objekts muss im betrachteten konsistenten Fall nur der Knoten herangezogen werden, der für die ID des Objekts zuständig ist. Dieses Auffinden ist mit Kosten von C_F zu bewerkstelligen. Hinzu kommt nur noch eine Kommunikationskonstante C , die den Aufwand für die Datenübertragung zum suchenden Knoten darstellt. Es ergibt sich also $C_R = \mathcal{O}(\log N) + C$.

Kosten, ein Objekt einzustellen (C_I): Auch zum Einstellen eines Objekts muss der zuständige Knoten gefunden werden und das Objekt zu ihm übertragen werden. Hinzu kommt noch die Replikation der Daten auf die k Knoten, die dem zuständigen Knoten direkt folgen. Dies erhöht den Kommunikationsaufwand noch einmal um k Verbindungen, die ebenfalls mit den Kosten C angesetzt werden können. Insgesamt kostet dann das Einstellen eines Objekts $C_I = \mathcal{O}(\log N) + (1 + k)C$.

Kosten, ein Objekt zu entfernen (C_D): Da die Speicherung von Objekten in DHash dynamisch geschieht, brauchen keine aktiven Aktionen unternommen werden, um ein Objekt zu löschen. Ein simples Stoppen der Auffrischung des Objekts reicht aus. Somit gilt für die expliziten Kosten $C_D = 0$, weil die Löschung implizit durch die *garbage collection* erfolgt.

Kommunikationsaufwand, wenn ein Knoten dem Netz beiträgt (C_J): Hier müssen zwei Fälle unterschieden werden: Zum einen der nur im Paper zu Chord [52] erwähnte *simple* Join, zum anderen der auf realistischen Einsatz hin entwickelte, weil parallele Joins berücksichtigende *concurrent* Join. Der einfache Join-Ansatz ist vom Aufwand her höher, da er alle für das Hinzukommen nötige Datenstrukturen sofort aktualisiert. Der *concurrent*-Ansatz hingegen verläßt sich auf das ohnehin periodisch erfolgende Stabilisieren, und initialisiert nur die wichtigen Strukturen, nämlich Vorgänger- und Nachfolgerzeiger.

Addiert man für den herkömmlichen Ansatz die Kommunikationskosten, so ergibt sich folgende symbolische Rechnung:

$$\begin{aligned}
 C_J = \text{join}() &= \text{init_finger_table}(n') + \text{update_others}() = \\
 &= [\text{find_successor}() + 2 \cdot \text{successor.predecessor} + (m - 1)\text{find_successor}()] + \\
 &+ m[\text{find_predecessor}() + \text{update_finger_table}()] = \\
 &= [\mathcal{O}(\log N) + 2C + (m - 1)\mathcal{O}(\log N)] + m[\mathcal{O}(\log N) + \mathcal{O}(\log N)] = \\
 &= 2C + 3m\mathcal{O}(\log N) = \boxed{2C + 3\mathcal{O}(\log^2 N)}
 \end{aligned}$$

Im *concurrent*-Fall ist die Rechnung viel einfacher: Es muss genau *eine* Knotenlokalisierung durchgeführt werden, und der Knoten gilt als initialisiert. Dann gilt also $C_J = \mathcal{O}(\log N)$.

Kommunikationsaufwand durch einen das Netz verlassenden Knoten (C_L): Ein bewußtes Verlassen des Chord-Rings beinhaltet zwei Dinge: Das Weitergeben aller in der DHash-Schicht gespeicherten Daten an den Nachfolger, und das Benachrichtigen des Vorgängers und des Nachfolgers über den Ausfall. Während der Aufwand für Datenweitergabe von der Menge der bereits gespeicherten Blöcke abhängt und sich damit einer allgemeinen Betrachtung entzieht, ist die Arbeit auf der Routing-Schicht leicht erfaßbar: Es müssen genau zwei Nachrichten verschickt werden, es ist also $C_L = 2$.

Anzahl der Knoten, die nach Join/Leave ihre Struktur auffrischen (N_J, N_L): Auch hier ist eine Unterscheidung in simple- und concurrent-Varianten gegeben: Während die simple Variante wirklich alle Finger Tabellen aller Knoten auffrischt, die auf den neu hinzugekommen Knoten zeigen müssen, verlässt sich die concurrent Variante auf die Stabilisierung. Für die simple Variante gilt folgende symbolische Rechnung:

$$\begin{aligned}
 N_J = \text{join}() &= \text{init_finger_table}() + \text{update_others}() = \\
 &= [\text{find_successor}() + 2 \cdot \text{successor.predecessor} + (m - 1)\text{find_successor}()] + \\
 &+ m[\text{find_successor}() + \text{update_finger_table}()] = \\
 &= m \cdot \mathcal{O}(\log N) = \boxed{\mathcal{O}(\log^2 N)}
 \end{aligned}$$

Die concurrent-Variante frischt außer dem eigenen Nachfolgerzeiger überhaupt keine Datenstrukturen auf, also ist dafür $N_J = 0$. Beim Verlassen müssen zwei Knoten benachrichtigt werden: Der Vorgänger und der Nachfolger, also $N_L = 2$.

Optimierung: Caching entlang des Lookup-Pfades

Da die Erfahrung zeigt, dass Anfragen verschiedener Knoten auf die gleiche ID zum Ende der absolvierten Routing-Pfade gegeneinander konvergieren, profitiert das System in der Zahl der Hops davon, die einmal angefragten Daten auch auf den schon berührten Knoten zwischenspeichern. Neben der Reduktion der Hopzahl findet auch eine Entlastung derjenigen Knoten statt, die häufig zugriffene Objekte beherbergen. Solche Knoten werden in der Literatur oft mit dem Begriff *hot spots* bezeichnet. Diese Zwischenspeicherung findet nicht auf Chord-Ebene statt (weil Chord keine Datenblöcke kennt), sondern auf der darüberliegenden DHash-Schicht.

Determinismus An keiner Stelle der Chord-Algorithmen fließt Zufall in die getroffenen Entscheidungen mit ein. Alle Programmschritte beruhen auf dem lokalen Knotenzustand und den getätigten Eingaben (hier den Netzwerknachrichten). Deshalb ist das Verhalten jedes einzelnen Knotens in jeder Situation vorhersagbar, und wir können Chord als *deterministischen* Peer-to-Peer Algorithmus bezeichnen.

Praktische Erfahrungen Der Code zu Chord ist vom CVS-Repository am MIT auszuchecken. Die Erfahrungen des Autors sind, dass an dem Code während der Erstellung dieses Dokuments intensiv gearbeitet wurde: Fast täglich waren Änderungen im Projekt zu verzeichnen. Obwohl hauptsächlich auf OpenBSD [36] entwickelt wird, hat das MIT noch nicht auf die aktuelle Version dieses Betriebssystems umgestellt. Auch wurde auf die Einsendung von Patches kaum reagiert, obwohl einige Monate später von den Chord-Forschern genau die vorgeschlagenen Änderungen selbst realisiert wurden.

Chord ist darüberhinaus abhängig von den Quellen des SFS (Self-certifying Filesystem), das ebenfalls am MIT entwickelt wurde. Eine Kompilation von Chord setzt also auch ein Kompilieren von SFS voraus. Bei beiden Quellcodebäumen waren Anpassungen nötig, um das Projekt überhaupt zum Bauen zu bewegen.

Anwendungen Chord und DHash eignen sich, um folgende Anwendungen zu realisieren: Filesystem (wegen verteilter und redundanter Blockspeicherung), verteilter Nameserver (wegen Im-

plementation einer flachen Namenshierarchie und Abbildbarkeit des heute verwendeten DNS⁶ in eine Schlüssel-Wert-Beziehung, wie sie in Chord realisiert wird), transaktionsorientiertes Filesystem oder Dateiarchiv (vgl. Ivy, Abschnitt A.1).

Chord legt keinen Wert auf Anonymität. Zwar wird im Paper darauf hingewiesen, dass solche Eigenschaften als Schicht über Chord und DHash implementiert werden könnten. Dieser Aussage wird allerdings durch die Fachliteratur widersprochen. SCHNEIER schreibt in seinem Buch *Secrets & Lies* [47], dass Anonymität und Sicherheit kein Produkt, sondern ein Prozess sind, und deshalb bei der Entwicklung von Software von Anfang an mit eingeplant sein müssen.

Nachteile Die DHash-Funktion `put_s()` muss eine übergebene kryptographische Signatur prüfen, was im Verhältnis zu den sonstigen Operationen einen überdurchschnittlichen Rechenaufwand bedeutet. Weil diese Funktion von jedem Knoten aus aufgerufen werden kann, ergeben sich auch wieder Möglichkeiten für Denial-of-Service Attacken. Ein Angreifer kann nämlich mit relativ wenig Aufwand und geringer Netzwerkbandbreite auf dem Zielknoten eine verhältnismäßig hohe Rechenlast erzeugen.

⁶Domain Name Service, definiert in RFC 1034 und 1035 [32, 33].

4.2 CAN

SYLVIA RATNASAMY von der Universität Berkeley in Kalifornien benutzt den Ausdruck *CAN* nicht nur als Bezeichnung für ein von ihr geschaffenes Content-Addressable Network, sondern auch als einen Oberbegriff für verteilte Hashtabellen¹. Der Algorithmus entstand im Rahmen einer Doktorarbeit von Frau RATNASAMY [44], und es existieren noch einige Papers, die das System auf Konferenzen präsentieren. Es ist der Arbeit zu entnehmen, dass zunächst bestehende Systeme begutachtet wurden, bevor CAN entstand. Dadurch tauchen bereits bekannte Konzepte anderer Systeme wie Chord [31] oder YOID [16] in CAN ebenfalls wieder auf. Im Gegensatz zu den anderen untersuchten Peer-to-Peer Systemen geht CAN nicht tiefer auf die redundante verteilte Datenspeicherung ein. Wir werden zeigen, dass es sich bei CAN um einen **strukturierten, deterministischen und homogenen** Algorithmus handelt.

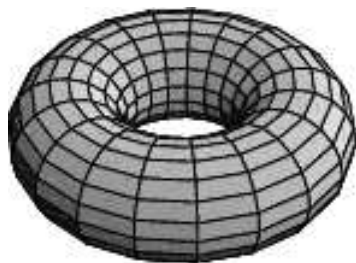


Abbildung 4.2.7: Ein zweidimensionaler Ringtorus als dreidimensionales Modell

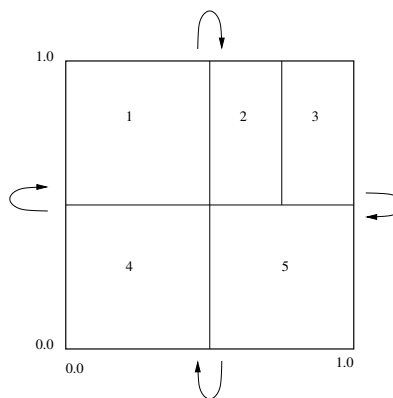


Abbildung 4.2.8: Ein zweidimensionaler Torus (flach) mit fünf Volumeneinheiten

Systemmodell

Als *Overlay Network* im Sinne von 3.2 für die Bilder der Hashfunktionen fungiert ein d -dimensionaler Torus (Beispiel in Abbildung 4.2.7), also ein d -dimensionales endliches kartesisches Koordinatensystem², das aber nicht unendlich groß ist. Die Endlichkeit tritt durch modulare Rechnung in jeder Koordinate auf. Dieser Raum wird unter den teilnehmenden Knoten in idealerweise gleich großen Volumeneinheiten aufgeteilt, was in Abbildung 4.2.8 veranschaulicht ist. Der Einfachheit halber wird im Folgenden bei Beispielen und Erläuterungen immer das Modell des 2-Torus herangezogen, das sich auch wie gesehen einfach in einem zweidimensionalen Bild darstellen lässt.

Jeder Knoten ist für Lookups in seiner (im folgenden als *Zone* bezeichneten) Volumeneinheit zuständig. Die Objekte werden als Schlüssel-Wert-Paare $\langle \vec{k}, v \rangle$ an den Koordinaten im Torus abgelegt, die durch den Schlüssel, bestehend aus einem oder mehreren Hashwerten, bezeichnet werden. CAN legt die Zuordnung von Schlüsseln zu Objekten in keiner Weise fest, außer, dass sie deterministisch und gleichverteilt zu erfolgen hat. In der Arbeit von Frau RATNASAMY wird also keine spezielle Hashfunktion angegeben, mit der die Zuordnung stattzufinden hat. Gleich mit welcher

¹Im Englischen gerne mit DHT (Distributed HashTable) abgekürzt.

²Der Raum selbst wird in RATNASAMYS Paper ebenfalls als CAN bezeichnet, was zu humorvollen Überschriften wie „Routing in a CAN“ führt.

Hashfunktion der Algorithmus ausgestattet wird, vom Prinzip her realisiert auch CAN eine verteilte Hashtabelle und darf deshalb als *strukturierter* Algorithmus gelten.

Eine Besonderheit bei CAN ist, dass zwar Knoten und Objekte den selben Adreßraum (das CAN) benutzen, aber dass die Objekte Punkte im Adreßraum sind, die Knoten aber nicht durch einen Punkt, sondern nur durch ein Teilvolumen beschrieben werden können.

Jeder Knoten erhält eine Nachbarschaftsbeziehung zu den im Raum unmittelbar benachbarten Knoten. Ein Knoten muss aufgrund der modularen Anordnung im Torus bei einer Dimension d des Raumes $2d$ Knoten als seine Nachbarn speichern. Jeder Knoten in CAN erfüllt die selben Aufgaben und stellt die selbe Funktionalität bereit, also handelt es sich bei CAN um einen *homogenen* Algorithmus.

Deterministische Zonenteilung

Beim Hinzukommen muss dem neuen Knoten ein Teil des Gesamtvolumens im d -Torus zugeteilt werden. Hierzu muss zunächst einmal eine bereits vorhandene Zone geteilt werden, da ja im konsistenten Zustand des CANs das gesamte Volumen bereits auf teilnehmende Knoten verteilt ist. Um später beim Wegfall einzelner Knoten die Zonen wieder in eindeutiger Weise zu größeren Fragmenten zusammenfassen zu können, ist die Teilung als deterministisches Verfahren beschrieben: Zunächst wird auf den d Koordinaten eine Ordnung definiert. Jede zu teilende Zone wird zuerst nach der ersten Variable, ihre Fragmente dann nach der zweiten, die daraus resultierenden Teile wieder nach der dritten Variable etc. geteilt.

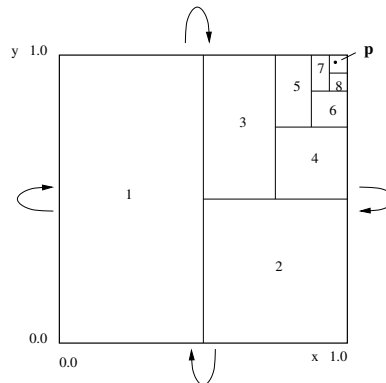


Abbildung 4.2.9: Veranschaulichung des deterministischen Verfahrens zur Zonenteilung am Beispiel eines 2-Torus: Achtmalige Teilung am Punkt \vec{p} .

Beispiel 4.2.1 (CAN-Zonenteilung) Wie in Abbildung 4.2.9 veranschaulicht, nehmen wir als gegebenen Raum einen zweidimensionalen Torus mit den Koordinatenachsen x und y an. Hierbei soll die Variablenordnung $x \succ y$ gelten. Das bedeutet, dass bei einer Teilung des Gesamtraums im ersten Schritt nach der ersten Variable, x , geteilt werden muss. Die Teilung halbiert den ursprünglichen Raum bei der Hälfte, gemessen in der Variable x , so dass zwei Teile entstehen: Teil 1 (links) und der Rest. Den Rest teilen wir zur Veranschaulichung des Verfahrens iterativ weiter: Beim Teilen an der jetzt anstehenden zweiten Variable y . Hierdurch entsteht der Teil 2 und der Rest, der wieder nach x in 3 und einen Rest geteilt wird, der wieder...

Selbstverständlich ist dieses Beispiel für die Praxis eine sehr ungünstige Art der Aufteilung des Gesamtraums, da wir aus verschiedenen Gründen Wert auf in etwa gleich große Fragmente legen.

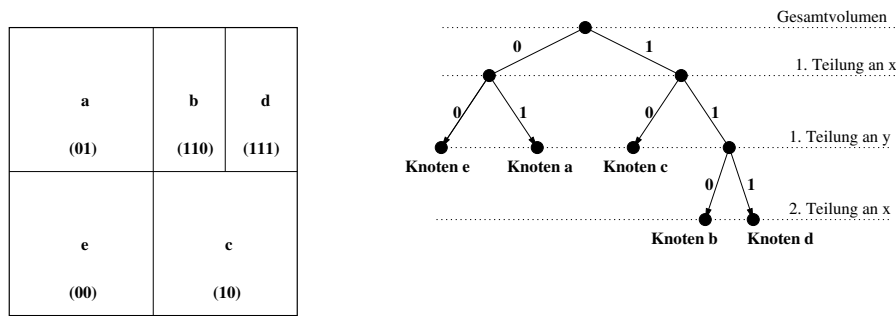


Abbildung 4.2.10: Zontenteilung im 2-Torus als Binärbaumdarstellung in CAN

Um die entstehenden Zonen algorithmisch effizient verarbeiten zu können, wird in der Arbeit zu CAN eine Darstellung als Binärbaum verwendet: Die oberste Ebene des Baumes repräsentiert eine Teilung des Gesamtraums in zwei Hälften, und zwar entlang der in der Variablenordnung höchsten Variable. Die zweite Baumebene teilt die verbleibenden Flächen entlang der zweiten Variable, usw. Die Darstellung im Binärbaum wird in Abbildung 4.2.10 veranschaulicht.

Zustandsspeicherung

Jeder Knoten kennt die Zone, die er lokal hält. Die Zone wird durch eine binäre Identifikation gekennzeichnet, aus der die Lage der Zone im Torus und die Größe der Zone deterministisch ausgerechnet werden können. Diese Identifikation ist exakt der Binärstring, der im Zonenteilungsbaum (Abbildung 4.2.10) zum Erreichen des Knotens nötig ist. Praktischerweise korrespondiert die Größe einer Zone reziprok mit der Länge der ID: Je länger die ID, desto öfter wurde die Zone bereits geteilt und desto kleiner ist sie folglich. Für das Volumen H_n der Zone des Knotens n gilt also:

$$H_n = \frac{H_{ges}}{2^{|id(n)|}}; \quad n \in V$$

Jeder Knoten hält lokal die Information über seinen numerisch niedrigeren und höheren Nachbarn für jede der d Dimensionen des d -Torus. Als algorithmisches Konstrukt können wir also die Variable `neighbor[i][j]` einführen, wobei die erste Variable die Dimensionen aufzählt ($i \in [d]$), und die zweite Variable zwischen niedrigerem und höherem Nachbarn entscheidet ($j \in \{0, 1\}$). Jedes Feld der `neighbor`-Variable hat folgende Komponenten:

Komponente	Beschreibung
<code>vid</code>	Die binäre ID, die angibt, über welche Zontenteilung die entsprechende Zone entstanden ist.
<code>coord</code>	Die Koordinaten des Volumenmittelpunkts der Zone. Diese Daten sind redundant, da sie sich aus der VID errechnen lassen. Es ist für das Funktionieren des Algorithmus völlig unerheblich, ob diese Komponente als aktuell gehaltene Variable oder als Funktion realisiert wird, da auf sie ohnehin nur lesend zugegriffen wird.
<code>address</code>	Für das wirkliche Kontaktieren des Nachbars ist eine Netzwerkadresse nötig (z. B. IP). Diese wird hier gespeichert.

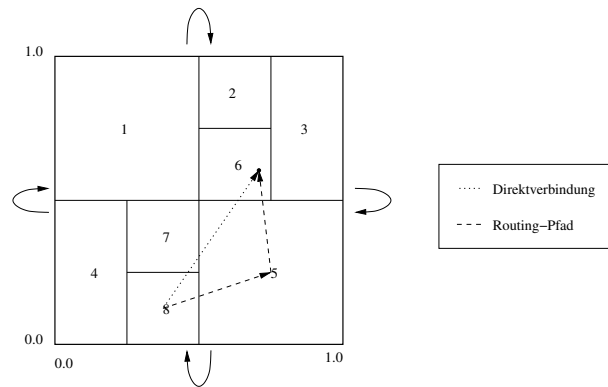


Abbildung 4.2.11: Beispiel eines Routing-Verlaufs im zweidimensionalen Torus (flach)

Der Lokalisierungsalgorithmus

Soll nun der für einen bestimmten Punkt \vec{p} im Suchraum zuständige Knoten gefunden werden (weil dieser beispielsweise zuständig für die mit \vec{p} assoziierten Daten einer Anwendung ist), so wird ausgehend von einem beliebigen Startknoten der Weg in die Zielzone durch gieriges Routing gefunden: die Koordinaten aller $2d$ Nachbarzonen des Startknotens werden überprüft und die Anfrage in die Zone weitergeleitet, die bezogen auf die Zielkoordinaten \vec{p} den größten Fortschritt an zurückgelegter Wegstrecke im Koordinatenraum bringt (Abbildung 4.2.11). Um zu wissen, was die größte Wegstrecke ist, muß wieder ein Entfernungsbegriff definiert werden:

Definition 4.2.1 (Entfernungsbegriff in CAN) Die Entfernung zweier Punkte \vec{x} und \vec{y} in einem d -Torus ist die EUKLIDISCHE METRIK:

$$d(\vec{x}, \vec{y}) := \left(\sum_{i=1}^d (x_i - y_i)^2 \right)^{\frac{1}{2}}$$

Weil man zum Messen der zurückgelegten Entfernung eine Verbindung zwischen zwei Punkten braucht, ist noch ein Weg nötig, den Ort einer Zone auszudrücken. Da der gierige Algorithmus der geradlinigen euklidischen Direktverbindung folgen möchte, bietet sich hierfür der Volumenschwerpunkt der Zonen an, der aufgrund der rechteckigen Form bequem durch Bildung des Mittelwertes in jeder Koordinate gefunden werden kann. Die Prozedur `lookup()` (Algorithmus 4.2.1) wird

Algorithmus 4.2.1 $n.\text{lookup}(\vec{p})$

Stellt her: Findet den für den Punkt \vec{p} zuständigen Knoten im CAN.

```

 $t_1 \leftarrow n;$ 
 $t_2 \leftarrow n;$ 
repeat
   $t_1 \leftarrow t_2;$ 
   $t_2 \leftarrow t_2.\text{bestpath}(\vec{p});$ 
until  $t_1 = t_2;$ 
return  $t_1;$ 

```

mittels der Funktion `bestpath()` (Algorithmus 4.2.2) solange von einem Knoten zum nächsten weiterleiten, bis eine Zone gefunden ist, deren Mittelpunkt der gesuchten ID sehr nahe liegt.

Algorithmus 4.2.2 $n.bestpath(\vec{p})$

Stellt her: Liefert den für den Punkt \vec{p} besten Nachbarn im CAN zurück.

```

minvertex  $\leftarrow n$ ;
mindistance  $\leftarrow d(n.coord, \vec{p})$ ;
for all  $i \in [d]$  do
  for all  $j \in \{0, 1\}$  do
     $t \leftarrow d(n, neighbor[i][j].coord)$ ;
    if  $t < mindistance$  then
      minvertex  $\leftarrow neighbor[i][j]$ ;
      mindistance  $\leftarrow t$ ;
    end if
  end for
end for
return minvertex;

```

Bemerkung: In pathologischen Extremfällen (eine Zone sehr groß, Nachbarzonen extrem fragmentiert, gesuchter Punkt ganz am Rand der großen Zone) kann es vorkommen, dass der gesuchte Punkt noch nicht in der mit `bestpath()` gefundenen Zone liegt. In diesem Fall muss jeweils über die Zonen-ID die räumliche Ausdehnung der aktuellen Zone und ihrer Nachbarn bestimmt und zum Routing verwendet werden. Dieses Verfahren stellt auf jeden Fall sicher, dass der gesuchte Punkt in der Zielzone liegt. Allerdings ist der zugehörige Algorithmus schnell entwickelt und liefert für diese Arbeit keinen Gewinn, weshalb wir diesen Fall vernachlässigen.

Join und Leave

Will ein Knoten n dem Netzwerk beitreten, so benötigt auch er einen Erstkontakt³ w . Über diesen Erstkontakt kann n den für einen gewählten Startpunkt p zuständigen Knoten ermitteln, der anschließend aufgefordert wird, seine Zone zu teilen und eine Hälfte an den neuen Knoten n zu übergeben. In Wirklichkeit geschieht die Zonenteilung noch etwas aufwändiger: Die Prozedur `split()` prüft erst, ob in der Nachbarschaft nicht noch eine größere Zone als die eigene existiert, die dann geteilt werden könnte. Dieses Vorgehen dient aber lediglich dazu, die Fragmentation des Gesamttraums zu verringern und ist deshalb für unsere Diskussion uninteressant. Um den Startpunkt herauszufinden, kann entweder ein Startpunkt über die Netzwerkadresse des eigenen Knotens gefunden werden (deterministische Variante, Algorithmus 4.2.6) oder über die Anwendung von Zufallsgeneratoren (probabilistische Variante, Algorithmus 4.2.5).

Das Verlassen eines CANs kann von dem wegfallenden Knoten bekannt gegeben werden. In diesem Fall wird der Stabilisierung vorgegriffen und seine Zone sofort an einen der Nachbarknoten vergeben. Hier wird aus Lastverteilungsgründen darauf geachtet, dass der Nachbar mit der kleinsten Zone die aktuelle Zone übernimmt. Ist von der Baumstruktur der Teilung her ein Zusammenfassen der zwei Zonen möglich, so wird dies durchgeführt. Ansonsten hält der Knoten einfach beide Zonen.

³In CAN wird auf eine mögliche praktische Lösung des Erstkontaktproblems eingegangen. Hier wird auf YOID [16] verwiesen, die das Problem über DNS-Anfragen lösen, die von Knoten innerhalb des CANs beantwortet werden. Dieses Vorgehen wird im Kapitel 5.2 noch näher besprochen.

Algorithmus 4.2.3 $n.join(w)$

Benötigt: w ist ein Knoten, der bereits am CAN teilnimmt. $|k.vid|$ gibt die Länge des Binärstrings $k.vid$ an.

Stellt her: Fügt den Knoten n zum CAN hinzu.

```
 $\vec{p} \leftarrow \text{get\_start\_point}();$   
 $v \leftarrow w.\text{lookup}(\vec{p});$   
 $M \leftarrow \{v\} \cup \left( \bigcup_{i \in [d]} \bigcup_{j \in \{0,1\}} v.\text{neighbor}[i][j] \right);$   
 $s \leftarrow \{k \in M \mid |k.vid| \leq |w.vid| \ \forall w \neq k; \ w \in M\};$   
 $s.\text{split}();$   
 $n.vid \leftarrow s.vid \oplus 00 \dots 01;$ 
```

Algorithmus 4.2.4 $n.split()$

Benötigt: Eine Operation $\text{concat}()$ zum Anhängen einzelner Bits an einen Bitstring.

Stellt her: Teilt die Zone des Knotens n .

```
 $n.vid \leftarrow \text{concat}(n.vid, '0');$ 
```

Algorithmus 4.2.5 $n.get_start_point()$ (probabilistisch)

Benötigt: Ein guter⁴ Pseudozufallsgenerator oder echter Zufall aus $[k]$ als Funktion $\text{random}(k)$.

Stellt her: Liefert einen zufälligen Startpunkt für den $\text{join}()$ -Algorithmus.

```
 $j \leftarrow N^{\frac{1}{d}};$   
for all  $i \in [d]$  do  
     $p_i \leftarrow \text{random}(j);$   
end for  
return  $\vec{p} := (p_0, p_1, \dots, p_{d-1});$ 
```

Algorithmus 4.2.6 $n.get_start_point()$ (deterministisch)

Benötigt: Eine Menge $\{h_0, h_1, \dots, h_{d-1}\}$ von kollisionsfreien Hashfunktionen.

Stellt her: Liefert einen nachprüfbaren Startpunkt für den $\text{join}()$ -Algorithmus.

```
for all  $i \in [d]$  do  
     $p_i \leftarrow h_i(n.\text{address}) \bmod N^{\frac{1}{d}};$   
end for  
return  $\vec{p} := (p_0, p_1, \dots, p_{d-1});$ 
```

Tolerierte Fehler, Redundanz

Für die Beschreibung der Fehlertoleranz und Stabilisierung ist die Definition der Konsistenz wesentliche Voraussetzung:

Definition 4.2.2 (Konsistenz in CAN) *Ein CAN ist konsistent, wenn zu jedem Zeitpunkt der gesamte Koordinatenraum dynamisch auf die teilnehmenden Knoten aufgeteilt ist, so dass jeder Knoten seine eigene Zone im Gesamtraum „besitzt“ und außerdem jeder Knoten seine 2d Nachbarn im Torus kennt.*

Ein einzelner Knoten kann prinzipiell den Ausfall aller seiner Nachbarn bis auf einen tolerieren, da über diesen Kontakt wieder alle Verbindungen und Nachbarschaftsbeziehungen wieder hergestellt werden können.

Um die Redundanz der Verbindungen zu erhöhen, schlägt Frau RATNASAMY das Konzept der *Realitäten* vor. Der Faktor r vervielfacht die Anzahl der im System existierenden Tori. Jeder Knoten hat in jedem Torus seine Zone, für die er zuständig ist. Dieser Ort liegt idealerweise in jeder Realität woanders. Muss nun ein Routing zu einem bestimmten Punkt \vec{p} erfolgen, so kann dies in jeder Realität geschehen. Der Knoten kann jeweils diejenige Realität benutzen, in der seine Zone dem Ziel am nächsten liegt. Durch die Einführung von Realitäten erhöht sich die Anzahl der Verbindungen linear auf $2dr$, und die Wahrscheinlichkeit eines Totalausfalls aller Nachbarn sinkt exponentiell. Dieses Verfahren zur Erhöhung der Redundanz bewirkt allerdings bei einer eventuellen Objektspeicherung, dass jeder Knoten um den Faktor r mehr Objekte bedienen muss als vorher.

Eine weitere Schaffung von Redundanz schlägt die Autorin vor, wenn Sie die Existenz mehrerer Knoten pro Zone einführt. Hierdurch übernehmen alle m Knoten einer Zone die selben Aufgaben, und ein Knoten kann den anderen Knoten bei Ausfall ersetzen. Außerdem kann durch die Auswahlmöglichkeit beim Weiterleiten einer Nachricht in eine andere Zone die physikalische Nähe berücksichtigt werden, und so der Weg mit der geringsten Netzwerklatenz oder der geringsten Zahl physikalischer Hops gewählt werden.

Stabilisierung

Die Stabilisierung in CAN erfolgt explizit und auf zwei Arten: Einerseits schickt jeder Knoten periodisch Nachrichten an seine Nachbarn, dass er noch online ist. Beim Ausbleiben solcher Nachrichten wird von den Nachbarn dann eine Übernahme der Zone initiiert.

Zum anderen läuft im Hintergrund ein Algorithmus, der für die Defragmentierung der Zonenstruktur im Torus sorgt. Die Fragmentierung tritt durch aufeinanderfolgende Join- und Leave-Operationen auf und wird durch die Variablenordnung verursacht. So kann es beispielsweise sein, dass eine freiwerdende Zone nicht mit ihrer Nachbarzone gleicher Größe vereinigt werden kann, weil deren gemeinsame Grenze an der falschen Koordinatenachse verläuft. Zusammen mit einer hohen Fluktuation von Teilnehmern leidet schnell die Struktur des Systems.

Verwaltungsaufwand

Im Gegensatz zu anderen Algorithmen wird in CAN beim Routing nicht logarithmisch gesprungen, sondern es werden nur die Nachbarschaftsbeziehungen benutzt. Deshalb gestaltet sich die Komplexitätsanalyse etwas einfacher als bei den anderen Systemen.

⁴Ein Buch mit einem ganzen Kapitel über gute Zufallsgeneratoren erscheint demnächst: Die Dissertation von PETER GUTMANN [22, Kapitel 6]

Finden eines Knotens in CAN (C_F): Um einen Knoten zu finden, wird der im Koordinatensystem kürzeste Weg gewählt. Will man die durchschnittlichen Kosten einer Lokalisierung herausfinden, so kann man diese Kosten mit der mittleren zurückgelegten Weglänge bei der Kommunikation zweier beliebiger Knoten gleichsetzen. Die mittlere Weglänge kann in Abhängigkeit der Dimension d und der aktuellen Anzahl N der Knoten im System formuliert werden:

$$C_F = \frac{d}{4} N^{\frac{1}{d}} = \mathcal{O}(dN^{\frac{1}{d}})$$

Größe der Verwaltungsstrukturen jedes Knotens (D_V): Jeder Knoten hält die Information über seine $2d$ Nachbarn. Wir nehmen wie bei den anderen untersuchten Systemen an, dass die Adreßinformation in der Größenordnung weniger Maschinenwörter kodiert werden kann und geben deshalb $D_V = 2d$ wie bei den anderen Analysen als Vielfaches dieser Größe an.

Kosten, ein Objekt zu lesen (C_R): Das Lesen eines Objekts beschränkt sich auch bei CAN auf das erfolgreiche Finden des zuständigen Knotens (C_F) und einer Konstante C für die Datenübertragung, so dass $C_R = \mathcal{O}(dN^{\frac{1}{d}}) + C$.

Kosten, Objekte einzustellen oder zu entfernen (C_I, C_D): Leider wird in RATNASAMYS Arbeit nicht detailliert auf die Verfahren zur Objektspeicherung eingegangen. Stattdessen beschränkt sie sich auf das Routing, so dass höchstens Mutmaßungen über redundanten Speicheraufwand und Umfang von Löschaufträgen gegeben werden könnten. Deshalb lassen wir die Variablen C_I und C_D für CAN unbenutzt.

Kommunikationskosten für Join/Leave Bei einem Hinzukommen müssen lediglich die Nachbarn des neuen Knotens über die Veränderung unterrichtet werden, weitere Verbindungen existieren nicht. Pro verwendeter Realität sind dies $2d$ Nachbarn. Also ist $N_J = 2dr$ und $C_J = 2dr$. Ebenso verhält es sich mit dem kontrollierten Verlassen des Systems: Auch hier werden alle Nachbarn verständigt, also auch $N_L = 2dr$ und $C_L = 2dr$.

Gesamteindruck

Positiv fällt die umfangreiche Herangehensweise auf: Das Papier problematisiert zunächst den status quo, und präsentiert dann als logische Schlussfolgerung und Lösung zu jedem kleinen Schritt einen weiteren Baustein von CAN. CAN eignet sich aufgrund des Torus als Systemmodell gut für Multicast-Anwendungen, da ein kontrolliertes Fluten von Nachrichten durch die Zonenstruktur begünstigt wird. Tatsächlich existiert eine solche Anwendung, die RATNASAMY auch in ihrer Dissertation vorstellt (siehe Anhang A).

Die Arbeit präsentiert darüberhinaus viele Lösungsansätze, wie Redundanz und kurze Routingpfade erreicht werden können. Leider sind es zu viele — die angedachten Methoden überschneiden sich teilweise in ihrer Wirkung und werden auch nicht in einem sinnvollen Gesamtkonzept präsentiert. Vielmehr liegen sie in der Doktorarbeit immer noch als Stoffsammlung für den Leser bereit.

Beim Analysieren des Algorithmus sind auch einige Schwachstellen bemerkbar geworden:

Frei wählbarer Startpunkt: Dass ein neu hinzukommender Knoten seinen Startpunkt \vec{p} für die Zonenteilung vollkommen frei wählen kann, ist eine Möglichkeit für Denial-of-Service-Angriffe. Ist durch einseitiges Joinverhalten (mutwillig oder durch einen schlechten Pseudozufallsgenerator)

ein Ungleichgewicht gegeben, so erzeugen aufeinanderfolgende Joins mit dem selben Startpunkt \vec{p} sehr kleine Zonen, die den Verwaltungsaufwand sowie die Zahl der Hops zwischen den Zonen immens erhöhen. Der Effekt wurde beim Design zwar bedacht, aber der vorgeschlagene „1-hop volume check“ (das Prüfen der Nachbarzonengrößen vor der Teilung) hilft nur bedingt, indem er den starken Split der Zonen auf die jeweiligen Nachbarzonen mitabwälzt. Laut den Messungen in der CAN-Simulation erreicht man mit diesem Verfahren im Normalbetrieb eine sehr gute Nivellierung der Zonenvolumen. Bei böartigem Verhalten ist dies aber kein effektiver Schutz.

Polynomialer Aufwand Wie schon bei der Aufwandsbetrachtung erwähnt, hat CAN bei der Lokalisierung einen polynomialen Aufwand in der Zahl der Teilnehmer, während andere Algorithmen mit einem logarithmischen Aufwand auskommen. Die Arbeit suggeriert, dass mit hinreichend hoher Zahl d der Dimensionen der Aufwand in etwa gleich ist. Trotzdem ist d für ein konkretes CAN eine Konstante. Ohne praktische Tests (die nicht Gegenstand dieser Arbeit sind) kann aber nichts darüber ausgesagt werden, ob diese Tatsache wirklich zum Problem wird. Es kann durchaus sein, dass der polynomiale Aufwand für konkrete Probleme bezahlbar ist.

Fragmentation und Stabilisierung des Zonenraums Das Modell des d -Torus mag viele Möglichkeiten bieten, Parameter zu verändern, die aufgeprägte deterministische Zonenteilung jedoch begünstigt Fragmentierung der Zonenstruktur. Die Nennung eines expliziten, im Hintergrund laufenden Defragmentierungsalgorithmus sowie die Tatsache, dass *jeder* Knoten im Normalbetrieb periodisch Kontrollnachrichten an seine Nachbarn verschickt, lassen auf eine hohe Aktivität des Systems auch im Ruhezustand schließen. Im Vergleich hat CAN am meisten Vorkehrungen, die ohne konkreten Anlaß Nachrichten verschicken, so dass die Stabilisierung als sehr aufwändig erscheint.

4.3 PLAXTONS Algorithmus

Dieser Algorithmus wird nicht selbst für konkrete Implementationen verwendet, da er für die praktische Anwendung zu viele Freiheitsgrade lässt und sehr allgemein formuliert ist. In seiner Theorie ist er aber grundlegend für Pastry und Tapestry. Deshalb werden hier die für die beiden Algorithmen relevanten theoretischen Punkte herausgearbeitet.

PLAXTONS Algorithmus konzentriert sich auf das Problem, aus einer Menge von Kopien von verteilten Objekten jeweils das zum aktuellen Knoten nächste Objekt zuzugreifen. Hierbei soll ein schneller Zugriff ebenso wie eine optimale Nutzung der vorhandenen Netzwerkressourcen hergestellt werden. Die Studie beschränkt sich auf das Lesen von verteilten Objekten und lässt auch die Einführung von redundanter Speicherung sowie das Problem aller praktischen Algorithmen, das verteilte Speichern und die konsistente Haltung der Kopien, außen vor. Das Netzwerk aller Knoten wird als gegebene, statische Struktur betrachtet, deshalb werden auch Join und Leave bei PLAXTON nicht behandelt.

Definition 4.3.1 (Konsistenz in PLAXTONS Algorithmus) *Die Konsistenz ist in Plaxton mangels Join, Leave oder Ausfall von Knoten im Modell immer gegeben.*

Der Algorithmus geht von einer bereits vorhandenen, zuverlässigen Kommunikation zwischen den Knoten aus.

Zunächst ist es wichtig, die Nomenklatur der von PLAXTON verwendeten Begriffe zu erklären. Hierzu sei auf Tabelle 4.3.2 verwiesen. Der Algorithmus setzt voraus, dass die Anzahl der Objekte

Symbol	Erläuterung
$\mathcal{A} = \{A, B, \dots\}; \quad \#\mathcal{A} = m$	Die Menge aller Objekte, die im System gespeichert werden können. Die Mächtigkeit der Menge wird als m bezeichnet. Jedes Objekt $A \in \mathcal{A}$ hat eine eindeutige $\lceil \log m \rceil$ bit ID, die synonym mit dem Objekt gebraucht wird. Um auf das i -te Bit zuzugreifen, verwenden wir die Notation A^i , $(0 \leq i \leq \lceil \log m \rceil - 1)$.
$G = (V, E); \quad \#V = N$	Das Netzwerk selbst mit der Knotenmenge V , die N Knoten besitzt. Jeder Knoten $x \in V$ erhält eine eindeutige ID aus $[N]$, wobei die IDs gleichmäßig über $[N]$ verteilt sein sollen. Auf allen Knoten wird eine Totalordnung $\beta : V \rightarrow [N]$ (bijektiv) definiert.
$c(x, y); \quad x, y \in V;$ $c : V \times V \rightarrow \mathbb{R}$	Die Kostenfunktion c , die die Kosten für die Übertragung eines Wortes vom Knoten x zum Knoten y berechnet (Ein <i>Wort</i> im Sinne des PLAXTON-Algorithmus ist ein Bitstring der Länge $\mathcal{O}(\log N)$ und dient als atomare Einheit zum Verschicken von Nachrichten oder Speichern von Objekten). c ist als Metrik zu sehen, da PLAXTON auch noch fordert, dass c symmetrisch und eindeutig ist, und die Dreiecksungleichung erfüllt (vgl. Anhang C).
$M(u, r) := \{v \in V \mid c(u, v) \leq r\}$ $\forall u \in V, \forall r \in \mathbb{R}$	Die Nachbarschaftsmenge des Knotens u mit dem Radius r . Insbesondere liegt auch u in dieser Menge.

Tabelle 4.3.2: Nomenklatur aus dem Paper von PLAXTON

höchstens polynomial in der Anzahl der Knoten ist, also $m = N^{\mathcal{O}(1)}$. Aus praktischen Gründen wird ausserdem noch vorausgesetzt, dass gilt $N = (2^b)^g$ mit $g \in \mathbb{N}$, dass also N eine beliebige Potenz von 2^b ist. Betrachtet man das System als Baum (vgl. Abbildung 4.3.12), so legt der Parameter b fest, wieviele Unterknoten ein Knoten besitzen kann, nämlich 2^b . Die Variable g gibt hingegen die maximale Tiefe des Baumes an. Die Kostenfunktion c dient dem Algorithmus zur Erzwingung der

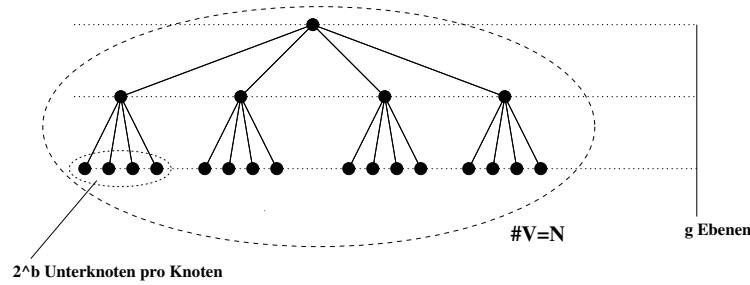


Abbildung 4.3.12: Das Systemmodell von Plaxton: Ein Baum, der alle $\#V = N$ Knoten beinhaltet. Weil $N = 2^{bg}$, gibt es pro Knoten 2^b Unterknoten und g Ebenen im Baum.

Lokalität beim Routing, so dass kein unnötiger Netzwerkverkehr erzeugt wird und als Konsequenz auch die Last durch Zugriffe auf die jeweils nächste Kopie des Objekts verteilt wird.

Definition 4.3.2 (Entfernungsbegriff in PLAXTONs Algorithmus) Als Entfernungsbegriff im Algorithmus dient die Metrik $c(x, y)$, die die Kosten für die Übertragung eines Wortes zwischen den Knoten x und y angibt.

Mit der Kostenfunktion kann errechnet werden, wie teuer es ist, eine Nachricht mit der Länge von l Worten vom Knoten u zum Knoten v zu verschicken:

$$f(l) \cdot c(u, v); \quad f : \mathbb{N} \rightarrow \mathbb{R}^+ \quad f(1) := 1$$

Diese Formel garantiert, dass das Verschicken einer Nachricht der Länge l mindestens so teuer ist wie das Versenden von l Einzelnachrichten. Mit der Funktion f kann noch auf besondere Gegebenheiten für größere Datenmengen hinsichtlich Kommunikationsoverhead eingegangen werden (etwa MTU¹ oder Fragmentierung).

Lokale Daten

Bei der Einteilung des lokalen Speichers auf einem Knoten wird wie bei anderen Peer-to-Peer Algorithmen unterschieden zwischen der Speicherung von Objekten und von Hilfsinformationen. Als Hilfsinformationen zählen hier speziell die *Zeigertabelle* und die *Nachfolgerliste*.

Die Zeigertabelle des Knotens n , genannt $Ptr(n)$, ist eine Sammlung von Zeigern auf Knoten, die Kopien von Objekten halten. Sie besteht aus einer Menge von Tupeln der Form $\langle A, x, k \rangle$. $A \in \mathcal{A}$ bezeichnet die Kopie eines Objektes A , x ist der Knoten, der die Kopie hält, und k ist eine obere Schranke für die Kosten $c(n, x)$ der Kommunikation zwischen lokalem und entferntem Knoten.

Die Nachbarliste besteht aus drei Tabellen mit jeweils $(\log N)/b = (\log 2^{bg})/b = g$ Reihen. Die Tabellen beinhalten sog. *primäre*, *sekundäre* und *umgekehrte* (i, j) -Nachbarn. Das Tupel (i, j) wird zum Indizieren eines Tabellenfeldes benutzt, und es gilt $i \in [g]; j \in [2^b]$, wodurch jede Tabelle $g \cdot 2^b$ Einträge besitzt. Hierbei hält ein Feld in der primären Tabelle genau *einen* Knoten, die beiden anderen Tabellen speichern *Mengen* von Knoten. Die in der Nachbarliste aufgeführten Knoten zeichnen sich durch eine bitweise Übereinstimmung bestimmter, für die Reihe spezifischer

¹Maximum Transmission Unit, siehe RFC 791 [40]

Länge mit der ID des eigenen Knotens aus. Hierzu werden für die Bits des Bitstrings x folgende Kurznotationen verwendet:

$$\begin{aligned} x &:= x^{(\log N)-1} x^{(\log N)-2} \dots x^1 x^0 \\ x^k &\text{ bezeichnet das } k\text{-te Bit von } x \text{ (von rechts, beginnend mit 0)} \\ x[i] &:= x^{(i+1)b-1} \dots x^{ib} \end{aligned}$$

$x[i]$ greift aus dem gesamten Bitstring x einen Teilstring der Länge b heraus. Der Parameter i bestimmt, an welcher Stelle der Teilstring herausgenommen wird. Es ist anzumerken, dass mit dieser Notation exakt soviele Teilstrings existieren, wie die Tabelle Reihen hat. Betrachtet man den String nun nicht als Binärzahl, sondern als Zahl zur Basis 2^b , so ist der Sinn von $x[i]$ völlig klar: Mit dieser Notation werden einzelne Ziffern zur Basis 2^b aus dem String gegriffen. Nun zum Inhalt der Nachbarliste:

Definition 4.3.3 (primärer (i, j) -Nachbar) Ein primärer (i, j) -Nachbar eines Knotens n ist derjenige bekannte Knoten y , **1.** für den gilt

$$y[k] = n[k]; \quad \forall k \in [i]$$

und **2.** der **(a)** entweder nicht auf der höchsten Ebene liegt, eine vollkommene Übereinstimmung mit dem Präfix der Tabellenspalte aufweist und für diese beiden Bedingungen die kürzeste Entfernung zu n besitzt (wenn ein solcher Knoten existiert):

$$\begin{aligned} K &:= \{l \in V \mid [(i < d - 1) \wedge (y[i] = j)]\} \\ y &:= \{k \in K \mid c(n, k) \text{ minimal}\} \end{aligned}$$

oder **(b)** der Knoten mit der höchsten ID $\beta(y)$ aus allen Knoten z mit der größten Übereinstimmung von $z[i]$ und j von rechts ist:

$$\begin{aligned} Z &:= \{z \in V \mid \max\{l \in \mathbb{N} \mid z^h = j \quad \forall h \in [l]\}\} \\ y &:= \{j \in Z \mid \beta(j) \text{ maximal}\} \end{aligned}$$

Die sekundären (i, j) -Nachbarn sind nun diejenigen nahen Knoten, die als primäre Nachbarn in Frage gekommen wären, aber nicht am kürzesten von n entfernt waren:

Definition 4.3.4 (sekundäre (i, j) -Nachbarn) Sekundäre (i, j) -Nachbarn sind die Knoten, die vollkommene Übereinstimmung im Präfix der Tabellenspalte bieten und höchstens d -mal so weit von x entfernt sind wie y , wobei $d \in \mathbb{N}$ ein systemweit konstanter Wert ist.

$$\begin{aligned} W_{i,j} &:= \{w \in V \mid w[k] = n[k] \quad \forall k \in [i] \quad \wedge \\ &\quad \wedge w[i] = j \quad \wedge \\ &\quad \wedge c(w, n) \leq d \cdot c(y, n)\} \end{aligned}$$

Es kann höchstens d sekundäre Nachbarn geben. Stehen mehr Kandidaten zur Auswahl, werden die d Knoten mit der geringsten Entfernung zu n ausgewählt. Die resultierende Menge ist eindeutig entscheidbar, weil die verwendete Metrik c per Definition ebenfalls eindeutig ist, d. h. es gibt keine zwei Knoten $a \neq b \in V$ für die gilt: $c(a, n) = c(b, n)$. Nun muss noch die dritte Art von Nachbarn definiert werden:

Definition 4.3.5 (umgekehrte (i, j) -Nachbarn) Als umgekehrte (i, j) -Nachbarn von n werden diejenigen primären (i, j) -Nachbarn von n bezeichnet, bei denen n ebenfalls als primärer (i, j) -Nachbar gilt.

Als nächstes wird eine eindeutige Zuständigkeit für das Wissen um Kopien von ins Netz gestellten Objekten definiert. Demnach muss mindestens der *Wurzelknoten* einen Zeiger auf das Objekt besitzen, für das er Wurzelknoten ist.

Definition 4.3.6 (Wurzelknoten) Als Wurzelknoten für ein bestimmtes Objekt $A \in \mathcal{A}$ wird derjenige Knoten $r \in V$ bezeichnet, dessen Identifikation von rechts gelesen am längsten bitweise mit der Identifikation des Objekts übereinstimmt:

$$r := \left\{ k \in V \mid l \in \mathbb{N} \text{ maximal für } \left\{ k^h = A^h \quad \forall h \in [l] \right\} \right\}$$

Bemerkung: Interessanterweise haben die Identifikationen von Knoten und Objekten nicht zwingendermaßen die selbe Bitlänge. Jedoch ist die Identifikation der Objekte zwingend mindestens so lang wie die der Knoten, da wir die Anzahl m der Objekte oben als polynomial in der Anzahl N der Knoten definiert haben: $m = N^{O(1)}$. Weil die Bits von rechts mit 0 beginnend indiziert werden, ist auch ein Vergleich einfach in einem Algorithmus auszudrücken.

Um den Weg einer Anfrage nach einem Objekt zu beschreiben, benutzt PLAXTON den Begriff der *Folge*. Eine Folge $\langle a \rangle_k$ ist ein Pfad im Graphen mit der Länge $k + 1$ und folgender Notation:

$$\langle a \rangle_k := a_0 a_1 a_2 \dots a_k$$

Ist der Wert von k eindeutig aus dem Zusammenhang erkennbar, so kann der Index auch weggelassen werden. Mit diesem Begriff in der Hinterhand kann die nächste Definition über den Pfad zum gewünschten Objekt (bzw. zum Zeiger darauf) folgen:

Definition 4.3.7 (primäre Nachbarfolge) Eine primäre Nachbarfolge für das Objekt $A \in \mathcal{A}$ ist diejenige maximale² Folge $\langle u \rangle_k$, so dass

1. $u_0 \in V$
2. u_k ist Wurzelknoten für A
3. u_{i+1} ist primärer $(i, A[i])$ -Nachbar von u_i für alle i .

Es gibt im Regelfall mehrere primäre Nachbarfolgen für A .

In PLAXTONS Arbeit wird noch bemerkt, dass durch die primäre Nachbarfolge folgende Übereinstimmung von Knoten auf der Folge und Objektidentifikation garantiert ist:

$$\forall u_i \in \langle u \rangle_k : \forall i : (u_i[i-1], \dots, u_i[0]) = (A[i-1], \dots, A[0])$$

Bei diesem Vergleich handelt es sich um Tupel mit jeweils i Einträgen, von denen jeder b Bit breit ist.

²Es wird (ebenso wie bei den Graph-Matchings in [54]) unterschieden zwischen einer *Maximalfolge* und einer *Maximumfolge*. Mit Maximalfolge ist die größte Folge gemeint, die jeweils einen bestimmten Knoten mit einschließt — für diesen bestimmten Knoten ist die Folge maximal. Der Begriff Maximumfolge drückt hingegen aus, dass es im Graphen keine längere Folge für A gibt.

Lesen von Daten

Der lesende Zugriff eines Knotens $x \in V$ auf ein Objekt $A \in \mathcal{A}$ geschieht nun folgendermaßen: Aus der primären Nachbarfolge für A , die sich durch x erstreckt, wird eine Teilfolge mit $x_0 := x$ betrachtet. Prinzipiell werden nun Anfragen nach dem Objekt A zum Wurzelknoten hin entlang der Nachbarfolge weitergeleitet. Beim Weitergeben der Nachricht informiert der Knoten x_{i-1} den Knoten x_i über die momentan beste obere Schranke k für die Kosten, eine Kopie von A zu x zu schicken (Anfänglich setzt der Knoten diese Schranke natürlich auf $k \rightarrow \infty$, da ihm beim ersten Anfordern des Objekts keine Kosten bekannt sind, und er deshalb jede Lösung hinsichtlich ihrer Kosten akzeptieren wird).

Bei Empfang einer Anfrage mit der zugeordneten oberen Schranke k verfährt ein Knoten wie folgt:

- Wenn x_i bereits der Wurzelknoten für A ist, fordert er an, dass die Kopie von A mit zugeordneten tatsächlichen Kosten k zu x geschickt wird. Der Wurzelknoten hält bekanntlich selbst nicht unbedingt eine Kopie von A , aber er kennt einen Knoten der dies tut, da er einen Eintrag hierfür in seiner Zeigertabelle $Ptr(x_i)$ hat.
- Sonst kommuniziert x_i mit seinen primären und sekundären $(i, A[i])$ -Nachbarn (veranschaulicht in Abbildung 4.3.13), um herauszufinden, ob die Zeigertabelle von irgendeinem dieser Knoten einen Eintrag $\langle A, z, k_1 \rangle$ besitzt, wobei z ein Knoten ist, der eine Kopie von A hält, und $k_1 \leq k$. Nach Erhalt der Antworten aktualisiert x_i sein lokales k aus dem Minimum aller Antworten und seiner bekannten Kosten:

$$k \leftarrow \min\{k_{prim}, k_{sec1}, k_{sec2}, \dots, k_{sec_d}, k_{x_{i-1}}\}$$

Liegt k danach noch in einem akzeptablen Verhältnis zum bisher zurückgelegten Weg, d. h.:

$$k = \mathcal{O}\left(\sum_{j=0}^{i-1} c(x_j, x_{j+1})\right),$$

dann fordert der Knoten x_i an, dass die gefundene Kopie von A an x geschickt wird. Sonst leitet x_i die Anfrage an x_{i+1} weiter.

Einfügen von Daten in das System

Nehmen wir an, Knoten x will eine Kopie von A , die er lokal hält, im Netz bekanntmachen. Die Updates der Verwaltungsstrukturen finden an der primären Nachbarfolge von A statt, die $x_0 = x$ als Startpunkt besitzt. Beim Empfang einer Nachricht mit dem Tupel $\langle A, x, k_{x_0} \rangle$ prüft der Knoten x_i , ob in seiner Zeigertabelle $Ptr(x_i)$ schon ein Eintrag für A vorhanden ist. Wenn dem nicht so ist, oder das neue Tupel geringere Kosten $k_{x_0} < k_{alt}$ aufweist, so wird das empfangene Tupel in die Zeigertabelle aufgenommen. Es überschreibt damit auch ein eventuell schon vorhandenes Tupel $\langle A, \cdot, \cdot \rangle$. Fand tatsächlich eine Veränderung dieser Art an der eigenen Zeigertabelle statt, so leitet der Knoten x_i die Nachricht auch noch an den nächsten Knoten auf der Nachbarfolge, x_{i+1} , weiter (falls es sich bei x_i nicht ohnehin schon um den Wurzelknoten für A gehandelt hat).

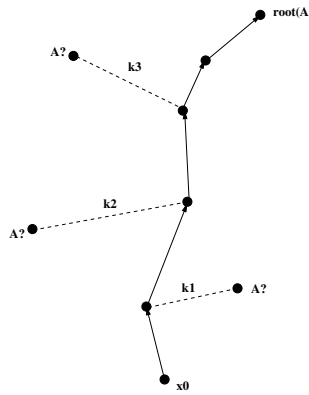


Abbildung 4.3.13: Primäre Nachbarfolge vom Knoten $x_0 := x$ aus zum Wurzelknoten von A . An den Seitenästen exemplarisch einige sekundäre Nachbarn und die Kosten k_1, k_2 und k_3 , um mit ihnen zu kommunizieren.

Löschen von Daten aus dem System

Wenn der Knoten x das Objekt A lokal nicht länger zur Verfügung stellen will, so müssen auch alle Verweise auf dieses Objekt entfernt werden. Weil derartige Zeiger durch die Natur des Einfügevorgangs nur entlang der primären Nachbarfolge liegen können, reicht eine Weiterleitung an dieser Folge zum Löschen aller Referenzen aus. Jeder Knoten, der eine Nachricht zum Löschen bekommt, entfernt das Tupel aus seiner Zeigertabelle. Nur, wenn eine Löschung stattgefunden hat, wird die Aufforderung an den nächsten Knoten weitergereicht, da sonst auch kein weiterer Knoten ein Tupel $\langle A, x, \cdot \rangle$ mehr besitzen kann.

Eine Besonderheit, die den Algorithmus robuster macht, ist die Suche nach einer Ersatzkopie: Löscht ein Knoten x_i das Tripel $\langle A, x, \cdot \rangle$ aus seiner Zeigertabelle, so sucht er bei seinem umgekehrten $(i - 1, A[i - 1])$ -Nachbarn, ob ein Eintrag $\langle A, y \neq x, k \rangle$ gespeichert ist und übernimmt $\langle A, x, k + c(y, x_i) \rangle$ in seine Zeigertabelle $Ptr(x_i)$.

Baumartige Struktur

Bei eingehender Untersuchung der nun dargelegten Eigenschaften des Algorithmus stellt man fest, dass eine Suche nach dem Zeiger auf ein Objekt A ein Traversieren eines Baumes darstellt. Die Wurzel ist hierbei der Wurzelknoten für A . In welchem Unterbaum man sich befindet, hängt vom Startknoten ab. Die in den nächsten beiden Abschnitten erläuterten Algorithmen, Pastry und Tapestry, machen direkten Gebrauch von den hier vorgestellten Verfahren.

4.4 Pastry

Pastry [45] wurde als Projekt bei Microsoft Research [30] von ANTHONY ROWSTRON (Microsoft Research) und PETER DRUSCHEL (Rice University, Houston, Texas, USA) ins Leben gerufen. Dieser Routing-Algorithmus ist prefixbasiert. Wir werden in diesem Abschnitt zeigen, dass es sich bei Pastry um einen **strukturierten, deterministischen** und **homogenen** Algorithmus handelt. Es existieren zwei Implementationen von Pastry, beide sind auf der Homepage¹ des Projekts referenziert:

SimPastry/VisPastry: Dieses Paket aus Pastry-Simulator und C#-Implementation (früher Visual Basic) des Protokolls wird nur als Binärdistribution unter einer nicht-freien Microsoft-EULA² vertrieben und ist somit für wissenschaftliche Untersuchungen ungeeignet. Es stellt vielmehr eine Microsoft-interne Machbarkeitsstudie dar. Dies ist schon einmal daran zu sehen, dass das Projekt nicht über die Versionsnummer 1.1 hinausgekommen ist und die letzte Änderung daran zum Zeitpunkt der Erstellung dieses Dokuments bereits über ein Jahr zurück liegt.

FreePastry: PETER DRUSCHEL hat eine eigene Implementation von Pastry in Java 1.4 verfaßt, die an der Rice University, Houston, zum Download bereitliegt. Auch hier liegen die letzten Änderungen bereits ein halbes Jahr zurück. Allerdings ist dieser Code im Gegensatz zu SimPastry verwertbar, da er unter einer BSD-ähnlichen Lizenz weitergegeben wird. Somit ist ein Verwenden, Verändern und Weitergeben bedenkenlos möglich.

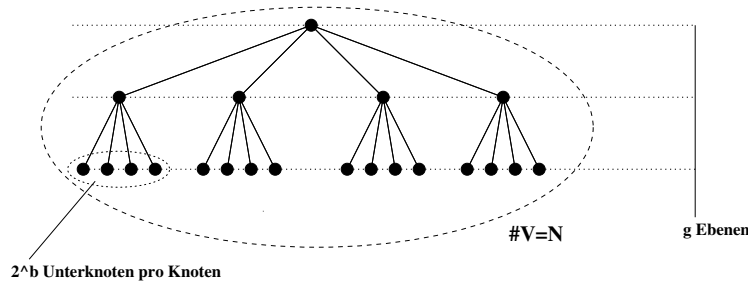


Abbildung 4.4.14: Das Systemmodell von Pastry: Ein Baum, der alle $\#V = N$ Knoten beinhaltet. Weil $N = (2^b)^g$, gibt es pro Knoten 2^b Unterknoten und g Ebenen im Baum.

Systemmodell

Pastry speichert selbst keine Objekte, sondern wird von den Autoren als reine Routing-Schicht präsentiert, deren einzige Aufgabe das Übermitteln von Nachrichten zwischen den teilnehmenden Knoten ist. Bei der Analyse des Algorithmus stellt man fest, dass sehr viele Konzepte von PLAXTON übernommen und konkretisiert wurden. Ein Problem bei der Verwendung des nur als theoretische Studie interessanten Algorithmus von PLAXTON ist, dass dort weder auf dynamische Elemente wie Join und Leave, noch auf Knotenausfälle oder Stabilisierung eingegangen wurde. Ebenso geht PLAXTONs Modell davon aus, dass ein anfragender Knoten die ID des gewünschten Objekts exakt kennt. Dies ist — wie die Praxis beweist — aber überhaupt nicht der Fall. Diese Konzepte mussten in Pastry (ebenso wie in Tapestry) erst noch hinzuerfunden werden.

¹<http://research.microsoft.com/~antr/Pastry/>

²End User License Agreement.

Das System verwendet einen 128 Bit breiten Adressraum, in dem präfixbasiertes Routing angewendet wird. Knoten und (in Bezug auf die Anwendung PAST [15]) Objekte tragen IDs, die im selben Adreßraum liegen. Die Zuordnung von Knoten und Objekten zu IDs findet durch eine Hashfunktion statt, im Paper wird SHA-1 angegeben. Dies führt wegen des 160 Bit breiten Outputs von SHA-1 zu einem Widerspruch zu dem 128 Bit breiten Adreßraum, auf den im Paper nicht eingegangen wird. Dieser Widerspruch könnte durch Verwendung einer passenden Funktion wie MD5 oder durch Abschneiden des 160 Bit Outputs auf 128 Bit erfolgen³. Auf jeden Fall wird in Pastry eine verteilte Hashtabelle realisiert, was den Algorithmus als *strukturiert* auszeichnet.

Die Sichtweise jedes Knotens auf das Netz, legt einen Baum (vgl. Abbildung 4.4.14) zugrunde, so dass das Netzwerk aller Knoten als ein System überlagerter Bäume betrachtet wird. Da jeder Knoten die gleichen Aufgaben wahrnimmt und die Software in der Funktionalität äquivalent ist, stellt Pastry ein *homogenes* Peer-to-Peer System dar. Um eine ID in Pastry zu lokalisieren, greift folgende Zuständigkeit:

Definition 4.4.1 (Zuständigkeit in Pastry) Für eine ID $a \in \mathbb{Z}_{2^{128}}$ ist derjenige Knoten $x \in V$ zuständig, der in der verwendeten Entfernungsdefinition am nächsten liegt, also $\forall y \neq x \in V : d(a, x) < d(a, y)$.

Dies erfordert natürlich eine sofortige Klärung des Entfernungsbegriffs: Pastry verwendet prinzipiell die EUKLIDISCHE Metrik. Im eindimensionalen Raum, wie er bei Pastry verwendet wird, vereinfacht sie sich aber wegen $d = 1$ zu:

$$d(\vec{x}, \vec{y}) := \left(\sum_{i=1}^d (x_i - y_i)^2 \right)^{\frac{1}{2}} = ((x - y)^2)^{\frac{1}{2}} = |x - y|; \quad \forall x, y \in \mathbb{Z}_{2^{128}}$$

Definition 4.4.2 (Entfernungsbegriff in Pastry) Zwei Knoten x und y , identifiziert durch ihre als Binärstrings vorliegenden IDs, haben in Pastry die Entfernung $d(x, y) := |x - y|$ zueinander. Dies ist eine echte Metrik. Die Metrik ist aber nicht eindeutig, d.h.

$$\exists x \in V : \exists a \neq b \in V : d(a, x) = d(b, x)$$

Zustandssicherung

Jeder Knoten hält bei Pastry drei Kategorien von Informationen, die seinen lokalen Zustand ausmachen. In Tabelle 4.4.3 ist ein Beispiel für einen Knotenzustand illustriert.

Bezeichnung	Beschreibung
Menge der Blätter $\text{leaf}[-l; l]$	Diese Menge wird als L bezeichnet und hält eine Auswahl der $ 2^b $ numerisch nächsten, d. h. gemessen an der Knoten-ID am wenigsten weit entfernten Knoten.
Routing Tabelle $\text{routing}[0; g - 1][0; 2^b - 1]$	Die Routingtabelle ist in g Zeilen unterteilt und enthält pro Zeile i eine Auswahl von 2^b Knoten, die mit der ID des eigenen Knotens um i Ziffern zur Basis 2^b von vorne übereinstimmen. Die $i + 1$ -te Ziffer eines Eintrags entspricht dann jeweils seiner Position in der Zeile. Hierdurch wird das präfixbasierte Routing realisiert.
Nachbartabelle $\text{neighbor}[0; 2^b - 1]$	Diese Menge ist zur Effizienzsteigerung des Protokolls vorhanden, da sie Informationen über <i>physikalisch</i> nahe Knoten enthält.

³In RFC 2104 [27, Abschnitt 5] über HMACs wird unter Hinweis auf ein Paper von VAN OORSCHOT [41] erwähnt, dass dieses Abschneiden generell unproblematisch ist, solange der verbleibende Output eine Restbreite von $b_{neu} = \max\{80, b_{alt}\}$ besitzt.

Knoten ID 10233102			
← kleiner		Blättermenge	größer →
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routingtabelle				
<i>i</i>	<i>j</i> = 0	<i>j</i> = 1	<i>j</i> = 2	<i>j</i> = 3
0	0 2212102	self	22301203	3 1203203
1	self	1 1301233	12230203	1 3021022
2	1 0031203	1 0132102	self	1 0323302
3	102 0 0230	102 1 1302	102 2 2302	self
4	1023 0 322	1023 1 000	1023 2 121	self
5	10233 0 01	self	10233 2 32	
6	self		10233 1 20	
7			self	

Nachbarmenge			
13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Tabelle 4.4.3: Beispiel des lokalen Knotenzustands in einer einfachen Pastry-Implementation mit $b = 2$ und einer stark verkürzten $ID = 10233102$. Es wird modulo 4 gerechnet, da $2^b = 2^2 = 4$. Jede Ziffer in der ID steht also für zwei Bits im Binärstring. Der Präfix der IDs in der Routingtabelle stimmt pro Zeile eine Ziffer mehr mit der eigenen ID überein. Die Ziffer danach entspricht der Spaltennummer j .

Der Lokalisierungsalgorithmus

Pastry basiert ebenso wie Tapestry (vgl. Abschnitt 4.5) auf PLAXTONs Algorithmus [39]. Deshalb werden wir auch mehrere Notationen aus Plaxton benutzen, vor allem die Darstellung des i -ten Bits von x als x^i und die Notation $x[i] := x^{(i+1)b-1} \dots x^{ib}$, um eine Ziffer zur Basis 2^b aus dem String herauszugreifen.

Je nach Entfernung des gewünschten Ziels werden unterschiedliche Teilmengen des eigenen Zustands für eine Routing-Entscheidung herangezogen. Das Vorgehen wird in der Prozedur `lookup()` (Algorithmus 4.4.1) beschrieben: Zuerst wird geprüft, ob sich der gesuchte Knoten bereits in der eigenen Blättermenge befindet, also in einer Ebene auf dem Baum liegt und den selben Elternknoten besitzt. Damit ist der Knoten direkt über einen logischen Hop erreichbar und wird sofort zurückgegeben. Ansonsten wird auf die Routing-Tabelle (die das eigentliche Verfahren von PLAXTON realisiert) zurückgegriffen und geprüft, ob ein Knoten existiert, der im Präfix mit einer Ziffer mehr übereinstimmt als der aktuelle Knoten. Existiert auch ein solcher Knoten nicht (seltener Fall), so wird aus der Vereinigungsmenge aller drei Zustandstabellen derjenige Knoten herausgesucht, der mindestens so viele Ziffern im Präfix gemeinsam mit dem Ziel hat wie der aktuelle Knoten, aber numerisch (d. h. auf die ID im Baum bezogen) näher am gesuchten Knoten liegt. Diese letzte Stu-

Algorithmus 4.4.1 `n.lookup(id)`

Stellt her: Gibt den Knoten zurück, der am nächsten an id liegt.

```

 $l \leftarrow 2^{b-1};$ 
if  $id \in [\text{leaf}[-l], \text{leaf}[l]]$  then
   $x \leftarrow \{t \in \text{leaf}[] \mid d(n, t) < d(n, y); \quad \forall y \in \text{leaf}[]\};$ 
else
   $x \leftarrow n;$ 
   $y \leftarrow \text{undef};$ 
  while  $(x \neq y)$  do
     $y \leftarrow x;$ 
     $t \leftarrow \text{shared\_length}(n, id);$ 
     $\text{digit} \leftarrow id[t]; \quad (\text{digit} \in [2^b])$ 
    if  $(\text{routing}[t][\text{digit}] \neq \text{undef})$  then
       $x \leftarrow \text{routing}[t][\text{digit}];$ 
    else
       $M \leftarrow \{v \in (n.\text{neighbor}[] \cup n.\text{leaf}[] \cup n.\text{routing}[]) \mid \text{shared\_length}(n, v) \geq t\};$ 
       $x \leftarrow \{v \in M \mid d(n, v) < d(n, y); \quad \forall y \neq v \in M\};$ 
    end if
  end while
end if
return  $x;$ 

```

fe der numerischen Annäherung an die Ziel-ID musste im Originalverfahren von PLAXTON nicht gemacht werden, weil dort eine exakte Kenntnis der ID vorausgesetzt wurde. Die Erklärung der Zuständigkeit von Knoten durch numerische Nähe zur ID behebt dieses Problem.

Algorithmus 4.4.2 $n.\text{shared_length}(x, y)$

Stellt her: Gibt die Länge des gemeinsamen Präfixes von x und y zurück.

```

 $b \leftarrow x \oplus y;$ 
 $i \leftarrow 0;$ 
while ( $b_{127-i} = 0$ ) do
   $i \leftarrow i + 1;$ 
end while
return  $i;$ 

```

Join und Leave

Es wird davon ausgegangen, dass der Knoten n , der sich an das System anmelden will, das Problem des *ersten Kontakts* bewältigt hat und den bereits teilnehmenden Knoten w kennt. Zusätzlich nehmen die Designer an, dass der Knoten w physikalisch nahe zum neu hinzukommenden Knoten n ist⁴. Dies ist aber nur für die Nachbartabelle wichtig, die für ein physikalisch effizienteres Routing benutzt wird. Ist w nicht physikalisch nahe, so muss n mit einer schlechten Nachbartabelle starten, die aber während der Laufzeit bei jedem Kontakt mit einem näheren Knoten immer besser wird.

Nun initiiert der neue Knoten n eine Selbstauskunft beim Knoten w , wobei die ID von n selbstgewählt (oder per Hashfunktion bestimmt) ist. Die Auskunft liefert einen Knoten z zurück, der der selbstgewählten ID logisch am nächsten ist. Aufgrund der Annahme einer dünnen Gleichverteilung der Knoten-IDs im 128 bit-Adreßraum ist es höchst unwahrscheinlich, dass die ID n bereits besetzt ist. Selbst wenn die angestrebte ID bereits besetzt wäre, könnte durch eine deterministische Kollisionsbehandlung, wie man sie auch bei Hashfunktionen mit Buckets kennt, eine neue, noch nicht besetzte ID in einer Iteration von wenigen Schritten gefunden werden. Jeder Knoten auf der bei der Selbstauskunft zurückgelegten Routing-Strecke schickt nun dem neuen Knoten n seinen lokalen Zustand zu. Aus dieser Menge an Informationen kann sich n die für ihn geeignetsten Informationen heraussuchen und gewinnt selbst einen eigenen Zustand. Die Nachbartabelle für n wird von w übernommen, da dieser Knoten laut obiger Annahme physikalisch nahe an n liegt. Der Knoten z ist das Ergebnis einer Lokalisierung der ID von n , er liegt also in der Metrik nahe an n und kann somit als Quelle für die Blättermenge verwendet werden. Für den Aufbau der Routingtabelle kann jeweils der i -te Knoten auf dem beim Lookup zurückgelegten Pfad im Graphen von w bis z dienen. Der i -te Knoten stimmt nämlich per Definition bereits in i Stellen mit dem Knoten n überein, deshalb kann n die i -te Zeile der Routingtabelle von diesem Knoten kopieren. Der Algorithmus für das Join (4.4.3) ist deterministisch, ebenso das Verfahren zur Lokalisierung von Knoten. Damit ist auch Pastry ein *deterministischer* Algorithmus.

Für das Verlassen der Knoten gibt Pastry kein besonderes Vorgehen an, die Knoten werden einfach implizit nach der Fail-Stop Semantik als ausgefallen betrachtet.

Tolerierte Fehler, Redundanz

Um feststellen zu können, ab wann ein Fehler im System nicht mehr tolerierbar ist, wird eine Konsistenzdefinition benötigt.

Definition 4.4.3 (Konsistenz in Pastry) *Das System in Pastry ist konsistent, wenn jeder Knoten*

⁴In den Augen des Autors ist diese Annahme vollkommen illusorisch, wenn man sich die realen Gegebenheiten im Internet verdeutlicht. Dort muss etwa ein auf einer Webseite angekündigter Kontakt keinesfalls nahe bei dem Konsumenten stehen, der mit seinem Heimcomputer am Netzwerk partizipieren möchte.

Algorithmus 4.4.3 $n.\text{join}(w)$

Benötigt: Eine Prozedur $\text{send_state}(v)$, die den lokalen Zustand an den Knoten v sendet. w ist ein Knoten, der schon am System teilnimmt.

Stellt her: Nimmt den Knoten n ins Netz mit auf.

```

 $z \leftarrow w.\text{lookup}(n);$ 
 $n.\text{neighbor}[] \leftarrow w.\text{neighbor}[];$ 
 $n.\text{leaf}[] \leftarrow z.\text{leaf}[];$ 
for  $i = 0$  to  $l(w \sim_G z) - 1$  do
   $k \leftarrow \{v \in e \mid \{e \in (w \sim_G z)\} \wedge \#(w \sim_G e) = i\};$ 
   $n.\text{routing}[i] \leftarrow k.\text{routing}[i];$ 
end for
for all  $v \in (n.\text{neighbor}[] \cup n.\text{leaf}[] \cup n.\text{routing}[])$  do
   $n.\text{send\_state}(v);$ 
end for

```

in jeder Zeile seiner Routingtabelle jeweils mindestens einen funktionierenden Knoten gelistet hat oder es für die betreffende Zeile keinen solchen Knoten im System gibt. Ausserdem müssen in der Blättermenge die jeweils 2^{b-1} numerisch in der ID vorangehenden und die 2^{b-1} numerisch folgenden Knoten gelistet sein, wenn es diese gibt.

Ist die Konsistenz nicht mehr gegeben, ist auch der Knoten nicht mehr in der Lage, korrektes Routing durchzuführen. Pastry wird allerdings schon weit vor Verlust der Konsistenz anfangen, ausgefallene Knoten durch funktionierende Knoten in seinen Tabellen zu ersetzen, dazu gleich mehr bei der Stabilisierung.

Stabilisierung

Die Stabilisierung in Pastry erfolgt explizit: Zu den Knoten, die physikalisch nahe sind (also in der Nachbartabelle) werden periodisch Nachrichten geschickt, um zu prüfen, ob diese immer noch online sind. Bei Knoten aus den anderen beiden Tabellen wird ein ausgefallener Knoten nur nach dem Ausbleiben einer Nachricht nach der Fail-Stop Semantik bemerkt. In beiden Fällen wird jeweils durch explizite Anfragen bei den physikalischen Nachbarn bzw. durch Lokalisierungsanfragen unter Zuhilfenahme des Routings Ersatz gesucht und geeignete Knoten anstelle der ausgefallenen in den lokalen Zustand übernommen.

Verwaltungsaufwand

Hier brauchen nur die Routing-relevanten Kosten betrachtet werden, da Pastry selbst keinerlei Objektspeicherung vorsieht.

Komplexität einer Lokalisierung (C_F): Bis auf den letzten Routing-Schritt werden alle Weiterleitungen über die Routingtabelle durchgeführt und die Übereinstimmung im Präfix mit jedem Schritt um mindestens eine Ziffer zur Basis 2^b erhöht. Also wird die Anzahl der noch in Frage kommenden Zielknoten mit jedem Schritt durch 2^b geteilt, so dass sich mit dem letzten Schritt eine Komplexität von $C_F = \mathcal{O}(\log_{2^b} N) + 1 \approx \mathcal{O}(\log_{2^b} N)$ ergibt.

Größe der Verwaltungsstrukturen auf jedem Knoten (D_V): Die Nachbartabelle und die Blättermenge haben jeweils 2^b Einträge, die Routingtabelle besitzt g Zeilen mit jeweils $2^b - 1$ Spalten. Aufsummiert ergibt das einen lokalen Zustand von $D_V = (2 + g)2^b - g$ Wörtern.

Kommunikationskosten für Join und Leave (C_J, C_L): Das Hinzukommen eines Knotens n bedeutet zunächst eine Lokalisierung des zu n nächsten Knotens, also Kosten von C_F . Weiterhin schickt jeder Knoten auf dem Lookup-Pfad dem Knoten n eine Nachricht mit seinem kompletten lokalen Zustand. Anschließend schickt der neue Knoten n jedem Knoten aus seiner Zustandsmenge ebenfalls seinen eigenen Zustand, was zusätzlich einen Aufwand von $(2 + g)2^b - g$ Mitteilungen bedeutet. Der Aufwand, den Zustand zu übertragen, wird mit der Konstante C angenommen, so dass sich ergibt: $C_J = \mathcal{O}(\log_{2^b} N)(1 + C) + C((2 + g)2^b - g)$. Weil aber $\log_{2^b} N = g$ und $C((2 + g)2^b - g) = \mathcal{O}(g)$ gilt, kann man noch zusammenfassen: $C_J = \mathcal{O}(\log_{2^b} N)$, was das Ergebnis allerdings etwas unschärfer macht.

Anzahl der Knoten, die nach Join ihre Strukturen auffrischen (N_J): Im schlimmsten Fall bemerkt jeder Knoten, der während des Joins mit n Kontakt hatte, dass n günstiger als ein anderer Knoten liegt und deshalb n in den lokalen Zustand mit aufgenommen werden muss. Dann ist $N_J = \mathcal{O}(\log_{2^b} N)$. Da das Verlassen implizit durch Fail-Stop behandelt wird, ist die Zahl $N_L = 0$.

Hinweis auf Probabilistik: Obwohl es nicht standardmäßig im Algorithmus eingebaut ist, wird im Paper zu Pastry [45] die Möglichkeit angeführt, probabilistische Elemente in das Verfahren einzubauen, und so die Routing-Entscheidung auf einem einzelnen Knoten unvorhersehbar zu machen.

Heilung von Netzteilungen durch Multicast: Eine Methode, um durch ausgefallene Knoten auseinandergerissene Teilnetze in Pastry wieder zusammenzuführen, soll laut dem Paper eine Suche per IP Multicast sein. Dies ist aber in den Augen des Autors wenig empfehlenswert (vgl. Abschnitt 5.2, Absatz über IPv4 Multicast).

Eignung für Anwendungen: Die Baumstruktur begünstigt ein Fluten des Baumes mit Nachrichten in rekursiver Weise (jeder Knoten gibt an seine Unterknoten weiter). Deshalb ist Pastry geeignet für Multicast, was mit Anwendungen wie Scribe (siehe Anhang A.3) realisiert werden kann. Desweiteren kann natürlich auch einfache Blockspeicherung mit Pastry ausgeführt werden, wobei jeweils der nächste Knoten mindestens für ein bestimmtes Objekt, gekennzeichnet durch seine dem Knoten nahe ID, zuständig ist. Es lassen sich also auch verteilte Dateisysteme oder Datenspeicher mit Pastry realisieren. Hierfür dient z. B. PAST als Beispiel, das auch im Anhang kurz beschrieben wird.

4.5 Tapestry

Tapestry stammt von der Universität Berkeley in Kalifornien. Auch Tapestry ist, genau wie Pastry (vgl. Abschnitt 4.4), stark an PLAXTONs Algorithmus [39] angelehnt, wenn es um den theoretischen Hintergrund des Routings und der Lokalisierung geht. Leider ist die Dokumentation zu Tapestry [58] an vielen Stellen oberflächlich gehalten und läßt daher viel Raum für Mehrdeutigkeiten bei der Interpretation. Das Routing ist präfixbasiert, deshalb ist das Systemmodell ein Baum, genau wie bei Pastry (siehe Abbildung 4.5.15). Wir werden im folgenden Tapestry als **strukturierten, deterministischen** und **homogenen** Algorithmus klassifizieren.

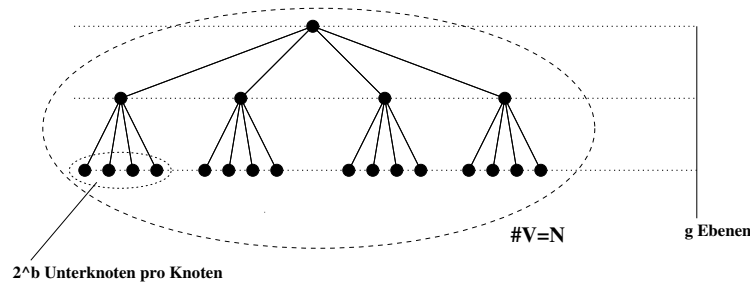


Abbildung 4.5.15: Das Systemmodell von Tapestry gleicht dem von PLAXTONs Algorithmus und Pastry: Ein Baum, der alle $\#V = N$ Knoten beinhaltet. Weil $N = (2^b)^g$, gibt es pro Knoten 2^b Unterknoten und g Ebenen im Baum.

Systemmodell

Wie alle untersuchten strukturierten Peer-to-Peer Algorithmen realisiert auch Tapestry eine verteilte Hashtabelle, daher das Attribut *strukturiert*. Als Hashfunktion kommt SHA-1 zum Einsatz. Tapestry ist ein präfixbasierter Algorithmus, was bedeutet, dass jeder teilnehmende Knoten eine (i, j) -Tabelle an bekannten Knoten hält ($i \in [g]; j \in [2^b]$ mit $N = (2^b)^g$), die mit der eigenen Knoten-ID eine i -stellige Übereinstimmung und in der folgenden der folgenden Stelle eine andere, durch j angegebene Ziffer, aufweisen. Diese Tatsache kann wie bereits bei Pastry durch einen Baum modelliert werden. Auch Tapestry sieht vor, dass alle Knoten die gleichen Aufgaben im System übernehmen sollen, also ist auch Tapestry ein *homogener* Algorithmus.

Tapestry muss — genau wie Pastry — das Problem umgehen, dass der theoretische Algorithmus von PLAXTON nur für Suchanfragen entworfen wurde, die tatsächlich auf ein echt existierendes Objekt A gestellt werden. Das Problem an der eindeutigen Existenz von Wurzelknoten für Objekte ist nämlich, dass diese Zuordnung bei PLAXTONs Algorithmus zu Beginn des Systems einmal aufgestellt wurde, und der Algorithmus Operationen wie Join und Leave oder den Ausfall von Knoten überhaupt nicht kennt. Für Anfragen auf willkürliche IDs im 160 Bit Namensraum musste also ein Routing zu einem Stellvertreter, dem für die ID zuständigen Knoten überhaupt erst entworfen werden. Während Pastry das Problem über eine Metrik löst, die numerische Nähe zur ID bevorzugt, ist im Paper zu Tapestry die Rede von einem *surrogate routing*, also dem Suchen eines Stellvertreters für die ID als Wurzelknoten.

Definition 4.5.1 (Zuständigkeit in Tapestry) *Zuständig für eine ID x in Tapestry ist derjenige Knoten, der aufgrund eines systemweit eindeutigen, deterministischen und total korrekten, auf den*

Routingtabellen der Knoten arbeitenden Algorithmus gefunden wird, nachdem mit PLAXTONs Routing keine genauere Übereinstimmung mit dem Präfix mehr herstellbar ist. Dieser Algorithmus wird Stellvertreter-Routing genannt.

Die genaue Implementation des Stellvertreter-Routings wird in der Arbeit über Tapestry offengelassen. Wichtig sei vor allem, dass der Algorithmus die o. g. Bedingungen erfüllt.

Die Entfernung in Tapestry wird auf zweierlei Art gemessen: Logische und physikalische Metrik. Die logische wird in der Zahl der Ziffern ausgedrückt, die im Präfix zweier Punkte x und y noch nicht übereinstimmen. Allerdings hat sie wegen des verwendeten Stellvertreter-Routings im Algorithmus weniger Bedeutung als in Pastry. Viel wichtiger ist die schon bei PLAXTON vorgestellte physikalische Entfernung c , die nach realen Begriffen wie Anzahl der Hops, Latenz, Kosten der Leitung etc. gemessen wird.

Definition 4.5.2 (Entfernungsbegriff in Tapestry) Zwei Punkte x und y haben in Tapestry die logische Entfernung

$$d(x, y) := \left\lceil \frac{\log_2((x \oplus y) + 1)}{b} \right\rceil$$

und die physikalische Entfernung c , wobei $c(x, y)$ die Kosten der Übertragung eines Wortes vom Knoten x zum Knoten y angibt.

Die logische Metrik begrenzt die Anzahl der Hops bis zum zuständigen Knoten einer ID nach unten, da mindestens i Hops erforderlich sind, um eine i -stellige Präfixübereinstimmung zwischen der gesuchten ID und dem aktuellen Knoten herzustellen.

Zustandssicherung

Zum Zustand eines Knotens in Tapestry gehören:

Bestandteil	Beschreibung
Routingtabelle routing[i][j]	Hier ist das von PLAXTON eingeführte präfixbasierte Routing realisiert (vgl. Pastry in Abschnitt 4.4). Das bedeutet, dass die mit $i \in [g]$ indizierten Zeilen Zeiger auf Knoten beinhalten, deren Präfix mit i Ziffern zur Basis 2^b mit der eigenen ID übereinstimmt. Die Variable $j \in [2^b]$ gibt den Wert der auf den Präfix folgenden Ziffer an. Jede Zelle beinhaltet einen primären Zeiger <code>.primary</code> und eine Menge an sekundären Zeigern <code>.secondary[]</code> . Diese Menge beinhaltet in Tapestry zwei Zeiger.
Objektspeicher	Auch wenn bei anderen Systemen stillschweigend vorhanden, werden die gespeicherten Objekte im Paper [58] explizit als Teil des lokalen Zustands aufgezählt. Die Speicherung der Objekte erfolgt in Tapestry statisch, es findet also keine eigenmächtige Löschung durch das System statt.
Objektzeigertabelle ptr[]	Alle dem Knoten bekannten Objekte, die auf fremden Knoten liegen, werden hier referenziert. Im Gegensatz zu den Objekten sind die Zeiger dynamisch gespeichert, sie müssen also aufgefrischt werden.
Hotspot Monitor	Eine Datenstruktur, in der Tupel der Form $\langle id(A), x, f \rangle$ gehalten werden. Ein solches Tupel zeigt an, dass Anfragen des Knoten x auf die ID des Objektes A eine bestimmte, systemweit festgelegte Häufigkeit übersteigen. Die genaue Frequenz wird in f festgehalten.
Rückwärts-Zeigertabelle back[]	Zeiger auf Knoten, die den lokalen Knoten als Nachbarn ansehen.

Der Lokalisierungsalgorithmus

Wie bereits erwähnt, wird zunächst eine Lokalisierung nach PLAXTONS Verfahren implementiert, die ganz ähnlich zu der von Pastry abläuft. Es wird also ein Routing in i Schritten durchgeführt, wobei bei jedem Schritt eine Ziffer mehr im Präfix des nächsten Knotens übereinstimmen muss. Hierzu wird, wie bei Pastry auch, die (i, j) -Routingtable benutzt. Die Prozedur `lookup()` (Algorithmus 4.5.1) beschreibt das Vorgehen. Ab dem Zeitpunkt, wo kein Fortschritt im Präfix mehr

Algorithmus 4.5.1 $n.lookup(id)$

```

 $x \leftarrow n;$ 
 $y \leftarrow \text{undef};$ 
while ( $x \neq y$ ) do
   $y \leftarrow x;$ 
   $t \leftarrow d(n, id);$ 
   $digit \leftarrow id[t];$  ( $digit \in [2^b]$ )
  if ( $\text{routing}[t][digit] \neq \text{undef}$ ) then
     $x \leftarrow \text{routing}[t][digit];$ 
  else
     $x \leftarrow y.surrogate(id);$  (offengelassen)
  end if
end while

```

gemacht werden kann, wird auf das in der Dokumentation nicht näher erläuterte in der Dokumentation nicht näher erläuterte Stellvertreter-Routing zurückgegriffen. Dieses besagt, dass bei Fehlen eines Knotens, der einen Fortschritt nach PLAXTONS Schema bringen könnte, nach einem deterministischen Verfahren ein alternativer Knoten ausgewählt wird. Diese Fortsetzung des Verfahrens hat laut dem Paper zu Tapestry nach 2 Schritten ein Ende und konvergiert von jedem Knoten aus auf den selben Knoten im System. Dieser Knoten ist der Stellvertreterknoten der ID von A , der in Tapestry den Wurzelknoten für A darstellt.

Wie man beim Vergleich der `lookup()`-Algorithmen von Pastry und Tapestry feststellt, sind diese nicht sehr unterschiedlich. Der Vorgang, der bei Tapestry als *surrogate routing* bezeichnet wird, wird bei Pastry dadurch realisiert, dass immer zu dem der ID numerisch nächsten Knoten geroutet wird. Dies ist auch ein eindeutiger und deterministischer Vorgang.

Join und Leave

Das Hinzukommen eines Knotens zu Tapestry ist mit der Prozedur `join()` (Algorithmus 4.5.2) realisiert und gestaltet sich so ähnlich wie in Pastry: Knoten n kennt bereits den Knoten w , der dem System angehört. Wie bei Pastry wird angenommen, dass der erste Kontakt physikalisch nahe liegt¹. Der Knoten n läßt nun über ihn mittels der Prozedur `lookup()` (Algorithmus 4.5.1) eine spezielle Join-Anfrage zu seiner eigenen ID routen. Jeder Knoten, der mit dieser speziellen Anfrage in Berührung kommt, schickt dem Knoten n eine Kopie seiner Routingtable. Aus den erhaltenen Daten kann n sich seine persönliche Sicht auf das System zusammenstellen. Da per Definition des Algorithmus der i -te Knoten auf dem Routingpfad eine i -stellige Übereinstimmung mit der eigenen ID bedeutet, kann n sich von dem jeweiligen Knoten — wie bei Pastry auch — die i -te

¹Nach Auffassung des Autors genau wie bei Pastry vollkommen unberechtigt, da im Internet die Information über erste Kontakte durchaus auf Webseiten oder ähnlichen Wegen publiziert werden kann, und der Empfänger dieser Information mitnichten lokal zum ersten Kontakt oder zum Webserver sein muss.

Zeile der Routingtabelle des i -ten Knotens auf dem Routingpfad kopieren und für seine eigenen Strukturen verwerten. Die kopierten Strukturen könnten für den Knoten n eventuell nicht optimal

Algorithmus 4.5.2 $n.join(w)$

Benötigt: Eine Prozedur `ping(v)`, die den Knoten v auf n aufmerksam macht. w ist ein Knoten, der schon am System teilnimmt.

Stellt her: Nimmt den Knoten n ins Netz mit auf.

```

 $z \leftarrow w.lookup(n);$ 
 $i \leftarrow 0;$ 
repeat
   $k \leftarrow \{v \in e \mid \{e \in (w \sim_G z)\} \wedge \#(w \sim_G e) = i\};$ 
   $n.routing[i] \leftarrow k.routing[i];$ 
   $i \leftarrow i + 1;$ 
until  $\#(k.routing[i]) \leq 1;$ 
 $n.optimize\_neighbors();$ 
for all  $\{k \in e \mid \{e \in (w \sim_G z)\}$  do
  for all  $b \in k.back[]$  do
     $ping(b);$ 
  end for
end for

```

liegen. Deshalb wird mittels der Prozedur `optimize_neighbors()` (Algorithmus 4.5.3) dafür gesorgt, dass der jeweilige primäre Nachbar der unter physikalischen Metrik günstigste ist. Nach

Algorithmus 4.5.3 $n.optimize_neighbors()$

Stellt her: Der primäre Nachbar in einer Zelle ist jeweils der Knoten mit dem geringsten Abstand zu n .

```

for all  $(t \in n.routing[][])$  do
  if  $(t \neq \emptyset)$  then
     $p \leftarrow t.primary;$ 
     $x \leftarrow \{y \in t.secondary[] \mid c(n, y) < c(n, z) \ \forall z \neq y \in t.secondary[]\};$ 
    if  $d(n, p) > d(n, x)$  then
       $t.primary \leftarrow x;$ 
       $t.secondary \leftarrow t.secondary \cup p;$ 
    end if
  end if
end for

```

der Optimierung der Routingtabelle werden noch diejenigen Knoten benachrichtigt, die von einem Hinzukommen von n betroffen sein könnten. Hierzu werden alle in den Rückwärtszeigern gelisteten Knoten aller auf dem Routing-Pfad von w nach z berührten Knoten über das Hinzukommen von n benachrichtigt. Die benachrichtigten Knoten prüfen, ob n in ihren Datenstrukturen eingebaut werden muss, weil er günstiger liegt und führen dies bei Bedarf durch.

Wie wir gesehen haben, bieten weder Join noch Lokalisierungsalgorithmus Freiheitsgrade für Probabilistik, so dass das Verhalten von Knoten mit dem selben Zustand vorhersehbar ist. Auch Tapestry ist deshalb ein *deterministischer* Algorithmus.

Tolerierte Fehler und Redundanz

Ein Novum im Gegensatz zu den bereits untersuchten Algorithmen ist die Fähigkeit von Tapestry, ausgefallene Knoten als *temporär* ausgefallen zu betrachten, anstatt sie sofort aus den internen Verwaltungsstrukturen auszutragen. Dieses Verfahren wird in dem Paper als *soft state* bezeichnet. Weil ein Rechner, der bereits am Netz teilgenommen hat, mit einer signifikanten Wahrscheinlichkeit wieder auftauchen wird (siehe statistische Untersuchungen am Gnutella Projekt [17], erwähnt bei Kademia in Abschnitt 4.6), nachdem der Reboot oder die aus anderen Gründen herbeigeführte Unerreichbarkeit überwunden sind. Der Knoten wird also erst nach einem erfolglosen Wiederholungsversuch aus der Tabelle herausgenommen. Dies erspart auch sehr oft das Ein- und Austragen des Knotens im Rahmen einer regulären Join/Leave-Operation.

Eine Definition des Zustands, in dem ein Knoten noch korrektes Routing durchführen kann, lautet ähnlich wie bei Pastry:

Definition 4.5.3 (Konsistenz in Tapestry) *Das System in Pastry ist konsistent, wenn jeder Knoten in jeder Zeile seiner Routingtabelle jeweils mindestens einen funktionierenden Knoten gelistet hat, oder es für die betreffende Zeile keinen solchen Knoten im System gibt.*

Eine weitere Möglichkeit, in Tapestry Redundanz zu erzeugen, ist die Einführung eines Hash-Saltings, also dem Hinzufügen eines Wertes an den zu hashenden String. Je nach hinzugefügtem String werden Objekte nicht nur mit einer, sondern mit vielen verschiedenen, durch das Salting entstandenen IDs assoziiert. Hierdurch sind mehrere Wurzelknoten zu einem Objekt verfügbar, und so steht nicht mehr nur ein Wurzelknoten als *single point of failure* im System.

Stabilisierung

Um die Konsistenz zu wahren, wird die Menge der Rückwärtszeiger periodisch mit einem `ping()` geprüft. Fehlerhafte Knoten in der Routingtabelle fallen implizit bei Benutzung auf. Bei Fehlern in der Routingtabelle werden nacheinander die sekundären Zeiger einer Zelle in der Reihenfolge ihrer physikalischen Entfernung c als Ersatz für einen ausgefallenen primären Zeiger verwendet. Neue Einträge kommen bei Empfang von `ping()`-Nachrichten implizit zur Tabelle hinzu, wenn nötig.

Auch mit der Betrachtung eines Knotens als ausgefallen wird im Verfahren zunächst gewartet: Nicht reagierende Knoten werden in Tapestry nicht sofort aus dem lokalen Zustand ausgetragen, sondern nach einem definierten Zeitintervall noch einige Male gepingt. Erst bei vollkommenem Ausfall werden sie wirklich aus den lokalen Tabellen entfernt. Hierdurch können kürzere Netzausfälle verkraftet werden, und die Gefahr einer Netzteilung durch einen Verbindungsausfall ist nicht mehr so hoch wie bei anderen Systemen.

Aufwandsbetrachtung

Durch das Offenlassen der *surrogate* Routingstrategie können nur die im Paper [58] behaupteten oberen Grenzen dafür in die Aufwandsbetrachtung mit eingehen — eine eigene Untersuchung ist nicht möglich. Laut der Dokumentation soll die Anzahl der Schritte, die beim *surrogate routing* benötigt werden, nicht größer als 2 sein.

Komplexität einer Lokalisierung (C_F): Durch schrittweises Weiterleiten einer Anfrage an einen anderen Knoten, der mit der gesuchten ID jeweils eine Ziffer zur Basis 2^b mehr übereinstimmt, wird der Raum der in Frage kommenden Knoten mit jedem Schritt um eine Größenordnung in der Basis

2^b eingeschränkt, was eine logarithmische Anzahl von Schritten in der Gesamtknotenzahl N ergibt. Hinzu kommen noch Kosten für die *surrogate routing* Strategie, die aber maximal 2 Hops betragen sollen. Somit ergibt sich für $C_F = \mathcal{O}(\log_{2^b} N) + 2$.

Größe der Verwaltungsstrukturen auf jedem Knoten (D_V): Vernachlässigt man den Objektspeicher, die Objektzeigertabelle und den Hotspot Monitor, da diese Strukturen mit der eigentlichen Lokalisierung nichts zu tun haben, so wird D_V von zwei Strukturen bestimmt: Der Routingtabelle und der Rückwärtszeigertabelle. Die Routingtabelle enthält g Zeilen und 2^b Spalten mit Zellen, die jeweils einen primären und zwei sekundäre Einträge beinhalten. Die Anzahl der Rückwärtszeiger pro Knoten ist die Anzahl der primären Zeiger anderer Knoten, die auf den eigenen Knoten zeigen und kann nur geschätzt werden: Es gibt N Knoten im System, diese besitzen $2^b \cdot g$ primäre Zeiger auf andere Knoten. Ohne sehr viel Genauigkeit aufzugeben, kann man von durchschnittlich $2^b \cdot g$ Zeigern sprechen, die auch wieder auf jeden einzelnen Knoten verweisen, weil von einer Gleichverteilung der Knoten-IDs im System ausgegangen wird. Deshalb ergibt sich $D_V = 3(2^b \cdot g) + 2^b \cdot g = \boxed{4(2^b \cdot g)}$.

Kosten, ein Objekt einzustellen oder zu entfernen (C_I, C_D): Da ein Objekt analog zu PLAXTONS Algorithmus immer auf dem lokalen Knoten eingestellt wird, entfallen die Kosten für das Kopieren. Auch die Replikation findet auf einer höheren Ebene statt, wodurch auch hierfür die Kosten entfallen. Alles, was getan werden muss, ist, den Wurzelknoten für das einzustellende Objekt A zu finden und dort einen Zeiger auf den lokalen Knoten abzulegen. Wir benötigen also nur die Kosten für die Lokalisierung und eine Konstante C , die die Kosten für die Kommunikation ausdrückt. Für das Löschen von Objekten verhält es sich ganz analog. Also gilt $C_I = C_D = \mathcal{O}(\log_{2^b} N) + 2 + C$.

Kommunikationskosten für Join und Leave (C_J, C_L): Beim Hinzukommen ist eine Lokalisierung der gewählten ID n des neuen Knotens nötig, was Kosten von C_L verursacht. Anschließend werden von den $X = \mathcal{O}(\log_{2^b} N)$ Knoten auf dem Weg die Routingtabellen kopiert, um die eigenen Strukturen zu initialisieren. Danach werden noch die Rückwärtszeiger aller Knoten auf dem Weg benachrichtigt, was für jeden der $X + 2$ Knoten noch einmal einen durchschnittlichen Nachrichtenaufwand von X Nachrichten verursacht. Es ergibt sich somit ein

$$\begin{aligned} C_J &= C_F + \mathcal{O}(\log_{2^b} N) + C_F(\mathcal{O}(\log_{2^b} N)) = \\ &= \mathcal{O}(\log_{2^b} N) + 2 + \mathcal{O}(\log_{2^b} N) + [\mathcal{O}(\log_{2^b} N) + 2] \cdot \mathcal{O}(\log_{2^b} N) = \\ &= \mathcal{O}(\log_{2^b} N) \cdot [4 + \mathcal{O}(\log_{2^b} N)] + 2 = \boxed{\mathcal{O}(\log_{2^b} N)^2} \end{aligned}$$

Ebensoviele Knoten sind es auch, die potenziell eine Auffrischung ihrer eigenen Strukturen betreiben müssen: $N_J = \mathcal{O}(\log_{2^b} N)^2$.

Bei einem Verlassen des Systems wird kein Unterschied zwischen einem beabsichtigten Leave und einer Trennung durch Fail-Stop oder Ausfall der Netzverbindung gemacht. Die Knotenauffrischung und die damit verbundene Kommunikation läuft also über die Mechanismen der Stabilisierung ab und verursacht keine expliziten Kosten: $C_L = N_L = 0$.

Vorteile

Tapestry zeigt einige bei anderen Peer-to-Peer Systemen nicht beobachtete Optimierungen und Feinheiten, die im Folgenden näher erläutert werden.

Hot Spot Monitoring: Um proaktive Lastverteilung durchzuführen, besitzt die Datenstruktur jedes Knotens den sogenannten *hotspot monitor*. Dabei handelt es sich um eine Menge von Tupeln $\langle id(A), x, f \rangle$, die angibt, von welchem Knoten x aus überdurchschnittlich viele, also über einem systemweiten Schwellenwert liegende, Anfragen pro Zeiteinheit (nämlich mit der Frequenz f) nach einem bestimmten Objekt A gestellt werden. Mithilfe eines speziellen Algorithmus (dessen Besprechung nicht Thema dieser Arbeit ist) ermittelt Tapestry die Quelle der Anfragen und sorgt durch gezieltes Verlagern von Objektkopien in die Nähe der Quelle für ein Verebben des *hot spots*.

Berücksichtigung physikalischer Nähe Ein fester Bestandteil des Algorithmus ist das Wählen des primären Nachbarn mithilfe der *physikalischen* Kostenmetrik c und nicht der logischen Metrik d . Hierdurch wird direkt Rücksicht auf physikalische Gegebenheiten genommen und der Aufwand im echten Netzwerk verringert.

4.6 Kademia

Kademia wurde von PETAR MAYMOUNKOV und DAVID MAZIÈRES an der New York University geschaffen. Das System wurde im Hinblick auf Resistenz gegen einfache Denial-of-Service Attacken entwickelt, außerdem integriert es die Stabilisation, die bei anderen Algorithmen wie z. B. Chord [52] einen eigenen Platz einnimmt, gleich implizit in das Verfahren. Wir werden zeigen, dass Kademia ein **homogener, strukturierter** Algorithmus ist, der die Möglichkeit bietet, **probabilistisch** zu arbeiten.

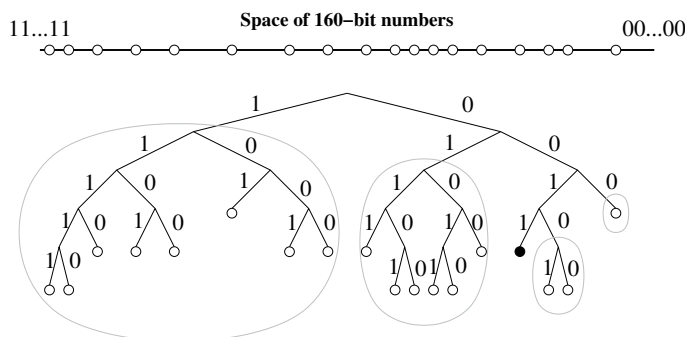


Abbildung 4.6.16: Das Systemmodell von Kademia: Ein Binärbaum, in dem die Knoten als Blätter eingeordnet sind. Die Tiefe jedes Knotens wird durch die Länge seines Präfixstrings bestimmt, der nötig ist, um den Knoten eindeutig unterscheiden zu können.

Systemmodell

Kademia routet (wie Pastry und Tapestry) präfixbasiert, formuliert diese Tatsache aber etwas anders. Das Systemmodell ist ein Binärbaum, in dem die Knoten, eingeordnet mit ihrem kürzesten eindeutigen Präfix, als Blätter auftauchen (siehe Abbildung 4.6.16). Den Knoten werden IDs zugewiesen, die sich aus den Resultaten einer Hashfunktion ergeben, hier wird auch wieder SHA-1 verwendet. Die Realisierung einer verteilten Hashtabelle deutet ebenfalls wieder auf einen *strukturierten* Algorithmus hin. Auch in Kademia übernimmt jeder Knoten identische Aufgaben und ist von seiner Funktionalität gleichwertig, weshalb der Algorithmus zusätzlich *homogen* ist. Die in dieser Analyse vorgestellten Algorithmen sind wie bei den anderen präsentierten Peer-to-Peer Systemen der Semantik nach als RPC-Aufrufe zu verstehen.

Kademia ist über zwei systemweit einstellbare Konstanten konfigurierbar: Der Parameter k drückt aus, wieviele Knoten an der Replikation eines Objektes teilnehmen und wieviele Knoten in einer Verwaltungsstruktur namens *Bucket* (siehe später im Text) enthalten sind. Mit der Konstante α wird angegeben, wieviele redundante Anfragen das System gleichzeitig stellen soll. Standardmäßig sind die Konstanten auf $k = 20$ und $\alpha = 3$ gesetzt.

Worauf sehr viel Wert gelegt wird, ist die Einführung einer neuen, in keinem anderen Algorithmus verwendeten Metrik zur Bestimmung der logischen Entfernung zweier Knoten. Die Autoren weisen nämlich darauf hin, dass ihre verwendete XOR-Metrik eine mathematisch echte Metrik im Sinne von Abschnitt C.3 ist, im Gegensatz zu der von Chord.

Definition 4.6.1 (Entfernungsbegriff in Kademia) Zwei Knoten $x, y \in V$ in Kademia, bezeichnet durch ihre binäre ID, besitzen die Entfernung $d(x, y) := x \oplus y$. Diese Entfernungsfunktion d

erfüllt die **Definitheit**, **Symmetrie** und die **Dreiecksungleichung** und ist somit eine im mathematischen Sinne echte Metrik.

Die XOR-Metrik realisiert das präfixbasierte Routing. Der Leser kann sich leicht selbst davon überzeugen, dass aus einer hohen Übereinstimmung zweier IDs von den hochwertigen Bits gesehen auch eine kurze Entfernung in der Metrik folgt. Neben der Symmetrie besitzt die XOR-Metrik die Eigenschaft der Eindeutigkeit. Zu einer gegebenen Entfernung Δ gibt es also höchstens einen Knoten, der vom aktuellen Knoten genau diese Entfernung besitzt.

$$\text{geg. } \Delta \in [2^m]: \forall x \in K: \#\{v \in V \mid d(x, v) = \Delta\} \leq 1$$

Zustandssicherung

Ein anschauliches Konzept zur Zustandsspeicherung jedes Knotens wird in der Einführung von Listen, den sog. *k*-Buckets, aufgebracht. Die Variablen für den notierten Algorithmus werden wir als `bucket[i][j]` bezeichnen, wobei $i \in [m]$ und $j \in [k]$. Da die deutsche Übersetzung durch *Eimer* hinkt, behalten wir für die Besprechung von Kademia den englischen Begriff bei. Der systemweite Parameter *k* gibt an, wieviele Nachbarknoten gleichzeitig in einem Bucket gespeichert werden können. Die Autoren nennen hierfür einen Standardwert von $k = 20$. Insgesamt gibt es wegen der Bitbreite der SHA-1 Hashes 160 *k*-Buckets. Jede Liste deckt einen bestimmten Entfernungsbereich in der verwendeten XOR-Metrik ab. Ein Analogon für die *k*-Buckets wäre in Chord die Fingertabelle: Beide Konstrukte haben eine optimierende Funktion, indem sie eine Einteilung in Knotenmengen mit exponentiell zunehmender Entfernung realisieren. Da die Metrik bei Kademia sich auf die 160 Bit breiten Knoten-IDs bezieht, erfolgt die Zuteilung zu Buckets folgendermassen: In `bucket[i]` sind Nachbarknoten gespeichert, die unter der XOR-Metrik eine Entfernung im Intervall von $[2^i; 2^{i+1})$ besitzen. Die Entfernungsintervalle der *k*-Buckets wachsen also exponentiell in ihrer Größe an.

Die Organisation der Buckets ist als LRU¹-Strukturen vorgesehen. Die LRU-Struktur dient der impliziten Stabilisierung und wird im entsprechenden Abschnitt erläutert. Vom Start des Knotens an existiert auch nicht für jeden Bereich ein Bucket, sondern die Buckets werden als Blätter hin den Präfixbaum (vgl. Abbildung 4.6.16) gehängt und bei Überlauf geteilt. Am Anfang existiert nur ein großes Bucket, nämlich für den gesamten Baum.

Der Lokalisierungsalgorithmus

Während eine Suche nach einer bestimmten ID bei anderen Algorithmen meist nur einen (den günstigsten) Knoten zurückgibt, liefert eine Lokalisierung per `lookup(id)` (Algorithmus 4.6.1) gleich eine Menge der *k* besten (also in der Metrik nächsten) Knoten zurück. Hierzu sucht sich der Knoten, der die Lokalisierung ausführt, zunächst α Knoten aus seinem lokalen Zustand aus. Die Arbeiten zu Kademia lassen offen, ob wirklich die α besten, oder nur α zufällige, lokal bekannte Knoten gewählt werden sollen. Abhängig von der Implementation dieser Wahl ist der Algorithmus dann deterministisch oder probabilistisch. Diese Auswahl kann beispielsweise mit der Hilfsfunktion `closest(id, α)` (Algorithmus 4.6.2) getroffen werden. Jedem dieser Knoten schickt er parallel

¹Least Recently Used

einen `find_node(id)`-Aufruf (Algorithmus 4.6.3), der bei Erfolg mit einer Menge von k Knoten quittiert wird, die der aufgerufene Knoten für die bestmögliche Menge hält (aufgrund seiner lokalen Sicht). Aus den zurückgelieferten Ergebnissen werden sukzessive wieder jeweils die k besten Knoten ausgewählt und ihrerseits mit `find_node(id)` nach ihrer Sicht befragt. Verändert sich die Ergebnismenge nicht mehr, so ist die Suche auf die bestmögliche Menge von k für die gesuchte ID zuständigen Knoten konvergiert. Das Suchen von Daten baut auf dem Wissen über die zuständige

Algorithmus 4.6.1 $n.lookup(id)$

Benötigt: n ist ein Knoten im System, id eine vom Format her gültige Identifikation.

Stellt her: Gibt die Menge der k am nächsten an id liegenden Knoten zurück.

```

 $B \leftarrow n.closest(id, \alpha);$ 
 $D \leftarrow \emptyset;$ 
 $E \leftarrow \text{undef};$ 
while ( $D \neq E$ ) do
   $D \leftarrow E;$ 
  for all  $x \in B$  do
     $T \leftarrow x.find\_node(id);$ 
     $E \leftarrow E \cup T;$ 
  end for
   $B \leftarrow E;$ 
end while
while  $\#D > k$  do
   $D \leftarrow D \setminus \{y \in D \mid d(y, id) > d(z, id); \forall y \neq z \in V\};$ 
end while
return  $D;$ 

```

Algorithmus 4.6.2 $n.closest(id, \alpha)$, deterministische Version

Benötigt: n ist ein Knoten im System, id eine vom Format her gültige Identifikation, $\alpha \in \mathbb{N}$.

Stellt her: Gibt α lokal bekannte Knoten zurück, die id relativ nahe bzw. am nächsten sind.

```

 $x \leftarrow d(n, id);$ 
if  $\#(n.bucket[\lceil \log_2 x \rceil]) = k$  then
   $D \leftarrow n.bucket[\lceil \log_2 x \rceil];$ 
  return  $D;$ 
else
   $D \leftarrow \bigcup_{j=0}^{m-1} n.bucket[j];$ 
  while  $\#D > \alpha$  do
     $D \leftarrow D \setminus \{y \in D \mid d(y, id) > d(z, id); \forall y \neq z \in V\};$ 
  end while
end if
return  $D;$ 

```

Knotenmenge auf:

Definition 4.6.2 (Zuständigkeit in Kademia) Für das Speichern eines bestimmten Objekts $A \in \mathcal{A}$ mit der Identifikation $id(A) \in \Sigma^*$ sind diejenigen k Knoten zuständig, deren Entfernung von der ID $h(id(A))$ des Objekts in der XOR-Metrik $d(x, y) = x \oplus y$ am geringsten ist. Die Menge ist aufgrund der Eindeutigkeit der Metrik d ebenfalls eindeutig entscheidbar.

Algorithmus 4.6.3 $n.find_node(id)$ **Benötigt:** n ist ein Knoten im System, id eine vom Format her gültige Identifikation.**Stellt her:** Gibt die k lokal bekannten Knoten zurück, die am nächsten an id liegen. $D \leftarrow n.closest(id, k);$ **return** D ;

Will also nun ein Knoten auf das Objekt A zugreifen, so startet er eine Anfrage auf die ID $h(id(A))$ mittels der Prozedur `lookup()`. Da per Definition die k zur ID nächsten Knoten zurückgegeben werden und außerdem das Objekt per Definition auf allen diesen Knoten gespeichert sein muss, kann der anfragende Knoten nun einen Knoten aus der Menge auswählen, um tatsächlich an die Daten zu gelangen. Hierbei ist zum einen eine weitere Möglichkeit gegeben, Probabilistik in das System zu bringen, zum anderen kann auf physikalische Nähe Rücksicht genommen werden und etwa der Knoten mit der geringsten Latenz, der billigsten Verbindung oder der höchsten Bandbreite selektiert werden.

Tolerierte Fehler und Redundanz

In Kademia existiert eine systemweite Konstante α , die angibt, wieviele Verbindungen gleichzeitig geöffnet werden sollen, um ein und dasselbe Ziel zu erreichen. Beispielsweise wird beim Lookup nicht nur eine, sondern gleichzeitig α Verbindungen verwendet. Durch die Verwendung asynchroner, redundanter Verbindungen, fallen nicht funktionierende oder langsame Knoten weniger ins Gewicht: Zum einen bleibt die Anfragezeit niedrig, zum anderen erhöht sich die Fehlertoleranz. Dies alles wird aber einem um den Faktor α erhöhten Nachrichtenaufwand bezahlt. Im Paper zu Kademia [29] wird als Standardeinstellung $\alpha = 3$ genannt.

Um zu wissen, was in Kademia noch als tolerierbarer Fehler gilt, muss der Begriff der Konsistenz in Kademia eingeführt werden. Hierzu rufen wir uns noch einmal die Aufteilung der Knotenmengen in Erinnerung: Der Binärbaum aller Knoten (Abbildung 4.6.16) kann auf jeder Ebene in einen Teilbaum unterteilt werden, der den eigenen Knoten enthält, und einen, bei dem dies nicht der Fall ist. Bis man den eigenen Knoten selbst erreicht, lassen sich also mehrere, sukzessiv kleinere Teilbäume finden, die alle den eigenen Knoten nicht enthalten.

Definition 4.6.3 (Konsistenz in Kademia) *Das System ist bei Kademia in einem konsistenten Zustand, wenn jeder Knoten pro Teilbaum, in dem er selbst nicht enthalten ist, wenigstens einen Knoten kennt oder der entsprechende Teilbaum keine Knoten enthält.*

Im Bezug auf Daten ist ein Verlust erst dann spürbar, wenn die für ein bestimmtes Objekt zuständigen k Knoten alle *gleichzeitig* ausfallen. Laut der Kademia-Dokumentation soll aber k genauso hoch gewählt werden, dass unter den gegebenen Voraussetzungen ein solcher simultaner Ausfall unwahrscheinlich ist.

Im Hinblick auf das Routing wäre ein einzelner Knoten erst dann vollkommen handlungsunfähig, wenn er überhaupt keinen Kontakt mehr zu anderen Knoten besitzt. Denn bereits mit einem bekannten Knoten können die internen Strukturen wieder bis zur vollen Operationsfähigkeit aufgefrischt werden. Hierzu vergleiche man den Join eines Knotens, bei dem nichts anderes vorliegt.

Stabilisierung

Die bei Kademia implizit stattfindende Stabilisierung profitiert von der Tatsache, dass jeder Knoten mit jedem beliebigen anderen Knoten kommunizieren darf. Deshalb kann auch jeder anfragende

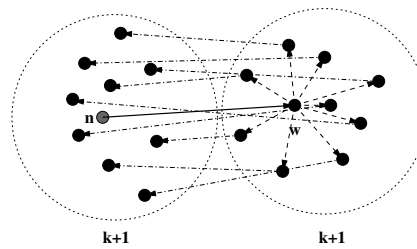


Abbildung 4.6.17: Der skizzenhafte Ablauf eines Joins: Anfrage durch n beim Erstkontakt w , Rückgabe von k Knoten, Suche konvergiert auf k Knoten in der Nähe des hinzukommenden Knotens. Überschneidung der Mengen wahrscheinlich.

Knoten als potenzieller Kommunikationspartner gewertet werden und folglich besitzt jede eingehende Kommunikation gleichzeitig (neben ihrem eigentlichen Zweck) Informationsgehalt für den Stabilisierungsalgorithmus.

Ein Knoten, von dem eine Nachricht empfangen wird, wird jeweils an die Spitze des zuständigen LRU-Buckets gesetzt. Existiert er bereits im Bucket, so wird sein Eintrag an die Spitze verschoben. Ist der Bucket schon voll und nicht mehr teilbar, so wird der älteste Knoten in der Liste geprüft: Ist er nicht mehr erreichbar, so wird er entfernt und der neue Kontakt an die Spitze der Liste eingefügt. Reagiert er noch, so wird der neue Kontakt verworfen. Durch das Beibehalten bewährter, zuverlässiger Knoten wird zum einen der Tatsache Rechnung getragen, dass Knoten mit hoher Onlinezeit eine geringere Ausfallwahrscheinlichkeit zeigen (siehe Abbildung 4.6.18 und den Abschnitt später im Text). Zum anderen ist es durch einen Denial-of-Service Angriff nicht möglich, den Knoten mit neuen Kontakten zu fluten und ihn dadurch seine funktionierenden Kontakte vergessen zu lassen.

Neben der impliziten Stabilisierung existiert für den Fall geringer Netzwerklast eine explizite Stabilisierung, auf die praktisch automatisch umgeschaltet wird: Wurde ein bestimmtes Zeitintervall lang kein Zugriff auf Knoten eines bestimmten Buckets gemacht, so wird mit der Prozedur `refresh_bucket()` (Algorithmus 4.6.5 zufällig eine ID im Bereich dieses Buckets ausgewählt und ein Aufruf von `lookup()` darauf gemacht. Dies führt zur Auffrischung des Buckets, weil meist auch noch andere Knoten im Rahmen der Anfrage berührt werden. Auch diese Prozedur bringt probabilistisches Verhalten in Kademia hinein.

Join und Leave

Um zum System hinzuzukommen, muss ein Knoten n das Problem des ersten Kontaktes bereits gelöst haben. Der erste Kontakt wird im weiteren als w bezeichnet. n startet jetzt eine Lokalisierung seiner eigenen Identifikation mit dem RPC `w.lookup(n)`. Im Algorithmus 4.6.4 wird die Rückgabe des Aufrufs nicht explizit weiterverarbeitet, da wir von einer automatischen Eintragung der Rückgabemenge in die k Buckets durch die implizite Stabilisierung ausgehen. Nachdem durch die Anfrage der nächste Nachbar herausgefunden wurde, ist auch bekannt, ab welchem Bucket überhaupt mit einer Belegung zu rechnen ist. Diese Buckets werden anschließend mit der Prozedur `refresh_bucket()` (Algorithmus 4.6.5) aufgefrischt.

Über das Verlassen der Knoten wird in der Dokumentation zu Kademia nichts weiter erwähnt, so dass die Leave-Operationen dem System als Knotenausfälle erscheinen: Ein abgemeldeter Knoten reagiert nicht mehr auf Anfragen und wird nach einiger Zeit aus den LRU-Listen entfernt.

Algorithmus 4.6.4 $n.join(w)$

Benötigt: w nimmt bereits am Netzwerk teil.**Stellt her:** Nimmt n in das Netzwerk auf.

```

 $w.lookup(n)$ ;
 $i \leftarrow 0$ ;
while  $\#(n.bucket[i]) = 0$  do
   $i \leftarrow i + 1$ ;
end while
for  $j = i$  to  $m$  do
   $n.refresh\_bucket(j)$ ;
end for

```

Algorithmus 4.6.5 $n.refresh_bucket(i)$

Stellt her: Frischt den Bucket mit der Nummer i auf.

```

 $j \leftarrow \text{random}(2^i) + 2^i$ ;
 $n.lookup(j)$ ;

```

Verwaltungsaufwand (Komplexität)

Bei der Komplexitätsbetrachtung fällt der Redundanzparameter α nicht ins Gewicht. Dies liegt daran, dass der zeitliche Aufwand der erfolgten Kommunikation in Hops gemessen wird, und nicht der Gesamtaufwand an Nachrichten. α parallele Anfragen haben in den Hops die selbe Komplexität wie eine einzelne Anfrage.

Finden eines Knotens (C_F): Die Lokalisierung eines Knotens beginnt mit dem Aussenden von α parallelen Anfragen an Knoten aus einem geeigneten, dem Ziel nahe liegenden Bucket. Diese werden jeweils mit einer Menge von k Knoten beantwortet. Da die Anfragen parallel laufen, ist die Länge einer Anfrage repräsentativ für die Dauer dieser Phase. Anschließend wird an jeden der k Knoten, die von allen zurückgelieferten Knoten der gesuchten ID am nächsten sind, wieder je eine Anfrage gestellt, ebenfalls parallel.

In der iterativen Verbesserung der Anfrageergebnisse werden nun so lange die Ergebnisse verfeinert, bis sich die Menge der abgefragten Knoten nicht mehr verändert, d. h. die besten k Knoten gefunden sind. Wie schnell diese Suche konvergiert, wird von der Struktur der k -Buckets beeinflusst: Da die durch die Buckets erfassten Entfernungen exponentiell steigen, wird bei jedem Schritt der Iteration mindestens die Hälfte des Wegs zum Zielknoten zurückgelegt. Der erste Schritt (Anfragen von α Knoten) besitzt ebenfalls schon diese Eigenschaft, weshalb das Finden eines Knotens in Kademia den Aufwand $C_F = \mathcal{O}(\log_2 N)$ besitzt.

Größe der Verwaltungsstrukturen: Kademia benutzt $\log_2 N$ viele Buckets, von denen jeder k Einträge enthält. Jeder Eintrag wiederum ist ein Tupel, das aus drei Wörtern besteht. Daher ist die Größe der Verwaltungsstruktur pro Knoten $D_V = 3k \cdot \log_2 N$, was noch in $\mathcal{O}(\log_2 N)$ liegt.

Kosten, ein Objekt zu lesen: Das Lesen eines Objekts ist in Kademia auch der Ablauf eines Lokalisierungsalgorithmus, nur dass im Gegensatz zum Finden eines Knotens nun der `find_value`-RPC statt des `find_node`-Aufrufs verwendet wird. Dies bedeutet, dass der Algorithmus terminiert, sobald ein Knoten nicht mehr die Menge von k Knoten, sondern gleich das angefragte Datum

zurückliefert. Es ist einsichtig, dass das Lesen eines Objekts höchstens so lang dauern kann wie das Finden der idealen Knotenmenge zu einer gegebenen ID. Im schlimmsten Fall dauert es genauso lange. Wir setzen daher $C_R = \mathcal{O}(\log_2 N) + C$, wobei die Konstante C die Kosten für das Kopieren des Objekts vom entfernten Rechner angibt.

Kosten, ein Objekt einzustellen: Das Einstellen eines Objekts setzt voraus, dass die k am nächsten an der ID liegenden Knoten bereits gefunden sind, also ist der Aufwand C_F ebenfalls wieder zu zahlen. Hinzu kommt, dass das Objekt an jeden der k Knoten kopiert werden muss, was also einen Gesamtaufwand von $C_I = \mathcal{O}(\log_2 N) + k \cdot C$ bedeutet.

Kosten, ein Objekt zu entfernen: Kademia verwendet dynamische Speicherung von Objekten. Dies bedeutet, dass die Kopien ohnehin regelmäßig aufgefrischt werden müssen, um nicht ungültig zu werden. Somit sind auch für das Entfernen eines Objekts keine weiteren Schritte zu unternehmen, als die Auffrischung nicht mehr durchzuführen. Es gilt also $C_D = 0$.

Anzahl der Knoten, die nach Join/Leave Strukturen updaten: Beim Hinzukommen eines Knotens u zum System kann im Schlimmsten Fall jeder Knoten, mit dem u kommuniziert, seine Verwaltungsstrukturen auffrischen müssen. Dies liegt an der symmetrischen XOR-Metrik, die es prinzipiell jedem Knoten erlaubt, jeden beliebigen anderen Knoten als Kontakt zu führen. Alle Knoten, mit denen u beim Hinzukommen Kontakt hat, sind: Der erste Kontakt, seine k genannten Knoten, die $\mathcal{O}(\log_2 N)$ Knoten auf dem Weg zu den k besten Knoten, und die k besten Knoten selbst. Somit ist $N_J = 1 + 2k + \mathcal{O}(\log_2 N)$ und ebenfalls $C_J = 1 + 2k + \mathcal{O}(\log_2 N)$.

Knoten, die das System verlassen, melden sich nicht gesondert ab. Wenn ihr Fehlen bemerkt wird, werden stattdessen die Ergebnisse anderer, parallel angefragter Knoten herangezogen und die Knoten werden ausgetragen. Es entstehen somit beim Verlassen keine Kommunikationskosten für Strukturupdates.

Berücksichtigung der Verfügbarkeit einzelner Knoten

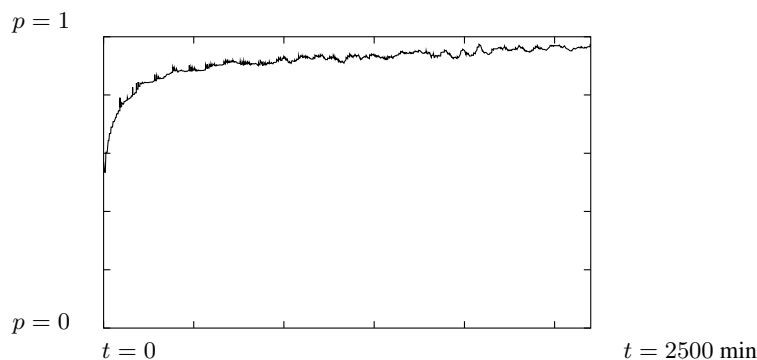


Abbildung 4.6.18: Wahrscheinlichkeit für einen Knoten, der bereits x Minuten online ist, auch weitere 60 Minuten online zu bleiben. Nach rechts ist die Zeitachse, nach oben die Wahrscheinlichkeit aufgetragen.

Bei Kademia fließen erstmalig direkt Betrachtungen der Verfügbarkeit von Knoten aufgrund von statistischen Untersuchungen am Gnutella-Projekt [17] mit ein. Die zentrale Aussage jener Untersuchungen ist, dass sich mit steigender bereits erreichter Verfügbarkeit eines Knotens die

Wahrscheinlichkeit für einen Ausfall weiter verringert, wie in Abbildung 4.6.18 dargestellt. Da in diesem Algorithmus erreichbare Knoten nie aus dem k -Bucket entfernt werden, solange sie weiter erreichbar sind, enthält ein solcher Bucket nach einiger Laufzeit eine spezifische Auswahl an sehr stabil arbeitenden Knoten. Dieses Vorgehen realisiert eine qualitätsbezogene Gewichtung im Bezug auf Verfügbarkeit der Knoten.

Flexible Wahl des UDP-Ports: Kademia bezieht bei der Beschreibung eines Nachbarknotens durch das Tripel (IP-Adresse, UDP Port, Knoten-ID) gleich den UDP-Port mit ein. Wissenschaftlich zwar uninteressant, gewährt der wählbare Port dem Anwender in Zeiten von Firewalls und selektiver Sperrung von Ports ein angenehmes Maß an Flexibilität.

Frei wählbare ID: Die Autoren lassen es auch bei Kademia offen, auf welchem Weg die IDs von Knoten zustandekommen. Im Hinblick auf Denial-of-Service Attacken empfehlen sich hier wie bei den anderen betrachteten Algorithmen nachprüfbar (z. B. mittels der Hashfunktion aus der IP generierte) IDs, wie etwa in Chord (siehe auch Anmerkungen zum Commitment in Abschnitt 5.3). Es bleibt aber anzumerken, dass eine solche Attacke gegen Kademia nicht in dem Maße erfolgreich wäre wie bei anderen Systemen, da durch die LRU-Strukturen der Buckets neue Knoten nur zögerlich in die Datenstrukturen der Knoten aufgenommen werden, und der Kern des Systems, der sich mit der Zeit aus den permanent online bleibenden Knoten bildet, von solchen Angreifern gar keine Notiz mehr nimmt.

Kapitel 5

Fazit

Dieses Kapitel präsentiert Erkenntnisse über die Peer-to-Peer Algorithmen, die erst aus der Analyse in Kapitel 4 hervorgegangen sind. Weiterhin werden wichtige Bestandteile strukturierter Peer-to-Peer Algorithmen näher untersucht, beispielsweise der Einfluß von Metriken und Hashfunktionen auf das Funktionieren des Gesamtsystems. Alle Daten der untersuchten Systeme sind am Schluß des Kapitels in einer übersichtlichen Gesamtschau in Tabelle 5.6.1 zusammengefaßt.

5.1 Betrachtung von Peer-to-Peer-Systemen als metrische Räume

Unabhängig von der zugrundeliegenden Struktur des Systemmodells (Graph, Baum, Torus, Ring) ist bei den untersuchten Algorithmen immer eine eindeutige Identifikation von Daten und Teilnehmern innerhalb des verwendeten Raums vorhanden. Ebenso werden Routing-Entscheidungen aufgrund von einer logischen Nähe von Knoten im System getroffen.

Um verschiedene strukturierte Peer-to-Peer Algorithmen zu vergleichen, bietet es sich also immer an, die Systemmodelle auf einen Raum zu abstrahieren und die Eigenschaften der verwendeten Metriken zu vergleichen. In vielen Papers wird das Konzept der Metrik eher unbewußt verwendet und nicht explizit erwähnt. Ist die vorliegende Metrik erst kategorisiert, lassen sich direkte Rückschlüsse auf das Verhalten des Algorithmus ziehen.

Bei den untersuchten Systemen fiel beispielsweise auf, dass jedes einen anderen Entfernungsbegriff realisiert. Nur CAN, Pastry und Kademlia besitzen beispielsweise eine im mathematischen Sinne echte Metrik. Eine eindeutige Entfernungsangabe (so dass mit jeder Entfernung Δ zum lokalen Knoten n nur ein Knoten y assoziiert werden kann mit $d(n, y) = \Delta$) besitzen nur Chord und Kademlia. Wie in Tabelle 5.6.1 am Ende des Kapitels deutlich wird, sind die verwendeten Entfernungsbegriffe auch sehr verschieden. Als einzige Autoren gehen MAYMOUNKOV und MAZIÈRES in ihrer Arbeit über Kademlia [29] explizit auf den Begriff der Metrik in Peer-to-Peer Algorithmen ein. Bei den anderen Systemen wird der Terminus nicht bewußt verwendet. Deshalb war es dort auch eine anspruchsvolle Aufgabe, die Formel für die Entfernung zu exzerpieren. Tapestry war in dieser Hinsicht am aufwendigsten zu charakterisieren, da jeder Hinweis auf Entfernung in den Arbeiten [58, 57] fehlte.

5.2 Problem des ersten Kontakts ungelöst

Allein die Tatsache, dass in allen betrachteten Arbeiten die Existenz eines bereits bekannten, dem System angehörenden Knotens einfach vorausgesetzt wird, wenn sich ein neuer Knoten dem Sys-

tem anschließen will, deutet darauf hin, dass das Problem des *ersten Kontakts* weder Gegenstand der Betrachtung ist, noch trivial lösbar. Obwohl sich dieses Problem nur einmal, nämlich vor dem Join eines Knotens stellt, hat es viele Autoren beschäftigt und vielfältige Ideen zur Lösung hervorgebracht. Es existieren einige Hinweise in der Literatur, wie das Problem in der Praxis zu umgehen wäre, und einige Möglichkeiten sind aus dem Stegreif denkbar. Die meisten Lösungen laufen allerdings auf den Zugriff auf einen zentralisierten Dienst hinaus, was dem Konzept dezentraler Peer-to-Peer Algorithmen widerspricht.

Round Robin DNS

Eine Möglichkeit für einen Teilnehmer an einem Peer-to-Peer Netz, einen ersten Kontakt zu finden könnte die Anfrage an einen Internet-Nameserver sein. Solche Zuordnungen liefern prinzipiell für einen eingegebenen Hostnamen die zugehörige IP-Adresse zurück. Im Falle von Lastverteilungskonzepten und bei Hochverfügbarkeitssystemen kommt es vor, dass pro Anfrage auf den gleichen Hostnamen unterschiedliche IP-Adressen zurückgegeben werden. Deshalb könnte ein für das Peer-to-Peer System zuständiger Nameserver eine Teilmenge der momentan im System aktiven Knoten in seinem Speicher halten, und auf Anfrage eine beliebige daraus zurückliefern. In der Arbeit zu CAN [44] wird auf ein System namens YOID [16] verwiesen, das sich dieser Methode bedient.

Dieses Konzept ist in der Komponente des DNS-Servers wieder zentral, aber für die Praxis akzeptabel, weil die Internet-Infrastruktur zum einen mehrere DNS-Server für eine Domain fordert¹, und zum anderen, weil fast jedes Netzwerk einen DNS-Server besitzt, der nur einige Hops entfernt steht. Anfragen werden also zwischengespeichert und der Hauptserver dadurch entlastet. Ausfälle des Hauptservers können dadurch vor den Klientenrechnern für kurze Zeit transparent gehalten werden. Bei einem Ausfall der Verbindung zum lokalen DNS-Server ist meistens auch die Anbindung zu allen potentiellen Peers unterbrochen, was ohnehin zu einem Ausschluß aus dem System führt.

Modifiziertes DNS: Zeroconf

Eine Initiative, die inzwischen auch bei Apple-Produkten unter dem Namen *Rendezvous* Verwendung findet, zielt darauf ab, den DNS-Server um zusätzliche Funktionalität zu erweitern, die über DNS-Anfragen abgefragt werden kann, die vom Standard abweichen. Das auf der Website [4] näher erläuterte Konzept wird dazu verwendet, den Konfigurationsaufwand in IPv4 Netzen so gering wie möglich zu halten. Denkbar wäre hier auch eine Nutzung durch Peer-to-Peer Anwendungen.

IPv4: Portscan im lokalen Netz

Es wäre auch denkbar, das komplette Subnetz, in dem sich der Knoten, der partizipieren möchte, befindet, nach offenen Ports der Peer-to-Peer Anwendung zu durchsuchen. Allerdings erscheint diese Methode wenig praktikabel. Einerseits, weil trotz aktivem Peer-to-Peer System und funktionierender Netzanbindung kein Knoten gefunden wird, wenn es im lokalen Netz nicht ohnehin schon einen gibt. Dies bedeutet, dass ein Peer-to-Peer System sich mit dieser Methode überhaupt nicht über Subnetze hinaus verbreiten könnte. Andererseits, weil Portscans in Netzwerken von manchen Systemadministratoren ungern gesehen werden. Außerdem ist dieser Ansatz extrem von der verwendeten Netzwerktechnik abhängig, auch wenn IP-Netze sehr verbreitet sind.

¹ siehe RFC 1034 und 1035 [32, 33].

Ein Beispiel für Broadcasts im lokalen Netzwerk ist das SMB-Protokoll², das unter den Namen *Lan Manager*, *NetBIOS* oder *CIFS* bekannt wurde. Ohne zentralen Server finden sich die Computer eines lokalen Netzwerks auch über Broadcasts (hier speziell auf dem UDP-Port 137).

IPv6: Multicast-Gruppen

Ein neuer, sehr vielversprechender Gedanke ist dagegen das Nutzen der in der neuen Protokollversion von IPv6 [2] existierenden Multicast-Gruppen. Bekanntlich ist es in einem IPv6-Netz möglich, auch ohne vorherige Konfiguration oder ein externes Wissen sofort an einem lokalen Netzwerk zu partizipieren. Ermöglicht wird dies, weil IPv6 die Funktionalität des unter IPv4 bekannten DHCP⁴ in ähnlicher Form bereits direkt im Protokoll realisiert. So kann man beispielsweise durch eine Anfrage im lokalen Netz sofort den zuständigen Gateway in Erfahrung bringen, ebenso wie den nächsten DNS-Server per Anycast-Anfrage.

Auf Peer-to-Peer Systeme angewendet bedeutet das, dass man auch den nächstgelegenen Knoten eines Peer-to-Peer Netzes auf diese Weise per Multicast erfragen kann. Damit wäre prinzipiell das Problem des ersten Kontakts gelöst. Einziger Nachteil dieser Methode ist wieder die Abhängigkeit von einer bestimmten Netzwerktechnik, in diesem Fall also IPv6 mit Multicast-Funktionalität. Das Problem bei dieser Technik ist die bisher mangelnde Akzeptanz. So wird Multicast beispielsweise noch nicht durchgehend geroutet.

Hätte man eine global geroutete Multicast-Adresse, würden allerdings wieder andere Schwierigkeiten vorliegen: Wenn jeder Knoten seine Erstkontakt-Anfragen an alle Knoten im System schicken könnte, entsteht wieder ein erheblicher Nachrichtenaufwand, der verarbeitet werden muss, da wohl auch jeder angesprochene Knoten eine Antwort schicken kann. Diese Antwort wäre aber Unicast, und so hätte man für einen Join gleich N Nachrichten verschickt, alleine um Kontakt aufzunehmen.

Exhaustive Suche

Der Vollständigkeit halber sollte erwähnt sein, dass im schlimmsten Fall jede mögliche Netzwerkadresse direkt angefragt werden kann, ob sie von einem Knoten aus dem Peer-to-Peer Netz bedient wird. Dieses Verfahren ist korrekt und unabhängig von der darunterliegenden Netzwerktechnik, da jede Adressfamilie (ob nun IP, IPX, Appletalk oder andere) eindeutige und aufzählbare Netzwerkadressen zur Verfügung stellt. Allerdings ist es praktisch vollkommen nutzlos, da eine exhaustive Suche über einen derart großen Raum viel zu lange dauert, viel zu viel Netzwerklast verursacht und ausserdem als weitere Art des Portscans betrachtet wird.

5.3 Auswirkungen und Anforderungen an die Hashfunktionen

Die Verwendung einer Hashfunktion ist für einen strukturierten Peer-to-Peer Algorithmus praktisch unabdingbar. Hier fassen wir zusammen, was eine Hashfunktion für ein Peer-to-Peer System leisten muss, und welche Vorteile sich daraus ergeben.

²Spezifikation unter <ftp://ftp.microsoft.com/developr/drg/CIFS/>

³IPv6 definiert in RFC 3513 [23].

⁴Dynamic Host Configuration Protocol, definiert in RFC 2131 [14]

Streuung und Lastverteilung

Da wir von der Verwendung einer kryptographisch brauchbaren Hashfunktion (siehe Anhang C.1) ausgehen, sind die Ergebnisse statistisch gleichverteilt im Bildraum. Die bei den untersuchten Algorithmen verwendeten Hashfunktionen SHA-1 und MD5 entsprechen den erwähnten Anforderungen. Weil die Algorithmen mittels der Hashfunktionen die Lokalisierung im Systemmodell vornehmen, und auch die Knoten-IDs noch per Hashfunktion gleichverteilt sind, ist mit sehr großer Wahrscheinlichkeit eine gute Lastverteilung der Anfragen auf die verschiedenen Knoten gegeben.

Kollisionsfreiheit

Eine weitere wichtige Eigenschaft ist, dass nicht mehrere Knoten oder lokalisierbare Ressourcen unter der selben ID abgespeichert werden. Dies ist nur dann gegeben, wenn die Hashfunktion kollisionsfrei arbeitet, d. h. wenn es rechnerisch praktisch unmöglich ist, für ein beliebiges Bild der Funktion noch ein zweites Urbild zu finden. Auch diese Anforderungen werden von den verwendeten Funktionen erfüllt.

Commitment: Nachprüfbarkeit von IPs

Commitment ist ein Fachbegriff aus der Kryptographie und bezeichnet die vorherige Festlegung (meist per Hashwert) auf eine noch zu sendende Nachricht.

In den Arbeiten zu Systemen wie CAN oder Pastry wurde darauf hingewiesen, dass die Knoten-ID vom neu hinzukommenden Knoten *zufällig* gewählt werden soll. Dies läßt aber dem Knoten prinzipiell freie Wahl über die ID, also kann auch Mißbrauch in Form von gezielt schlecht gewählten Folgen von IDs getrieben werden, die etwa die Gleichverteilung der IDs im Gesamtraum aufheben und damit die Dienstgüte beeinträchtigen (Denial-of-Service). Auch ohne böse Absicht kann die freie Wahl der ID durch schlechten Zufall beeinträchtigt werden. Auf manchen Systemen steht beispielsweise kein oder nur ein pseudo-Zufallsgenerator mit wenig Input zur Verfügung. Auch hier sei wieder auf eine Arbeit über gute Pseudozufallsgeneratoren verwiesen [22, Kapitel 6].

In beiden Fällen ist die Verwendung einer Hashfunktion die bessere Alternative. Fixiert man den Knoten auf ein Ergebnis aus der auf die IP-Adresse und einen wählbaren Parameter angewendete Hashfunktion (wie z. B. bei Chord), so ist für jeden der anderen Knoten nachprüfbar, ob die ID auf einem anerkannten Weg zustande gekommen ist, Mißbrauch ausgeschlossen. Auch ist durch ein Verändern des wählbaren Parameters trotzdem keine Häufung von IDs im Gesamtraum möglich, da hierzu Kollisionen in der Hashfunktion gefunden werden müßten. Dies haben wir aber per Definition ausgeschlossen.

Zusammenfassend sind die drei großen Vorteile der richtigen Hashfunktion die Gleichverteilung, die Unvorhersagbarkeit des Ergebnisses, ohne die Funktion schon einmal ausgeführt zu haben (auch als Einwegberechenbarkeit bekannt) und die Nachprüfbarkeit eines bekannten Ergebnisses durch Ausrechnen.

Angriff auf verteilte Hashtabellen: Partielle Kollisionen

Wie wir aus Anhang C.1 wissen, ist eine exakte Kollision einer Hashfunktion hinreichender Outputlänge praktisch nicht herbeizuführen. Betrachtet man aber den Einsatzzweck in Peer-to-Peer Algorithmen, so fällt auf, dass die Hashfunktion meistens nur den Beginn eines Bereiches markiert, in dem ein bestimmter Knoten für Anfragen zuständig ist.

Für einen erfolgreichen Denial-of-Service Angriff auf einen Knoten würde es also vollkommen ausreichen, wenn ein Treffer in den Bereich gelingt, für den er zuständig ist. Hier kann sehr gut von der Aufteilung des Adreßraums profitiert werden. In Chord ist z. B. der modulare eindimensionale Adreßraum für große N statistisch gleich unter den N Knoten verteilt. Also bedient jeder Knoten im Mittel $1/N$ des gesamten Adreßraums.

Beispiel 5.3.1 (Angriff auf Chord) Nehmen wir ein $N = 2048$ an, so entfällt im Mittel auf jeden Knoten ein Adreßraum von

$$2^{160 - (\log_2 2048)} = 2^{149}.$$

Also benötigen wir für eine erfolgreiche Attacke lediglich 11 Bit Übereinstimmung mit dem Key des anzugreifenden Knotens. Diese 11 Bit sind mit akzeptabler Rechenzeit zu erreichen. Unser Vorgehen ist es nun, von vielen virtuellen Servern oder von einem großen lokalen Netzwerk (um die Quota-Beschränkung von Chord zu umgehen) ständig Speicheranforderungen für DHash-Datenblöcke abzusetzen, die auf 11 Bit Präfixlänge mit dem angegriffenen Knoten kollidieren. Irgendwann wird dem angegriffenen Knoten der Speicherplatz ausgehen, um den echten Anforderungen nachzukommen.

5.4 Vorteile redundanter Verbindungen

Beim Kademia-Projekt eingeführt wurde die Maßnahme, prinzipiell mehr Verbindungen zum Erreichen eines bestimmten Ziels zu öffnen, als wirklich notwendig wären. Das gleichzeitige Abfragen verschiedener Knoten bietet mehrere Vorteile: Unterscheiden sich die Antworten der Knoten, so kann durch eine demokratische Auswertung der Antworten (z. B. bei $\alpha = 3$, sog. *triple modular redundancy*) das richtige Ergebnis doch herausgefunden werden (außer, wenn die Mehrzahl der Knoten fehlerhaft arbeitet), und böswillige Auskünfte byzantinisch fehlerhafter Knoten werden nicht berücksichtigt. Zudem erfährt man einen Geschwindigkeitsgewinn im Falle von ausgefallenen Knoten. Würde man im nicht redundanten Modell zunächst auf die Antwort des fehlerbehafteten Knotens warten, so kann bei Redundanz nach der ersten gültigen Antwort schon weitergearbeitet werden. Ein markanter Nachteil des Konzepts ist das Wachstum des Kommunikationsaufwands im Netzwerk um den Redundanzfaktor α .

Verbesserung durch Probabilistik

Weiter steigern lässt sich der Effekt, wenn auch die herausgesuchten α Knoten nicht nach einem deterministischen Muster, sondern zufällig aus einer Menge potenzieller Kandidaten selektiert werden. Hierdurch wird Anfragelast verteilt, und durch den probabilistischen Effekt werden fehlerhafte Knoten, die beim deterministischen Verfahren immer ausgewählt würden, mit einer gewissen Wahrscheinlichkeit umgangen. Gerade im Hinblick auf die reale Situation mit permanenter Knotenfluktuation macht ein solches Verfahren Sinn.

5.5 Replikation, Lastverteilung und Anfrageoptimierung

Die Analyse der Systeme hat gezeigt, dass im Wesentlichen zwei Mechanismen existieren, die beide Kopien von Objekten erstellen, aber in ihrer Zielsetzung unterschiedlich sind.

Replikation

Die Replikation von Objekten im System stellt eine echt fehlertolerante Maßnahme dar. Ihr Hintergrund ist, dass ein Objekt verloren ist, sobald kein Replikat mehr im System kursiert. Die Replikation ist deshalb eine im Vorhinein wirkende, statische Aktion, die immer ausgeführt wird, um das fehlertolerante Funktionieren des Systems zu sichern. Nur als positiver Nebeneffekt ist die Haltung von Replikaten auch lastverteilend, der Zweck ist ein anderer.

Caching

Das Caching von Objekten ist zwar auch eine replizierende Maßnahme, jedoch mit einem gänzlich anderen Fokus: Die Lebensdauer von Cacheobjekten ist meist auf einen kurzen Zeitraum begrenzt und das Caching selbst findet als Reaktion und nicht als Aktion statt. Ziel der Maßnahme ist nicht eine höhere Fehlertoleranz des Systems, sondern eine Lastverteilung und Anfrageoptimierung durch kürzere Wege bis zum Auffinden eines Objekts. Durch Caching lassen sich sogenannte *hot spots* zerstreuen, also Knoten entlasten, die oft angefragte Daten bereithalten.

Als sehr effizient hat sich das Caching entlang des Anfragepfads einer Lokalisierung erwiesen: So werden nicht nur die für das Objekt zuständigen Knoten entlastet, sondern Anfragen sind schneller erfolgreich beendet.

5.6 Die Systeme in der Übersicht

Zunächst gehen wir auf die Parametrisierbarkeit der untersuchten Algorithmen ein. Die ungefähre Angabe einer Variable soll hier für die Standardeinstellung bzw. Empfehlung des jeweiligen Algorithmus stehen.

Funktion	Chord	CAN	Pastry	Tapestry	Kademlia
Länge der Nachfolgerliste	$r \approx 320$	-	-	-	-
Redundante Speicherung	$2 \leq k \leq 321$	r	-	-	$k \approx 20$
Parallele Verbindungen	-	-	-	-	$\alpha \approx 3$
Baumbreite	-	-	$2^b \approx 4$	$2^b \approx 4$	2

Anschließend geben wir eine Übersicht über alle Systeme: Tabelle 5.6.1 faßt die Ergebnisse der Analyse aus Kapitel 4 zusammen. Die fünf praktisch relevanten Algorithmen werden im Hinblick auf die bekannten Kriterien aufgelistet.

Unter *Einordnung* ist die Positionierung im Modell der orthogonalen Dimensionen zu verstehen. Es fällt auf, dass die untersuchten Algorithmen in diesem Fall nicht sehr weit auseinanderliegen. Das *Systemmodell* gibt die Struktur des Algorithmus an. Es wird gleich auch auf die Dichte der Vernetzung im Graphen, die verwendete(n) Hashfunktion(en) und die damit häufig korrelierende Breite des Adreßraums in Bit eingegangen. Die *Komplexitätsbetrachtung* gliedert sich folgendermaßen: Gemessen in der Gesamtzahl N der maximal im System vorhandenen Knoten wird untersucht, wie teuer das Lesen, und das Verfügbarmachen und Entfernen eines Objekts ist. Hierzu ist aber jeweils erforderlich, den zuständigen Knoten zu lokalisieren, weshalb auch dieser Wert extra aufgeführt ist. Eine weitere Rolle spielt, wieviel Speicherplatz für die reinen Hilfsdaten des Algorithmus pro Knoten bereitgestellt werden muss. Schließlich interessiert noch, auf wieviele anderen Knoten sich ein Hinzukommen oder Verlassen eines Knotens im System auswirkt. Wichtig ist, dass sich die Komplexitätsbetrachtungen auf den Basisalgorithmus beziehen und nicht auf von den Autoren der

Arbeiten vorgeschlagene Optimierungen und Spezialfälle. Auch kann im Rahmen dieser Arbeit nur eine *worst-case* Betrachtung angestellt werden und nicht die viel schwierigere *average-case* Untersuchung. Das Wort *Metrik* steht für die Wahrnehmung des Entfernungsbegriffs zwischen zwei Knoten im jeweiligen Systemmodell. Nicht immer ist sie im mathematischen Sinne wirklich eine Metrik nach Definition, was durch das Feld „echte Metrik“ gekennzeichnet wird. Die Eindeutigkeit ist die Eigenschaft, dass von einem beliebigen Knoten x aus immer nur höchstens *ein* anderer Knoten y genau die diskrete Entfernung $\Delta = d(x, y)$ hat.

Die nächste Gruppe an Werten beschreibt das dynamische Verhalten im Netz: Alle Systeme beschäftigen sich mit dem *ersten Kontakt* nicht weiter, sondern setzen ihn voraus, da das Problem nicht im Fokus der Arbeiten lag. Das Feld *Stabilisation* gibt an, auf welche Weise die Verwaltungsdaten aktualisiert und konsistent gehalten werden. Die letzte Gruppe geht auf die Redundanz ein, die der Algorithmus bereithält. Einerseits interessiert, wieviele Kommunikationsverbindungen gleichzeitig zum Erreichen desselben Ziels geöffnet werden, zum anderen, wie oft ein Datum im System gespeichert wird.

Alle untersuchten Algorithmen sind **strukturiert** und **homogen**, ebenso setzt jede Ausarbeitung über den Algorithmus einen **ersten Kontakt** voraus und geht nicht weiter darauf ein, wie er zu bekommen ist. Diese Punkte werden in der Tabelle nicht mehr gesondert aufgeführt.

Kriterium	Chord	CAN	Pastry	Tapestry	Kademlia
Systemmodell	Ring	d -Torus	Baum	Baum	Binärbaum
Dichte (max. bekannte Nachbarknoten)	480	$2dr$	$(2 + g)2^b - g$	$4(2^b \cdot g)$	3200
Hashfunktion	SHA-1	beliebig ⁵	SHA-1 ⁶	SHA-1	SHA-1
Adreßraum	160 bit	beliebig ⁵	128 bit	160 bit	160 bit
Algorithmus deterministisch?	☑	☑	☑	☑	variabel ⁷
Kosten für Lokalisierung (C_F)	$\mathcal{O}(\log_2 N)$	$\mathcal{O}(dN^{\frac{1}{d}})$	$\mathcal{O}(\log_{2^b} N)$	$\mathcal{O}(\log_{2^b} N) + 2$	$\mathcal{O}(\log_2 N)$
Größe der Verwaltungsstrukturen (D_V)	$1 + r + \log_2 N$	$2dr$	$(2 + g)2^b - g$	$4(2^b \cdot g)$	$3k \log_2 N$
Kosten, ein Objekt einzustellen (C_I)	$\mathcal{O}(\log N) + (1 + k)C$	k. A.	k. A.	$\mathcal{O}(\log_{2^b} N) + 2 + C$	$\mathcal{O}(\log N) + kC$
Kosten, ein Objekt zu entfernen (C_D)	implizit	k. A.	k. A.	$\mathcal{O}(\log_{2^b} N) + 2 + C$	implizit
Kommunikationskosten für Join (C_J)	$2C + 3\mathcal{O}((\log_2 N)^2)$ (simple) $\mathcal{O}(\log N)$ (concurrent)	$2dr$	$\mathcal{O}(\log_{2^b} N)$	$\mathcal{O}(\log_{2^b} N)^2$	$1 + 2k + \mathcal{O}(\log N)$
Kommunikationskosten für Leave (C_L)	2	$2dr$	implizit	implizit	implizit
#Knotenupdates nach Join (N_J)	$\mathcal{O}(\log^2 N)$ (simple) 0 (concurrent)	$2dr$	$\mathcal{O}(\log_{2^b} N)$	$\mathcal{O}(\log_{2^b} N)$	$1 + 2k + \mathcal{O}(\log N)$
#Knotenupdates nach Leave (N_L)	2	$2dr$	implizit	implizit	implizit
Entfernungsbegriff: $d(x, y) =$	$(y - x) \bmod 2^{160}$	EUKLID	$ x - y $	$\lceil (\log_2((x \oplus y) + 1)/b) \rceil$	$x \oplus y$
echte Metrik	☐	☑	☑	☐	☑
eindeutig	☑	☐	☐	☐	☑
Stabilisation	explizit	explizit	explizit	explizit und implizit	implizit und bei Bedarf explizit
Redundante Anfragen (Faktor)	nein	nein	nein	nein	$\alpha \approx 3$
Redundante Speicherung (Faktor)	$2 \leq k \leq 321$	r	k. A.	durch Applikation	$k \approx 20$
Art der Objektspeicherung	dynamisch	k. A.	k.A.	Objekte statisch, Zeiger dynamisch	dynamisch

Tabelle 5.6.1: Gesamtübersicht über die Eigenschaften der untersuchten Peer-to-Peer Algorithmen

⁵CAN macht keine Vorgaben über die zu verwendende(n) Hashfunktion(e)n.

⁶Widersprüchliche Angaben in der Dokumentation zu Pastry, SHA-1 hasht auf 160 Bit.

⁷Kademlia verhält sich probabilistisch, je nachdem, ob aus den lokalen Knotenmengen zufällig ausgewählt wird oder nicht.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Studienarbeit wurden verschiedene repräsentative Peer-to-Peer Algorithmen hinsichtlich ihrer Funktionalität, Ressourcen in einem verteilten System zu lokalisieren, untersucht. Die entdeckten Auffälligkeiten wurden gesondert von der Analyse in Kapitel 5 diskutiert.

Dabei wurden wichtige, für alle Peer-to-Peer Systeme geltende Gemeinsamkeiten herausgestellt und erläutert. So haben wir beispielsweise Dimensionen aufgestellt, in die untersuchte Algorithmen eingeordnet werden können. Es fiel auf, dass die untersuchten Algorithmen in diesem Koordinatensystem sehr nahe beieinander liegen, da keiner von ihnen hybrides oder unstrukturiertes Peer-to-Peer realisiert. Dies legt die Vermutung nahe, dass die vorgefundene Konfiguration von *strukturierten* und *homogenen* Peer-to-Peer eine sehr günstige ist. Zum einen gewährt die strukturierte Variante eine potenziell vollständige Sicht auf das Netz, zum anderen ist eine Gleichberechtigung aller Knoten einfacher zu realisieren als die Aufgabenteilung im hybriden Fall.

Alle untersuchten Algorithmen sind nicht gegen byzantinische Angriffe abgesichert. Das bedeutet, dass ein Knoten generell von der korrekten Funktion eines anderen, am Protokoll beteiligten Knotens ausgeht. Analog zum IP-Protokoll findet auf dieser Ebene noch keine Prüfung der Vertrauenswürdigkeit und Authentizität der Gegenstelle statt. Deshalb ist es zu begrüßen, dass sich die Forschung bereits dem Thema von Vertrauensnetzen und Reputation in Peer-to-Peer Netzen angenommen hat (z. B. EigenTrust [25]).

Außerdem stellten wir fest, dass jedes Peer-to-Peer System für sich vorteilhafte Funktionalität bietet, leider jedoch kein System mit allen potenziellen Features derzeit existiert. Darüberhinaus lassen sich derartige Algorithmen nach ihren Parametern (strukturiert/unstrukturiert, deterministisch/probabilistisch, hybrid/homogen), ihrem jeweiligen Einsatzgebiet (verwendete Netzwerk- und Rechnerhardware) und ihren Anforderungen (Zuverlässigkeit, wenig verursachter Netzverkehr, schnelle Antwortzeiten) charakterisieren. Anstatt nun den Versuch zu unternehmen, noch ein „bestes“ System mehr ins Leben zu rufen, könnte man neue Wege gehen, und eine Peer-to-Peer Algorithmenfamilie entwerfen, die ein großes Featuremodell beinhaltet. Dass dieses Vorhaben nicht illusorisch ist, zeigt ein Blick auf die modulare Struktur der Peer-to-Peer Algorithmen: Chord beweist uns, dass eine Trennung von Routing und Datenspeicherung sehr sauber möglich ist. CAN demonstriert, dass die Hashfunktion austauschbar ist. In PLAXTONs Arbeit bemerken wir eine hohe Zahl von Freiheitsgraden im Algorithmus, die in den konkreten Systemen meist bereits durch Einsetzen der verschiedenen Parameter reduziert wurde.

Zusammen mit den Vorteilen aspektorientierter Programmierung wäre es somit möglich — idealerweise auf Knopfdruck — für jeden spezifischen Zweck ein schlankes, den Vorgaben angepasstes System zusammenzubauen.

Anhang A

Anwendungen

Manche der untersuchten Algorithmen eignen sich aufgrund ihrer Struktur besser für eine bestimmte Sorte von Anwendungen als andere. Einen generischen „besten“ Algorithmus zu finden ist daher illusorisch, weil immer die gesuchte Anwendung mit berücksichtigt werden muß. Dieser Anhang gibt einen sehr knappen Überblick über bereits realisierte oder mögliche Anwendungsgebiete der betrachteten Systeme. Natürlich besteht kein Anspruch auf Vollständigkeit, da die Untersuchung von Anwendungen nicht Fokus dieser Arbeit ist. Die genannten Systeme sollen vielmehr als Ausblick und Anregung für eigene Lektüre dienen.

A.1 Filesysteme

Die Anforderungen an ein Peer-to-Peer System, um ein Dateisystem zu realisieren, sind vergleichsweise gering: Zuverlässige (also ausfallresistente) Speicherung von binären Blöcken und deren Wiederfinden per ID reichen aus.

CFS (Chord/DHash)

Die PhD-Arbeit von FRANK DABEK [11] beschreibt anschaulich den Aufbau von lesend zugreifbaren verteilten Dateisystemen auf der Basis von Chord und DHash (Abschnitt 4.1). Hierbei können praktisch beliebig viele voneinander unabhängige Dateibäume existieren. Die Pflege eines Dateibaums obliegt einem Autor, der einen für den Baum charakteristischen Public-Key Kryptoschlüssel kennt und die öffentliche Hälfte davon publiziert. Die Wurzel des Dateibaums ist mit dem Schlüssel signiert. Alle Verweise, die die Wurzel beinhaltet, sind mit einem Hashwert versehen, der ebenfalls Gegenstand der Signatur ist. Alle rekursiven Verweise enthalten wieder Hashes über tieferliegende Blöcke. Ein Benutzer kann sich davon überzeugen, dass die Daten authentisch sind, indem er die Signatur der Wurzel und rekursiv alle Hashes bis herunter zu dem Blatt prüft, das er lesen will. Das Prinzip wird in Abbildung A.1.1 veranschaulicht. Ein Angreifer müsste entweder den Kryptoschlüssel brechen oder die Hashfunktion zu einer Kollision führen, um Daten im Baum zu verfälschen. Kryptosystem und Hashfunktion werden aber so gewählt, dass ein Brechen derselben *praktisch unmöglich*¹ ist.

¹ *computationally infeasible*, vgl. Anhang C.1

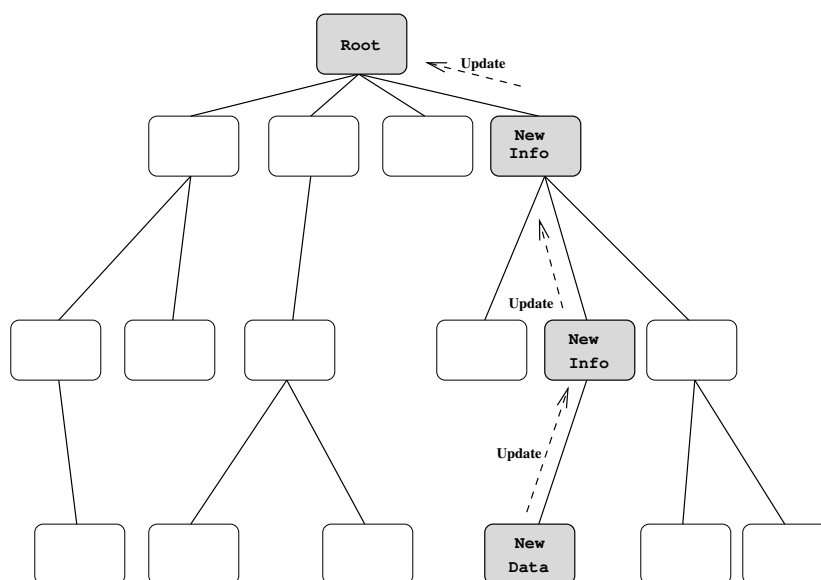


Abbildung A.1.1: Die Struktur von CFS: Eine signierte Wurzel enthält die Hashes aller direkten Unterknoten, ebenso jeder normale Knoten jeweils die seiner Unterknoten. Eine Änderung an beliebiger Stelle im Baum zieht jeweils eine Änderung bis hoch in die Wurzel nach sich.

Ivy (Chord/DHash)

Ivy [34] basiert auf Chord und DHash (Abschnitt 4.1). Es realisiert ein dezentrales, verteiltes Dateisystem, das dem Phänomen nicht vertrauenswürdiger Knoten dadurch begegnet, dass jede Modifikation am Dateisystem durch einen Knoten in dessen eigenes Logfile geschrieben wird. Die Sicht eines bestimmten Knotens entsteht dann aus der Komposition eines oder mehrerer Logs von Knoten, die als vertrauenswürdig angesehen werden. Da die Logs eine Transaktionssemantik realisieren, verhält sich Ivy in manchen Fällen etwas anders als ein herkömmliches Dateisystem. Ivy bietet jedoch Werkzeuge zur automatischen Auflösung von Konflikten, z. B. wenn zwei Knoten unabhängig voneinander die selbe Datei schreiben möchten. Die Entscheidung wird mithilfe von Versionsvektoren, ähnlich wie bei LAMPORTS Uhren oder Vektoruhren in Kommunikationssystemen (vergleiche hierzu [48, Kapitel 5.3 und 5.4], durchgeführt. Weil das Projekt nicht mit dem Hintergedanken hoher Performanz oder der Anwendung als Datenspeicher für sich ständig ändernde Dateien entworfen wurde, eignet es sich eher für Daten mit geringerer Fluktuation. Im Paper wird ganz direkt von der Anwendung als verteiltes CVS-Repository [5] gesprochen, da solche Codebasen meist nur geringer Fluktuation unterworfen sind.

GrassRoots Content Distribution (CAN)

In der Beschreibung von CAN wird auf eine Anwendung namens GrassRoots [42] aufmerksam gemacht. Die Literaturangabe ist der Vollständigkeit halber aus dem CAN-Paper übernommen, denn leider ist unter der angegebenen Adresse keine Dokumentation zu dieser Anwendung erhältlich. Auch eine Suche im Internet (google, citeseer) lieferte keine weiteren Ergebnisse.

A.2 Verteilte Nameserver

Nameserver im Internet-Namenssystem DNS haben die Aufgabe, Anfragen nach einem bestimmten Schlüssel (z. B. `www.uni-erlangen.de`) mit einem dazugehörigen Wert zu beantworten (z. B. `131.188.3.67`). Diese Funktionalität wurde in den Ursprüngen des Internet von einer Datei namens `hosts.txt` erfüllt, die an zentraler Stelle gepflegt wurde.

Eine verteilte Hashfunktion verrichtet ebenfalls keine andere Aufgabe, als Schlüssel-Wert Beziehungen aufzulösen. Der einzige Unterschied ist, dass das DNS-System von seiner Semantik her hierarchisch organisiert ist, eine Hashfunktion nicht. Dies stört aber zunächst nicht weiter, so dass alle Peer-to-Peer Algorithmen, die eine verteilte Hashtabelle implementieren, auch für die Bereitstellung eines verteilten DNS-Dienstes geeignet wären.

DDNS (Chord/DHash)

Eine Implementation eines verteilten DNS-Dienstes ist DDNS, das auf Chord und DHash aufsetzt. Die Autoren haben die Ergebnisse ihrer Arbeit in einem Paper [10] zusammengefaßt und ziehen folgendes Resümée: Die Bereitstellung von DNS durch einen verteilten Lokalisierungsdienst ist fehlertoleranter und erfüllt exakt die Funktionalität des oben erwähnten `hosts.txt`, also die Auflösung von Schlüssel-Wert-Zuweisungen. Leider wird die Peer-to-Peer Lösung den schnellen Antwortzeiten des herkömmlichen DNS nicht gerecht, darüberhinaus kann DDNS keine serverseitigen Berechnungen ausführen, so wie sie von Konzepten wie *Round Robin DNS* (das Zurückliefern wechselnder IP-Adressen für ein und denselben Namen zum Zwecke der Lastverteilung) gefordert werden.

A.3 Multicast-Dienste

Voraussetzung für Multicast ist eine Möglichkeit, Nachrichten in einem Netz flutwellenartig zu verbreiten, ohne dadurch exponentiellen Aufwand mit einhergehen zu lassen. Die Zellenstruktur von CAN und die Baumstrukturen von Pastry und Tapestry erscheinen hierfür besonders geeignet. Chord ist aufgrund des Drehsinns im Ring und der daraus folgenden nichtsymmetrischen Kommunikation eine weniger geeignete Wahl zur Realisierung von Multicast.

Scribe (Pastry)

Hier zitieren wir TIMO HOLM, der im Rahmen seines Hauptseminars [24] eine prägnante Zusammenfassung von Scribe erstellt hat:

Scribe ist ein verteiltes Nachrichtenpublikationssystem. Die Schlüssel werden hier sogenannten Nachrichtenkategorien zugeordnet und zum Knoten mit der am nächsten gelegenen ID als Rendezvous-Punkt für Sender und Empfänger geroutet. Jeder Knoten des Scribe Netzwerks darf beliebig viele Nachrichtenkategorien erstellen. Andere Knoten können diese Topics abonnieren und (sofern sie dazu berechtigt sind) Nachrichten in diese Kategorie posten. Scribe sorgt anschließend dafür, dass die Nachricht entlang eines Multicast-Trees mit dem Rendezvous-Punkt als Wurzel an alle Abonnenten verteilt wird.

M-CAN (CAN)

M-CAN ist der von SYLVIA RATNASAMY in ihrer Dissertation beschriebene Multicast-Client für CAN. Auf das systematische Fluten der CAN-Zonen mit Nachrichten kann hier nicht weiter eingegangen werden.

A.4 Dateiarhive

Im Gegensatz zu Dateisystemen sind Dateiarhive nur einer geringen Fluktuation unterworfen, und es findet auch meist kein unstrukturierter Zugriff statt, sondern sie werden in einem Lauf geschrieben und meist auch gelesen. Es kann vorkommen, dass nur einzelne Dateien wiederhergestellt werden sollen, auch dies ist aber zur Not im Rahmen eines kompletten Lesens des Archives machbar.

Leider hätte die Untersuchung der folgenden beiden Systeme den Rahmen der Arbeit endgültig gesprengt, deshalb sollen die Namen für den Leser nur genannt werden, ohne eine nähere Betrachtung folgen zu lassen.

PAST (Pastry)

Auch hier die Zusammenfassung aus TIMO HOLMS Hauptseminar [24]:

Past ist ein auf Pastry aufbauendes verteiltes Speichersystem. [...] Die Datei wird dabei mit einem erzeugten Schlüssel versehen und ins PAST-Netzwerk eingespeist. Dieses leitet die Nachricht an denjenigen Knoten weiter, dessen NodeID am nächsten liegt, welcher wiederum das File an k viele benachbarte Knoten weitergibt.

OceanStore (Tapestry)

OceanStore ist ein Speichersystem, das auf Tapestry aufbaut. Die Homepage² führt an, dass das System für eine große Zahl Teilnehmer weltweit zuverlässig Daten speichern soll. OceanStore sei, so die Autoren, auch gegen byzantinische Fehler abgesichert.

A.5 Kooperative Webcaches

Auch beim Thema des kooperativen Webcachings wollen wir an dieser Stelle nur darauf aufmerksam machen, dass es derartige Systeme gibt, und sie nicht im einzelnen erörtern.

Squirrel (Pastry)

Squirrel basiert auf dem vorgestellten Pastry und verfolgt, so die Autoren auf der Homepage³, die Idee, dass Benutzer in einem Netzwerk die lokalen Caches ihrer Webbrowser doch teilen könnten.

²<http://oceanstore.cs.berkeley.edu/info/overview.html>

³<http://research.microsoft.com/~antr/SQUIRREL/default.htm>

Anhang B

Ausser Konkurrenz: Pseudo Peer-to-Peer

Dieses Kapitel fasst Algorithmen zusammen, die dem interessierten Besucher von Peer-to-Peer Seiten in Stichworten öfters begegnen, und auch im Volksmund oder auf diversen Internetseiten [37] als Peer-to-Peer Systeme bezeichnet werden. Jedoch liegt allen eine erhebliche Zentralisierung zugrunde, die einen einzigen oder mehrere ausgezeichnete Server dazu benutzt, dynamisch hinzukommenden Client-Rechnern eine gewisse Aufgabe zu geben oder zentral einen Überblick über die Ressourcen der Knoten zu behalten. Zur Erledigung der Aufgabe ist keine Verbindung zum Server nötig, und erst nach Beendigung ist wieder Kommunikation zur Übertragung der Ergebnisse nötig. Die Gesamtsicht auf das System liegt auf den zentralen Servern, nicht aber auf den teilnehmenden Clients. Ebenso nehmen auch nur die Server Notiz von Join- und Leave-Operationen der Teilnehmer.

Das Novum der Systeme liegt in der Tatsache, der breiten Öffentlichkeit an Besitzern eines gewöhnlichen PCs die Teilnahme an einem großen Gesamtsystem zu ermöglichen und brachliegende Ressourcen dadurch zu nutzen. Diese Systeme sind jedoch im technischen Sinne der Arbeit nicht verwertbar, da keine echtes Peer-to-Peer vorliegt. Aufgrund ihrer Popularität sollen sie dem Leser trotzdem nicht vorenthalten werden, schon einmal zur Abgrenzung von den echten Peer-to-Peer-Anwendungen. Natürlich besteht kein Anspruch auf Vollständigkeit, vielmehr soll die Aufzählung einen Einblick vermitteln, welche Spielarten von Pseudo Peer-to-Peer anzutreffen sind.

B.1 Verteiltes Rechnen

grub.org

Diese verteilte Suchmaschine benutzt die Clients zum Durchsuchen und Indizieren von Internetseiten. Paketweise werden hierbei zu durchsuchende URL¹s übergeben, die Ergebnisse der Bearbeitung werden schlussendlich wieder an den Server übermittelt. `grub.org` [21] baut dann aus den Ergebnissen einen Web-Index zusammen, der laut der Homepage den Web-Indizes anderer bekannter Suchmaschinen (vgl. `google`, `altavista`, `fireball` etc.) in nichts nachstehen soll und aufgrund der P2P-Vorgehensweise noch eine höhere Aktualität besitzen soll.

Seti@Home

Ähnlich verfährt das aus den Medien hinreichend bekannte Seti@Home-Projekt [49]. Die im Überfluß vorhandenen Daten eines Radioteleskops sollen mit Methoden der Mustererkennung nach Indi-

¹Uniform Resource Locator, definiert in RFC 1738 [6]

zien für außerirdisches Leben durchsucht werden. Weil im Institut in Berkeley aber nicht genügend Rechenleistung zur Verfügung steht, hat man sich dafür entschieden, jeweils kleine Bruchstücke der Radiodaten an freiwillige Teilnehmer zu übermitteln und dort auswerten zu lassen. Auch hier empfangen die Clients Rechenaufträge, deren Resultate sie wieder an den Server zurückschicken.

distributed.net

Eher mathematischer Natur sind die Projekte, die auf bei `distributed.net` gelöst werden. Hierunter fallen das Brute-force-Knacken von kryptographischen Schlüsseln und eng verknüpfte Probleme wie das Faktorisieren von Zahlen, aber auch andere mathematische Probleme wie das Berechnen von sogenannten GOLOMB-Linealen. Die Art der Bearbeitung gleicht der von `grub.org` und `Seti@Home`.

B.2 Verteilte Datenspeicherung

Hierzu kann man die bekannten zentralisierten Filesharing-Systeme zählen, die sich auf zentrale Server verlassen, um eine Lokalisierung von Ressourcen durchzuführen.

Napster

Obwohl der Datentransfer bei Napster [35] von Client zu Client direkt läuft, wird der Index über alle im System verfügbaren Daten auf einem zentralen Server gehalten. Clients kommunizieren zuerst mit dem Indexserver, bevor es ihnen möglich ist, auf die Daten zuzugreifen. Wie die Geschichte zeigt, ist ein zentraler Server nicht nur verwundbar für Ausfälle und Denial-of-Service-Attacken, sondern auch für juristische Angriffe.

B.3 Clustercomputing

Bekannt wurde das Prinzip des Clustercomputing, weil es vom wirtschaftlichen Standpunkt her meistens viel günstiger, sehr viele kleine Rechner (z. B. den billigen und leicht ersetzbaren Standard-PC) zu kaufen als einen großen Supercomputer.

BEOWULF, Mosix und andere

Diese für Linux existierenden Projekte ermöglichen die Verteilung von parallelisierbaren Prozessen auf viele Computer. Das System ist deshalb kein echtes Peer-to-Peer, weil es im Cluster einen Rechner gibt, der die Jobs zentral verteilt. Eine medienwirksame Anwendung solcher Software war der CLOWN 98 [1], eine Veranstaltung in Paderborn, bei der im Jahre 1998 über 512 PCs unter anderem eine minutenlange Videosequenz verteilt berechneten. Die PCs wurden teilweise von Firmen, größtenteils aber von Privatanwendern für das Event zur Verfügung gestellt.

Anhang C

Mathematischer Hintergrund

Dieser Anhang versorgt den Leser der Arbeit mit allen für das Verständnis der Algorithmen notwendigen mathematische Grundwissen. Die behandelten Teilbereiche werden alle bei der Untersuchung der verschiedenen Systeme benötigt.

C.1 Hashfunktionen

Alle in dieser Arbeit analysierten Algorithmen sind strukturiert und realisieren damit eine verteilte Hashtabelle. Der Begriff der Hashfunktion ist demnach wichtig, und die Systeme stellen an die verwendete Funktion gewisse Ansprüche. Hier wird, angelehnt an BUCHMANN [9], eine kurze Einführung gegeben.

Definition C.1.1 (Hashfunktionen) *Unter dem Begriff Hashfunktionen versteht man Funktionen h , die einen beliebig langen String über einem Alphabet Σ auf einen String der festen Länge n über dem selben Alphabet abbilden:*

$$h : \Sigma^* \rightarrow \Sigma^n; \quad n \in \mathbb{N}$$

Es ist sofort ersichtlich, dass eine Abbildung von einem potentiell unendlich mächtigen Raum in einen begrenzten Raum nicht injektiv sein kann. Der Beobachter erwartet also zu Recht Kollisionen folgender Form:

$$\exists(x, x') \in \Sigma^* \times \Sigma^*; \quad x \neq x' : \quad h(x) = h(x')$$

Obwohl die Praxis genügend komplexe Hashfunktionen (wie z. B. SHA-1, MD5, RIPEMD160) bereithält, muss man auch nicht viel Mühe aufwenden, um selbst eine einfache Hashfunktion zu konstruieren:

Beispiel C.1.1 (Parität) *Die k -stellige Paritätsfunktion ist ein Exemplar einer Hashfunktion nach obiger Definition. Sie bildet einen Bitvektor b beliebiger Länge $|b| = k$ auf ein Bit ab:*

$$\begin{aligned} h : \Sigma^k &\rightarrow \Sigma \\ h : b_1 b_2 \dots b_k &\mapsto b_1 \oplus b_2 \oplus \dots \oplus b_k; \quad k \in \mathbb{N}; \quad b \in \{0, 1\}^* \end{aligned}$$

Allerdings läßt die Paritätsfunktion einige wichtige Eigenschaften vermissen, die sie für Kryptographie, oder aber auch nur für eine banale Streuspeicherung tauglich machen würden (Die Paritätsfunktion ist insbesondere *nicht kollisionsresistent*). Diese sollen im folgenden erläutert werden:

Einwegberechenbarkeit: Für gegebenes $y = h(x) \in \Sigma^n$ ist es *praktisch unmöglich*¹, irgendein $x' \in \Sigma^*$ (auch $x = x'$ erlaubt) zu berechnen, so dass $y = h(x')$. Mit anderen Worten: Es ist entweder kein effizienter Algorithmus zur Berechnung von Urbildern bekannt, oder es kann prinzipiell keinen geben (je nach untersuchter Funktion).

Kollisionsresistenz: Für ein *beliebiges* $x \in \Sigma^*$ ist es praktisch unmöglich, irgendeine Kollision (x, x') zu finden.

Effiziente Berechenbarkeit: Um die Funktion auch praktisch einsetzen zu können, muss sie mit annehmbarem Aufwand berechnet werden können. Diese Einschätzung ist abhängig von der derzeit verwendeten Rechnergeneration und dem Einsatzgebiet der Hashfunktion.

Gleichverteiltheit: Die Wahrscheinlichkeit, durch eine zufällig gewählte Eingabe x auf einem bestimmten Hashwert $h(x)$ zu kommen ist für jeden Hashwert $h(x)$ gleich hoch.

Es gibt noch weitere Unterschiede in den Anforderungen für bestimmte Anwendungsgebiete, z. B. Kryptographie im Vergleich zu Datenbanken. Darauf soll aber in dieser Arbeit nicht weiter eingegangen werden.

C.2 Public-Key Kryptographie

Die DHash-Schicht des Peer-to-Peer Systems Chord (Abschnitt 4.1) stellt eine spezielle Funktion zum Speichern kryptographisch signierter Blöcke bereit. Weil diese Funktion sonst eher am Rande auftaucht, wird das Thema nur angeschnitten und die wichtigsten Begriffe geklärt.

Auch als asymmetrische Kryptographie bezeichnet, nimmt die Public-Key-Kryptographie heute einen wichtigen Stellenwert ein, z. B. in der digitalen Signatur und dem Verschlüsseln von Nachrichten.

Definition C.2.1 (Kryptosystem) Ein Kryptosystem sei ein Fünf-Tupel $\mathbf{K} = (\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ mit folgenden Eigenschaften:

Klartextraum \mathcal{P}	Menge aller Klartexte
Chiffretextrraum \mathcal{C}	Menge aller verschlüsselten Texte
Schlüsselraum \mathcal{K}	Menge aller Schlüssel
Verschlüsselungsfunktionen \mathcal{E}	Familie der Funktionen, die vom Klartext- in den Chiffretextrraum abbilden
Entschlüsselungsfunktionen \mathcal{D}	Familie der Funktionen, die vom Chiffretextr- in den Klartextraum abbilden

Asymmetrische Kryptographie funktioniert nun folgendermaßen:

$$\forall e \in \mathcal{K} : \exists d \in \mathcal{K} : \forall p \in \mathcal{P} : D_d(E_e(p)) = p; \quad D \in \mathcal{D}, E \in \mathcal{E}$$

¹Mit *praktisch unmöglich* ist der in der Kryptographie oft verwendete englische Begriff *computationally infeasible* gemeint, der ausdrücken soll, dass es keinen bekannten Weg zum Urbild ausser einem exhaustiven Durchprobieren aller Kombinationen gibt. Zudem soll dieser Weg noch so rechenaufwendig sein, dass selbst für heutige Verhältnisse überdimensionierte Rechenleistung in der Größenordnung von Jahrzehnten oder Jahrhunderten beschäftigt wäre, um auf diese Weise das Urbild zu finden.

In Worten: Man verschlüsselt den Klartext p zunächst mit dem Schlüssel e unter Benutzung der Verschlüsselungsfunktion E . Anschliessend gibt es eine Entschlüsselung unter Benutzung der Funktion D und des Schlüssels d (der aber nicht notwendigerweise aus der Kenntnis von e folgen muss), so dass man aus dem Chiffretext wieder den Klartext p gewinnen kann.

C.3 Metrische Räume

Vor allem Kademia (Abschnitt 4.6) betont in besonderer Weise seine verwendete Metrik. Auch andere Systeme hängen in ihrer Arbeitsweise von der zugrundegelegten Entfernungswahrnehmung ab, auch wenn dies nicht immer explizit artikuliert wird. Die folgende Definition ist dem Skript [18] von Herrn Professor HANS GRABMÜLLER zu seiner Vorlesung „Grundlagen der Angewandten Analysis“ entnommen.

Definition C.3.1 (Metrische Räume) Gegeben sei eine nichtleere Menge X .

- Eine Abbildung $d : X \times X \rightarrow \mathbb{R}$ mit den Eigenschaften:

$$(M1) \quad d(x, y) > 0 \quad \Leftrightarrow x \neq y \quad (\text{Definitheit})$$

$$(M2) \quad d(x, y) = d(y, x) \quad \forall x, y \in X \quad (\text{Symmetrie})$$

$$(M3) \quad d(x, z) \leq d(x, y) + d(y, z) \quad \forall x, y, z \in X \quad (\text{Dreiecksungleichung})$$

heisse eine **Metrik** auf X .

- Das Paar (X, d) heisse ein **metrischer Raum**, wenn d eine Metrik auf X ist.

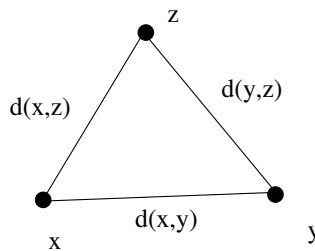


Abbildung C.3.1: Veranschaulichung der Dreiecksungleichung: Eine direkte Strecke von x nach z ist höchstens so lang wie irgendeine Verbindung über andere Punkte, hier exemplarisch über y .

Die Dreiecksungleichung wird durch Abbildung C.3.1 offensichtlich. Die Definition der EUKLIDischen Metrik ist aus einem anderen Skript von Herrn GRABMÜLLER zur Grundlagenvorlesung „Mathematik für Ingenieure I“ [19]:

Definition C.3.2 (Euklidische Metrik) Die Metrik

$$d(\vec{x}, \vec{y}) := \|\vec{x} - \vec{y}\| = \left(\sum_{k=1}^n (x_k - y_k)^2 \right)^{\frac{1}{2}} \quad \forall \vec{x}, \vec{y} \in \mathbb{R}^n \quad (\text{C.1})$$

heisst EUKLIDische Metrik (Distanz, Entfernung).

Die EUKLIDische Metrik findet durch die Verwendung von gierigem Routing in mehrdimensionalen Koordinatensystemen in CAN (Abschnitt 4.2) Verwendung.

Übertragung der EUKLIDischen Metrik auf binäre IDs

Da bei der Untersuchung der Metriken in Peer-to-Peer Algorithmen oft Binärstrings zum Einsatz kommen, muss noch geklärt werden, wie eine solche Metrik im Raum \mathbb{Z}_2^m , d. h. auf der Interpretation der Binärzahl als Vektor der Länge m aussehen würde. Hierzu vergegenwärtigen wir uns, dass eine Quadrierung der beiden zur Verfügung stehenden Ziffern keine Änderung ergibt:

$$0^2 = 0, \quad 1^2 = 1$$

Weiterhin ist das Subtrahieren zweier Bits auf einer Stelle eines Bitvektors äquivalent mit der XOR-Operation. Die Metrik lässt sich deshalb vereinfachen zu:

$$d(\vec{x}, \vec{y}) := \left(\sum_{k=1}^m x_k \oplus y_k \right)^{\frac{1}{2}} \quad \forall \vec{x}, \vec{y} \in \mathbb{Z}_2^m$$

Mit dem Wissen, dass die Anzahl der unterschiedlichen Bits in zwei Bitvektoren in der Informatik auch als *HAMMING-Abstand* bezeichnet wird [8], stellt man fest, dass die EUKLIDische Metrik in Vektorräumen auf dem Binäralphabet genau der Wurzel aus dem HAMMING-Abstand der Vektoren entspricht. Vernachlässigt man nun das Ziehen der Wurzel (was den Fixpunkt der Funktion nicht verschiebt, die Definitheit bleibt also unverletzt), so kann man HAMMING-Abstand und EUKLIDische Metrik synonym verwenden.

Definition C.3.3 (HAMMING-Abstand) Die Anzahl d_H der unterschiedlichen Komponenten zweier Bitvektoren \vec{x} und \vec{y} mit der Länge m ,

$$d_H(\vec{x}, \vec{y}) := \sum_{k=1}^m x_k \oplus y_k; \quad \forall \vec{x}, \vec{y} \in \mathbb{Z}_2^m$$

nennt man den HAMMING-Abstand der Vektoren. Auf der Menge der binären Wörter gleicher Länge stellt er eine Metrik dar.

Bemerkung: Die Dreiecksungleichung ist bei Verwendung des HAMMING-Abstandes immer exakt erfüllt, d. h.

$$d_H(x, z) \stackrel{!}{=} d_H(x, y) + d_H(y, z)$$

Dies mag ein Grund sein, wieso der HAMMING-ABSTAND als Metrik eher ungünstig für die Verwendung in Peer-to-Peer Algorithmen erscheint: Umwege im Graphen des Systemmodells würden in der Metrik keine zusätzlichen Kosten verursachen.

C.4 Grundbegriffe der Graphentheorie

Weil jedes Systemmodell eines Peer-to-Peer Algorithmus aufgrund der darunterliegenden Netzstruktur als Menge von Rechnerknoten und ihrer Verbindungen gesehen werden kann, bildet der allgemeine Graph die Oberklasse aller Systemmodelle, die es in Peer-to-Peer Algorithmen überhaupt geben kann. Aus diesem Grund sind hier die wichtigsten Definitionen der Graphentheorie

aufgeführt, die aus dem Skript zur Vorlesung „Topics in Algorithms and Complexity“ von Herrn Professor STREHL [54] entnommen sind. Zunächst wird geklärt, worum es sich beim Begriff selbst handelt:

Definition C.4.1 (allgemeiner Graph) Der allgemeine (einfache, ungerichtete) Graph $G = (V, E)$ besteht aus der Knotenmenge² V und seiner Kantenmenge³ E . V und E sind endliche Mengen, und es gilt $E \subseteq \binom{V}{2}$, die Kantenmenge ist also eine Teilmenge aller 2-Tupel von V .

Sei $n = |V|$ die Anzahl der Knoten, so ist die maximale Anzahl $m = |E|$ der Kanten beschränkt durch

$$m \leq \binom{n}{2} = \frac{n(n-1)}{2}$$

Anschließend wird auf die in der Arbeit angesprochene, für Peer-to-Peer Systeme ungünstige Vollvernetzung eingegangen:

Definition C.4.2 (vollständiger Graph) Ein komplett verbundener Graph mit n Knoten besitzt die maximal mögliche Menge $E = \binom{V}{2}$ Kanten, wenn jeder Knoten $x \in V$ paarweise mit jedem anderen Knoten $y \neq x \in V$ verbunden ist. Ein solcher vollständiger Graph wird mit der Notation K_n bezeichnet.

Wichtig für das Routing in Peer-to-Peer Anwendungen ist es, entlang der Kanten von einem Knoten zu einem anderen zu kommen. Hierbei sind die beiden Begriffe *Weg* und *Pfad* von entscheidender Bedeutung:

Definition C.4.3 (Weg im Graphen) Eine Folge W von verschiedenen Kanten

$$W = \{e_1 = \{v_1, w_1\}, e_2 = \{v_2, w_2\}, \dots, e_l = \{v_l, w_l\}\}; \quad \text{mit} \quad \begin{aligned} v_i, w_i &\in V, \\ w_i &= v_{i+1}, \\ 1 &\leq i < l, \end{aligned}$$

die von v_1 nach w_l führt, wird als *Weg* in einem Graphen bezeichnet. Die Zahl l gilt als Länge des Wegs W .

Manche Peer-to-Peer Algorithmen wie etwa Chord (Abschnitt 4.1) brauchen explizit Zyklen in ihrem Systemmodellgraph, andere wie Pastry (Abschnitt 4.4) oder Tapestry (Abschnitt 4.5) vermeiden wiederum Zyklen in ihren Graphen:

Definition C.4.4 (Zyklus (Kreis)) Ein Zyklus in einem Graphen ist ein Weg, dessen Anfangs- und Endknoten identisch sind, also $v_1 = w_l$, alle anderen Knoten aber paarweise verschieden sind.

Der Begriff des Pfads verschärft die Bedingungen des Wegs noch zusätzlich:

Definition C.4.5 (Pfad im Graphen) Ein Pfad ist ein Weg im Graphen, bei dem kein Knoten doppelt berührt wird, oder mit anderen Worten: ein zyklentreier Weg.

²von engl. *Vertex*

³von engl. *Edge*

Um festzustellen, ob durch einen Verbindungsfehler im Netzwerk aus einem Peer-to-Peer System plötzlich zwei separate Gebilde entstanden sind, ist der Begriff des *zusammenhängenden* Graphen sehr nützlich:

Definition C.4.6 (zusammenhängender Graph) *Ein Graph G ist zusammenhängend, wenn für jedes Knotenpaar $(u, v) \in V \times V, u \neq v$ ein Pfad $W = (u \sim_G v)$ im Graphen existiert, der die beiden Knoten verbindet.*

Pastry und Tapestry verwenden wie alle auf PLAXTONs Algorithmus basierenden Verfahren einen *Baum* als Systemmodell. Auch dieser muss noch definiert werden:

Definition C.4.7 (Baum und Wald) *Ein Baum ist ein zusammenhängender Graph ohne Zyklen. Ein Wald ist eine Menge nicht zusammenhängender Bäume, oder mit anderen Worten: ein Graph ohne Kreise. Insbesondere ist auch ein einzelner Baum bereits ein Wald.*

Bemerkung: Ohne Beweis stellen wir fest, dass ein Baum mit n Knoten immer $n - 1$ Kanten besitzt.

Literaturverzeichnis

- [1] *Cluster of Working Nodes*. URL <http://www.tlachmann.de/linux-cluster/html/clusterinfo/index.html>.
- [2] *IPv6: The next Generation Internet*. URL <http://www.ipv6.org>.
- [3] *YAPC 2001: Yet Another Perl Conference*. URL <http://www.yapc.org/America/previous-years/2001/>.
- [4] *Zero Configuration Networking*. URL <http://www.zeroconf.org>.
- [5] B. BERLINER. *CVS II: Parallelizing Software Development*. In *Proceedings of the USENIX Winter 1990 Technical Conference*, Seiten 341–352. USENIX Association, Berkeley, CA, 1990. URL <http://www.fnal.gov/docs/products/cvs/cvs-paper.ps>.
- [6] T. BERNERS-LEE, L. MASINTER und M. MCCAILL. *RFC 1738: Uniform Resource Locators (URL)*, Dezember 1994. URL <ftp://ftp.internic.net/rfc/rfc1738.txt>.
- [7] KENNETH P. BIRMAN. *Building secure and reliable network applications*. Manning Publications Co., Greenwich, CT, USA, 1996. ISBN 1-884777-29-5.
- [8] I. N. BRONŠTEIN, K. A. SEMENDJAEV, GERHARD MUSIOL und HEINER MÜHLIG. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Frankfurt am Main, Thun, vierte Auflage, 1999. ISBN 3-8171-2014-1.
- [9] JOHANNES BUCHMANN. *Einführung in die Kryptographie*. Springer, Berlin, erste Auflage, 1999. ISBN 3-4540-66059-3.
- [10] RUSS COX, ATHICHA MUTHITACHAROEN und ROBERT T. MORRIS. *Serving DNS using a Peer-to-Peer Lookup Service*. In *Proceedings*. First International Workshop on Peer-to-Peer Systems, Cambridge, MA, March 2002. URL <http://www.pdos.lcs.mit.edu/chord/papers/ddns.ps>.
- [11] FRANK DABEK. *A Cooperative File System*. Submitted in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering, Massachusetts Institute of Technology, September 2001. URL http://www.pdos.lcs.mit.edu/papers/chord:dabek_thesis/dabek.ps.
- [12] FRANK DABEK, EMMA BRUNSKILL, M. FRANS KAASHOEK, DAVID KARGER, ROBERT MORRIS, ION STOICA und HARI BALAKRISHNAN. *Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service*. MIT Laboratory for Computer Science. URL <http://pdos.lcs.mit.edu/chord>.
- [13] FRANK DABEK, M. FRANS KAASHOEK, DAVID KARGER, ROBERT MORRIS und ION STOICA. *Wide area cooperative storage with CFS*. In *Proceedings*. ACM SOSP 2001, Banff, Canada, October 2001. URL http://www.pdos.lcs.mit.edu/papers/cfs:sosp01/cfs_sosp.ps.
- [14] R. DROMS. *RFC 2131: Dynamic Host Configuration Protocol*, März 1997. URL <ftp://ftp.internic.net/rfc/rfc1541.txt>, <ftp://ftp.internic.net/rfc/rfc2131.txt>.

- [15] PETER DRUSCHEL und ANTHONY ROWSTRON. *PAST: A large-scale, persistent peer-to-peer storage utility*. In *Proceedings. HotOS VIII*, Schloss Elmau, Germany, May 2001. URL <http://research.microsoft.com/~antr/PAST/hotos.ps>.
- [16] PAUL FRANCIS. *Yoid: Extending the Internet Multicast Architecture*, April 2000. URL <http://www.icir.org/yoid/docs/>.
- [17] Gnutella. *Gnutella*. URL <http://www.gnutella.co.uk>.
- [18] HANS GRABMÜLLER. *Grundlagen der Angewandten Analysis*. Friedrich-Alexander-Universität Erlangen-Nürnberg, 1999. URL <http://www.am.uni-erlangen.de/~script/grabm/angmath.ps.gz>.
- [19] HANS GRABMÜLLER. *Mathematik für Ingenieure I*. Friedrich-Alexander-Universität Erlangen-Nürnberg, 1999. URL <http://www.am.uni-erlangen.de/~script/grabm/ingmath1.ps>.
- [20] ROSS LEE GRAHAM. *Peer-to-Peer: Towards a Definition*. URL <http://www.ida.liu.se/conferences/p2p/p2p2001/p2pwhatis.html>.
- [21] Grub Inc. *grub.org - an Open Source, Distributed Internet Crawler*. URL <http://www.grub.org>.
- [22] PETER GUTMANN. *Design and Verification of a Cryptographic Security Architecture*. Springer New York. ISBN 0-387-95387-6.
- [23] R. HINDEN und S. DEERING. *RFC 3513: Internet Protocol Version 6 (IPv6) Addressing Architecture*, April 2003. URL <ftp://ftp.internic.net/rfc/rfc3513.txt>.
- [24] TIMO HOLM. *Pastry: Ein skalierbarer und verteilter Location-Service*. In *Hauptseminar. Moderne Konzepte für weitverteilte Systeme: Peer-to-Peer-Netzwerke und fehlertolerante Algorithmen (DOOS)*, June 2002. URL <http://www4.informatik.uni-erlangen.de/Lehre/SS02/HS.DOOS/pdf/handout-sitiholm.pdf>.
- [25] SEPANDAR D. KAMVAR, MARIO T. SCHLOSSER und HECTOR GARCIA-MOLINA. *The EigenTrust Algorithm for Reputation Management in P2P Networks*. In *Proceedings. WWW2003*, Budapest, Ungarn, May 2003. URL <http://dbpubs.stanford.edu:8090/pub/2002-56>.
- [26] DONALD E. KNUTH. *Fundamental Algorithms*, Band 1 von *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, dritte Auflage, 10 Januar 1997. ISBN 0-201-89683-4.
- [27] H. KRAWCZYK, M. BELLARE und R. CANETTI. *HMAC: Keyed-Hashing for Message Authentication*, February 1997.
- [28] DAVID LIBEN-NOWELL, HARI BALAKRISHNAN und DAVID KARGER. *Analysis of the Evolution of Peer-to-Peer Systems*. MIT Laboratory for Computer Science. URL <http://www.pdos.lcs.mit.edu/chord/papers/podc2002.ps>.
- [29] PETAR MAYMOUNKOV und DAVID MAZIÈRES. *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*. In *Proceedings. 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, MA, March 2002. URL http://kademlia.scs.cs.nyu.edu/kpos_iptps.ps.
- [30] Microsoft Corporation. *Microsoft Research Home*. URL <http://research.microsoft.com>.
- [31] MIT. *The Chord Project*. URL <http://pdos.lcs.mit.edu/chord>.
- [32] P. V. MOCKAPETRIS. *RFC 1034: Domain names — concepts and facilities*, November 1987. URL <ftp://ftp.internic.net/rfc/rfc1034.txt>.
- [33] P. V. MOCKAPETRIS. *RFC 1035: Domain names — implementation and specification*, November 1987. URL <ftp://ftp.internic.net/rfc/rfc1035.txt>.
- [34] ATHICHA MUTHITACHAROEN, ROBERT MORRIS, THOMER M. GIL und BENJIE CHEN. *Ivy: A Read/Write Peer-to-Peer File System*. In *Proceedings. USENIX OSDI 2002*, Boston, MA, December 2002. URL <http://www.pdos.lcs.mit.edu/ivy/osdi02.ps>.

- [35] Napster. *Napster*. URL <http://www.napster.com>.
- [36] The OpenBSD Project. *Free, Functional and Secure*. URL <http://www.openbsd.org>.
- [37] O'Reilly. *OpenP2P.com - P2P development, open source development*. URL <http://www.openp2p.com>.
- [38] O'Reilly. *O'Reilly P2P Conference Coverage*. URL <http://www.openp2p.com/pub/a/p2p/conference/index.html>.
- [39] C. GREG PLAXTON, RAJMOHAN RAJARAMAN und ANDRÉA W. RICHA. *Accessing Nearby Copies of Replicated Objects in a Distributed Environment*. In *Proceedings. ACM 9th Annual Symposium on Parallelism in Algorithms and Architectures*, Newport, Rhode Island, June 1997. URL <http://www.cs.utexas.edu/users/plaxton/ps/1997/spaa.ps>.
- [40] J. POSTEL. *RFC 791: Internet Protocol*, September 1981. URL <ftp://ftp.internic.net/rfc/rfc760.txt>, <ftp://ftp.internic.net/rfc/rfc791.txt>.
- [41] B. PRENEEL und P. VAN OORSCHOT. *Building fast MACs from hash functions*. In *Advances in Cryptology – Proceedings*, Seiten 1–14. CRYPTO '95, Springer, 1995.
- [42] S. RATNASAMY, P. FRANCIS, M. HANDLEY, R. KARP, J. PADHYE und S. SHENKER. *GrassRoots Content Distribution: „RAID meets the Web“*, January 2001. URL <http://www.aciri.org/sylvia/>, unpublished, but available at the author's web site.
- [43] SYLVIA RATNASAMY. *A scalable, efficient Content-Addressable Network*. Qualifying exam proposal, University of California, Berkeley, März 2001. URL <http://www.icir.org/sylvia/>.
- [44] SYLVIA RATNASAMY. *A scalable, efficient Content-Addressable Network*. PhD thesis, University of California, Berkeley, October 2002. URL <http://www.icir.org/sylvia/>.
- [45] ANTONY ROWSTRON und PETER DRUSCHEL. *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Seiten 329–350. November 2001.
- [46] BRUCE SCHNEIER. *Applied Cryptography*. Wiley and Sons, 1996. ISBN 0-471-11709-9.
- [47] BRUCE SCHNEIER. *Secrets & Lies - Digital Security in a Networked World*. John Wiley & Sons, Inc., New York, erste Auflage, 2000. ISBN 0-471-25311-1.
- [48] MUKESH SINGHAL und NIRANJAN G. SHIVARATRI. *Advanced Concepts in Operating Systems*. McGraw-Hill, Inc., 1994. ISBN 0-07-057572-X.
- [49] Space Sciences Laboratory, University of California, Berkeley. *Seti@Home: Search for Extraterrestrial Intelligence*. URL <http://setiathome.ssl.berkeley.edu/>.
- [50] R. SRINIVASAN. *RFC 1831: RPC: Remote Procedure Call Protocol Specification Version 2*, August 1995. URL <ftp://ftp.internic.net/rfc/rfc1831.txt>.
- [51] RICHARD M. STALLMAN. *Various Licenses and Comments about Them*. Free Software Foundation. URL <http://www.gnu.org/licenses/license-list.html>.
- [52] ION STOICA, ROBERT MORRIS, DAVID KARGER, M. FRANS KAASHOEK und HARI BALAKRISHNAN. *Chord: A scalable Peer-to-Peer Lookup Service for Internet Applications*. In *Proceedings. ACM SIGCOMM 2001, San Diego, California, USA, August 2001*. URL http://www.pdos.lcs.mit.edu/papers/chord:sigcomm01/chord_sigcomm.ps.
- [53] ION STOICA, ROBERT MORRIS, DAVID LIBEN-NOWELL, DAVID KARGER, M. FRANS KAASHOEK, FRANK DABEK und HARI BALAKRISHNAN. *Chord: A scalable Peer-to-Peer Lookup Service for Internet Applications*. Laboratory of Computer Science, MIT, January 2002. URL <http://pdos.lcs.mit.edu/chord>.

- [54] VOLKER STREHL. *Topics in Algorithms and Complexity*. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2003. URL <http://www8.informatik.uni-erlangen.de/IMMD8/Lectures/TAC/Skript>.
- [55] NATHAN TORKINGTON. *Why you should care about P2P and web services*. In *Vortragsfolien. Yet another Perl Conference 19101*, 2001. URL <http://prometheus.frii.com/~gnat/yapc/2001-ws-p2p/slide008.html>.
- [56] DAVID WEINBERGER. *Is P2P anything?* Darwin Online Magazine, November 2001. URL <http://www.darwinmag.com/read/swiftkick/column.html?ArticleID=196>.
- [57] BEN Y. ZHAO, LING HUANG, JEREMY STRIBLING, SEAN C. RHEA, ANTHONY D. JOSEPH und JOHN D. KUBIATOWICZ. *Tapestry: A Resilient Global-scale Overlay for Service Deployment*. IEEE Journal on Selected Areas in Communications, 2003.
- [58] BEN Y. ZHAO, JOHN KUBIATOWICZ und ANTHONY D. JOSEPH. *Tapestry: An Infrastructure for Fault-tolerant Wide-Area Location and Routing*. University of California, Berkeley, April 2001. URL <http://www.cs.berkeley.edu/~ravenben/publications/CSD-01-1141.pdf>.