Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Hochperformante Auflösung kleiner Referenzen in verteilten Systemen

Joel Waimer

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Christian Becker
Betreuer/in:	Lukas Epple, M.Sc.
Beginn am:	11. November 2022
Beendet am:	11. Mai 2023

Kurzfassung

Bereits mit dem Aufkommen erster Filesharing-Systeme wurde die Entwicklung effektiver Verfahren zum Auffinden von mittels global eindeutiger Referenzen bezeichneter Datenobjekte in Peer-to-Peer-Systemen intensiv diskutiert und vorangetrieben, mit dem Ergebnis zahlreicher konkret ausgearbeiteter Lösungsansätze. Unbeachtet geblieben ist dabei jedoch der, für Filesharing-Systeme nicht lohnenswerte, für kleinere verteilte Datenspeichersysteme aber durchaus vorteilhafte Einsatz kleiner Referenzen, in der, aufgrund neuer Fortschritte im Bereich der Speicherdichte, damit einhergehenden dünnen Besetzung dieser kleinen Adressräume, durch welche allerdings die den Verfahren zueigenen Garantien bezüglich der benötigten Anzahl an Schritten zur Auflösung einer Referenz innerhalb des Systems stark verzerrt werden und sich die je Auflösung nötige Laufzeit vergrößert. Diese Arbeit beleuchtet zunächst die Grundlagen der mit der Auflösung von Datenreferenzen in verteilten Speichersystemen einhergehenden Problematiken, beschreibt die beiden Verfahren Chord [SMK+01] und Koorde [KK03] und misst anschließend deren Leistungsfähigkeit in dünn besetzen Adressräumen, unter der Verwendung kleiner Referenzen; mit den Messungen kann schließlich die Vermutung eines negativen Einflusses der dünnen Besetztheit des Adressraums auf die benötigte Laufzeit je Auflösung bestätigt werden. Eingegangen wurde hierbei auch auf mögliche Gegenmaßnahmen zur Verbesserung der Leistungsfähigkeit der beiden Verfahren, wobei hier die Verbindung der beiden untersuchten Verfahren mit einer Abwandlung des beim Distance-Halving-Netzwerk [NW03] eingesetzten Initialisierungsverfahrens zu einer nahezu gleichmäßigen Aufteilung des Adressraumes auf die einzelnen Knoten hier großes Potential besitzt, da so einer Entartung der Pfadlänge je Auflösung entgegengewirkt werden kann; zudem zeigten sich in den Messungen stark ungünstige Auswirkungen einer naiven, iterativen Implementierung des Chord-Verfahrens gegenüber Koorde.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1. Einleitung

Mit dem Aufkommen der Massentauglichkeit des in heutiger Zeit mittlerweile alltäglich gewordenen Internets einhergehend war die Konzeption und Entwicklung zahlreicher neuer, auf verteilten Systemen als deren Grundstruktur aufbauender Anwendungssysteme und Technologien, die in genau dieser Zeit zum ersten Mal weitläufige Verbreitung und Nutzung durch die nun neu vorhandene Masse möglicher Nutzer fand und damit gleichzeitig zu einer erstmaligen größeren Beachtung der Peer-to-Peer-Systeme allgemein, sowie zu einer Beschäftigung mit möglichen Herausforderungen in deren Entwurf führte. Hervorzuheben ist in diesem Zusammenhang die Rolle sogenannter Filesharing-Systeme (zu Deutsch etwa: Dateiaustauschsysteme), den in der Allgemeinbevölkerung in dieser Zeit weitverbreitetsten Systemen dieser Art, in welchen sich erstmalig die Möglichkeit eines Einsatzes einer Peer-to-Peer-Architektur als Grundlage eines großen, weitverbreiteten und spezifisch auf wenige Anwendungsfälle zugeschnittenen Anwendungssystems zeigte; ausgehend von der Bekanntheit dieser Systeme wurden schließlich viele der weiterführenden Überlegungen und Entwicklungen im Bereich der Peer-to-Peer-Systeme angestoßen. Als essentielle, in nahezu allen Anwendungsfällen benötigte Kernfunktionalität eines solchen Systems zu nennen ist die verteilte und gleichzeitig global im System funktionsfähige, effektive Verwaltung von verteilten Ressourcen jedweder Art, beispielsweise in Form gespeicherter Daten, ohne welche die allermeisten Peer-to-Peer-Anwendungssysteme nicht funktionsfähig wären; insbesondere für Filesharing-Systeme ist diese Tatsache offensichtlich einleuchtend. Konkret damit verbunden ist die Frage nach der eindeutigen Auffindbarkeit einer solchen Ressource innerhalb des Gesamtsystems, oder anders formuliert: die Frage, ob und wie genau ein im System enthaltener Peer/Knoten eine bestimmte, nicht lokal bei sich selbst verfügbare Ressource auf den nächstmöglichen, über die gesuchte Ressource verfügenden anderen Peer/Knoten ausfindig machen kann, um anschließend auf selbige Ressource zuzugreifen; beispielsweise eine bestimmte Datei innerhalb eines Filesharing-Systems. Dieses folgend genauer festgesetzte und auf eine bestimmtere Klasse von Anwendungsfällen beschränkte Problem wird allgemein als „Lookup-Problem“ bezeichnet und wurde bereits mit dem Aufkommen erster hochperformanter Lösungsansätze konkreter beleuchtet [BKK+03].

Bezogen auf Filesharing-Systeme ließ sich betreffs der grundlegenden Systemarchitektur bei den ersten entwickelten Ansätzen zur Lösung dieses Problems zunächst eine starke Neigung in Richtung einfacher, weitgehend strukturloser Ansätze in der Topologie des aufzubauenden Systems feststellen; prominentester Vertreter solcher, im Folgenden als unstrukturierte Ansätze oder Verfahren bezeichneter Architekturen ist Gnutella [Kir03], als Implementierung eines verteilten Dateiaustauschsystems. Um diesen unstrukturierten und meist eher mit schlechter Leistungsfähigkeit ausgestatteten, für deren Anwendungsfälle jedoch genügenden Verfahren algorithmisch fundiertere und allgemein leistungsfähigere Verfahren entgegenzusetzen, entstand daher, auf bereits vorig verfasste Arbeiten aufbauend, die Klasse der verteilten Hashtabellen (zu Englisch: distributed hash table; im Folgenden kurz DHT genannt) als eine Art Architekturstil für mit strukturell ausgefeilteren Systemtopologien arbeitende Ansätze, welcher rasche Verbreitung fand und eine Vielzahl an konkret implementierbaren Verfahren hervorbrachte. Der Umfang in der Funktionalität einer solchen

DHT entspricht dabei weitgehend dem der nicht-verteilten, gewöhnlichen Hashtabellen, wobei zum Erzeugen des ausschließlich numerisch gewählten Schlüsselwertes sogenanntes konsistentes Hashing in Form einer beliebigen, kollisionsresistenten Hashfunktion auf den Bezeichner des zu speichernden Inhaltes oder auch auf den Inhalt selbst angewandt wird, mittels welchem eine bestimmte zu speichernde Ressource dann auf dem für den Bezeichner verantwortlichen Knoten gespeichert und im Falle einer Suche durch einen anderen Knoten auf den verwaltenden Knoten aufgelöst wird. Als einzige Besonderheit einer DHT gegenüber einer normalen Hashtabelle ist ihre Verteiltheit zu nennen, die Tatsache also, dass die Hashtabelle nicht nur auf einem einzigen Knoten lokal, sondern als verteiltes Peer-to-Peer-System verwaltet und verwendet wird; die Zuordnung eines Schlüssels zu einem gespeicherten Wert ist auch hier das Hauptmerkmal dieser Speicherdatenstruktur, wodurch sich auch deren Potential zur Lösung des Lookup-Problems ergibt.

Trotz der zeitlichen Abstände besteht auch in heutiger Zeit eine nicht zu unterschätzende Relevanz der Bereitstellung und näheren Betrachtung dieser Komponente eines Peer-to-Peer-Systems, bzw. von verteiltem Speicher nutzenden Systemen im allgemeinen, da die grundlegende Anforderung einer effektiven Auflösung von Ressourcen auf deren verwaltende Knoten weiterhin und auch für solche Systeme mit nicht speicherbezogenen Anwendungsfällen besteht; wenngleich ein Bedeutungsverlust klassischer Filesharing-Systeme feststellbar ist, für welche diese Verfahren eigentlich erdacht worden sind. Bemerkenswert ist insbesondere die Tatsache, dass für die meisten dieser vergleichsweise leistungsfähig arbeitenden strukturierten Verfahren lediglich Analysen über das Laufzeitverhalten bei Unterscheidung der Netzwerkgröße, sprich der Anzahl an im System enthaltener Knoten, nicht aber in Bezug zur Variation der Länge der verwendeten numerischen Bezeichner in den DHTs und damit gleichzeitig in der Größe des verwendeten Adressraumes selbst existieren; meist wird ein Adressraum in der Größe des Raumes der durch die gewählte Hashfunktion erzeugten Hashwerte festgesetzt, welcher im Falle von SHA-1 beispielsweise alle Zahlen der Länge 160 Bit, im Falle von SHA-2 sogar eine Länge größer 200 Bit umfasst. Kleineren Adressräumen ist aufgrund dieser Beschränkung auf die Ausgabenlänge einer Hashfunktion nach bisherigem Kenntnisstand keine gesonderte Betrachtung eingeräumt worden, wiewohl auch für solche Adressräume durchaus Potential zur Verwendung in kleineren Systemen mit einem geringeren Umfang an zu speichernden Daten/Ressourcen besteht; selbst für einen 128-Bit-Adressraum wird immer nur ein geringer Teil der zur Verfügung stehenden Werte als Referenzen verwendet. Neben einer besseren Ausnutzung des zur Verfügung stehenden Adressraumes könnten sich durch die Verwendung kleiner Referenzen in Verbindung mit einem hochperformant arbeitenden strukturierten Ansatz, etwa durch die Tatsache eines in einem solchen Fall vorliegenden, nur äußerst spärlich mit Knoten besetzten Adressraumes beispielsweise Möglichkeiten zum Einsatz bestimmter, speicherintensiverer Verfahren zur Lösung des Lookup-Problems, oder auch weitergehende Optimierungsansätze der bisher bestehenden strukturierten Verfahren ergeben, was eine genauere Betrachtung kleinerer Adressräume zumindest lohnenswert erscheinen lässt, wodurch sich auch deren Potential zur Lösung des Lookup-Problems ergibt.

Ziel dieser Arbeit ist somit die Analyse bereits bestehender, hochperformanter strukturierter Verfahren zur Auflösung eines Bezeichners, bzw. einer Referenz auf den für die sich hinter jener Referenz verbergende Ressource verantwortlichen Knoten durchzuführen und diese dabei bestimmte Stärken und Schwächen hinsichtlich dieser Gegebenheit zu untersuchen. Die verwendeten Verfahren werden zu diesem Zweck beschrieben und implementiert, um anschließend die nötige Auswertung, insbesondere in Betrachtung von deren Leistungsfähigkeit in Abhängigkeit von der verwendeten Anzahl an im Netzwerk enthaltenen Knoten, durchführen zu können; weiter soll auf mögliche Anpassungen und Verbesserungen der Verfahren allgemein, wie auch speziell kleine Adressräume

betreffend eingegangen werden. Als Größe zur Ermittlung dieser Leistungsfähigkeit verwendet werden soll dabei die reale, durchschnittlich benötigte (Lauf)zeit je Auflösungsvorgang für eine im System gespeicherte Referenz, wobei die Menge an je Knoten zu speichernden Informationen über andere im System enthaltene Knoten ebenfalls miteinbezogen wird. Vergleichspunkt aller Messungen ist dabei ein laufzeitmäßig optimal ausgestaltetes, jedoch aus Gründen der Ausfallsicherheit ungeeignetes, einen zentralen Indexknoten zur Zuordnung und Auflösung der einzelnen Referenzen zu und auf den jeweils verwaltenden Knoten verwendendes Verfahren.

Während in Kapitel 2 zunächst einen allgemeinen Grundriss der übergeordneten Überlegungen zu dem dieser Arbeit zugrundeliegenden Lookup-Problem skizziert wird und verschiedene Arten an Lösungsansätzen, sowie damit verbundene Herausforderungen angerissen werden, bietet Kapitel 3 einen kurzen Überblick über die bisherigen Arbeiten zu konkret einsetzbaren Verfahren zur Lösung des Lookup-Problems und umreißt diese in kurzer Form. In Kapitel 4 werden die in dieser Arbeit verglichenen Lookup-Verfahren Chord und Koorde näher beschrieben; in Abschnitt 4.3 werden schließlich mögliche Optimierungen der Verfahren allgemein genannt. Der eigentliche Vergleich der Verfahren gegeneinander mit entsprechender Auswertung findet sich in Kapitel 5; Kapitel 6 gibt schließlich eine kurze Zusammenfassung über die erhaltenen Ergebnisse und schließt die Arbeit mit einem Ausblick auf weiterführende Möglichkeiten zur Analyse von Lookup-Verfahren in kleinen Referenzbereichen ab.

2. Motivation und Problemstellung

2.1. Grundlegung

Als eine mögliche Alternative zur „klassischen“ Client-Server-Architektur bildet die Klasse der sogenannten Peer-to-Peer-Systeme, im folgenden mit P2P-Systeme abgekürzt, eine Möglichkeit, verteilte Systeme auf Basis leichtgewichtigerer Einzelgeräte/-instanzen, im folgenden auch als Knoten bezeichnet, zu entwerfen. P2P-Systeme zeichnen sich, nach einer Beschreibung der mittlerweile eingestellten IRTF Peer-to-Peer Research Group [HP] unter anderem dadurch aus, dass jeder im System enthaltene Knoten sowohl als Client, als auch als Server agieren kann, womit eine Kommunikation zwischen zwei beliebigen Knoten jederzeit möglich ist und somit eine stärkere Dezentralisierung bestimmter Funktionen des Systems ermöglicht werden kann; somit besitzen P2P-Systeme eine höhere Stabilität, aufgrund größerer Autonomie der einzelnen Knoten. Aufgrund dieser Eigenschaften erscheint die Nutzung eines solchen Systemmodells zum Aufbau verteilter Speichersysteme äußerst vorteilhaft, da ein solches System durch den Zutritt weiterer Knoten nach Belieben beständig vergrößert, hochgradig dezentralisiert und gleichzeitig auch mittels schwächerer Infrastruktur betrieben werden kann. Insbesondere vor dem Hintergrund der ebenfalls mit der dezentralen Natur einhergehenden Anonymität der einzelnen Teilnehmer am Netzwerk erscheint die Nutzung von P2P-System zum zensurresistenten Filesharing logisch.

In Anbetracht der genannten Vorteile erscheint eine nähere Betrachtung der mit der Implementierung verteilter Speichersysteme auf Basis eines P2P-Systems also durchaus relevant und wichtig.

2.2. Das Auflösungs-/Lookup-Problem

Von grundlegender Wichtigkeit für die Implementierung eines verteilten Speichersystems ist es, die in den einzelnen, im Gesamtsystem enthaltenen Knoten gespeicherten Daten(einheiten) global auffindbar zu machen; ohne eine solche hätte das Gesamtsystem als solches keinen sonderlich großen Nutzen. Eine solche Funktion kann dabei auf unterschiedliche Arten und Weisen bereitgestellt werden, löst aber letztendlich immer nur das folgende Problem; konkret auch als Auflösungs- oder (engl.) Lookup-Problem bezeichnet; in der folgenden Formulierung *Balakrishnan et al.* [BKK+03] entnommen:

Definition 2.2.1 (Lookup-Problem)
Gegeben ein bestimmtes Datum X, finde den/einen im System enthaltenen Rechnerknoten k, der eine Kopie von X gespeichert hat.

Zwar handelt es sich dabei nur um eines von vielen zu lösenden Problemen bei der Entwicklung und Verwaltung eines verteilten Systems, allerdings gewinnt die genauere Betrachtung und Auswertung möglicher Lösungsansätze dieses Problems in einem verteilten Speichersystem, wie es dieser Arbeit

zugrundeliegt, zusätzlich an Bedeutung, da das effektive Auffinden von Speicherorten für die zu verwaltenden Daten eine der Hauptfunktionen eines solches Systems darstellt; zumal auch die meisten anderen von einem solchen System angebotenen Funktionen die Auflösungs-Funktion mitverwenden. Soll beispielsweise als Anfrage im Stil einer relationalen Datenbank die Summe einer bestimmten Menge an Werten berechnet werden, muss der Nutzer zuerst die für das Speichern der einzelnen Werte zuständigen Knoten kontaktieren und die Werte abfragen.

Von grundlegender Bedeutung für die genaue Ausführung der Lösung des Lookup-Problems ist dabei der Aufbau der verwendeten Referenzen; konkret also die Frage, ob eine Referenz für Menschen lesbare semantische Informationen über das referenzierte Datum enthält, wie beispielsweise in Form eines Dateinamens/-pfades, oder nicht, wie beispielsweise in Form einer Ganzzahl oder einer willkürlich wirkenden Zeichenkette. Im ersten Fall, auch als semantische Referenzierung bezeichnet, besteht unter anderem die Möglichkeit, den Datenstand nach bestimmten Schlüsselwörtern zu durchsuchen oder anhand der Referenz bestimmte Informationen über das referenzierte Datum zu erhalten, allerdings häufig auf Kosten der Tatsache, dass ein Auffinden eines bestimmten existierenden Datums nicht unbedingt garantiert werden kann. Den zweiten Fall stellt dann die semantikfreie Referenzierung dar, wie sie meist in DHTs eingesetzt wird. Zur Referenzierung eines Datums wird dazu in vielen Fällen ein Hashwert des Dateninhalts, eines Dateinamens oder eines anderen Bezeichners des Datums gebildet, mittels welchem das Datum dann global eindeutig bezeichnet und eine mögliche Auflösung des Bezeichners auf den verwaltenden Knoten erfolgen kann. Offensichtlich ist, dass eine Suche nach Schlüsselwörtern in einem solchen Fall nahezu unmöglich ist, da der Hashwert schon per definitionem keinerlei Rückschlüsse auf den ursprünglichen Bezeichner geben soll. Da sich mit semantikfreien Referenzen aber die Möglichkeit des Einsatzes strukturierter Lookup-Verfahren ergibt, ist ein garantiertes Auffinden eines im System enthaltenen Datums in fast allen Fällen gegeben.[MR07].

Um die in Definition 2.2.1 beschriebene allgemeine Formulierung des Lookup-Problems nun für das gegebene Systemmodell hinsichtlich der verwendeten semantikfreien Referenzierung anzupassen, wird im folgenden folgende spezielle Formulierung des Lookup-Problems verwendet, wie sie auch den meisten strukturierten Verfahren zugrundeliegt:

Definition 2.2.2 (Lookup-Problem, speziell)
Gegeben eine bestimmte n-Bit Ganzzahl X', mittels welcher ein bestimmtes Datum X referenziert wird: Finde den/einen Rechnerknoten k, bezeichnet durch eine m-Bit Ganzzahl k', welcher eine Kopie von X im Speicher hält.

Die Verwendung einer Ganzzahl anstelle einer Zeichenkette erschließt sich dabei aufgrund der Tatsache, dass semantikfreie Zeichenketten auf Digitalrechnern auch als andere Darstellungsweise einer binären Ganzzahl aufgefasst werden können.

Ein Auflösungs-/Lookup-Verfahren, welches genau dieses Lookup-Problem löst, lässt sich unter den gegebenen Bedingungen dann wie folgt als verteilt berechenbare Funktion definieren:

Definition 2.2.3 (Lookup-Verfahren, speziell)
Eine verteilt berechenbare Funktion $f : [0, 2^{n-1}] \rightarrow [0, 2^{m-1}]$ *wird als Lookup-Verfahren bezeichnet, wenn diese auf Eingabe einer zu einem bestimmten Datum X gehörenden eindeutigen Referenz in Form einer n-Bit Ganzzahl X' die eindeutige Referenz k' eines Knotens k in Form einer m-Bit Ganzzahl ausgibt, der eine Kopie von X speichert.*

Wie sich nun allerdings aus dieser recht offen gestalteten Formulierung schließen lässt, ist der Spielraum an möglichen Umsetzungen eines Lookup-Verfahrens entsprechend groß und unübersichtlich; dennoch lassen sich die entsprechenden Verfahren anhand bestimmter Merkmale bezüglich der zur Weiterleitung/zum Routing einer Anfrage verwendeten Informationen oder dem Aufbau einer bestimmten Struktur zwischen den einzelnen im System enthaltenen Peers/Knoten kategorisieren.

2.3. Routing in Peer-to-Peer-Systemen

Eng mit dem eigentlich Auffinden der einzelnen Objektreferenzen verbunden ist die Frage nach dem Aufbau der Verbindungen einzelner Knoten in dem zugrundeliegenden verteilten System, sowie deren Kenntnis übereinander: Besitzt ein Knoten beispielsweise Informationen über die Zuständigkeit bestimmter anderer Knoten für eine bestimmte Menge an Referenzen, so kann er diese Informationen zur Beschleunigung des Auflösungsvorgangs verwenden. Um also eine effiziente Auflösung einer gegebenen Datenreferenz zu ermöglichen, ist der Einsatz eines bestimmten Verfahrens zur gezielten Weiterleitung einer Auflösungs-/Lookup-Anfrage von großem Vorteil; oftmals werden zu genau diesem Zweck auf den Knoten bestimmte, die für das physisch vorhandene Netzwerk nötigen Routinginformationen erweiternde Informationen zu den im System enthaltenen Knoten gespeichert.

Die genaue Gestaltung des Routings in einem solchen Auflösungs-/Lookup-Verfahren bietet ebenfalls eine Vielzahl an Möglichkeiten, allerdings wird die Menge der bestehenden Verfahren zur besseren Übersicht meist in zwei verschiedene Kategorien aufgeteilt, wie u.a. in [LCP+05], [MR07] und auch in zahlreichen anderen Arbeiten:

So finden sich einerseits sogenannte strukturierte Verfahren, die zur Beschleunigung der Lookups aus den einzelnen Knoten ein systematisch aufgebautes, das physischen Netzwerk erweiterndes, sogenanntes Overlay-Netzwerk aufbauen, in welchem jedem Knoten ein bestimmter Bereich, im folgenden als Referenzbereich bezeichnet, aus einem vorgegebenen Adressraum (d.i. die Menge aller möglicher Referenzen im System; im folgenden als solcher bezeichnet) der Referenzen zuordnet und den jeweiligen Knoten dann mit bestimmten Knoten verbindet, die für andere Referenzbereiche zuständig sind. Zur Zuweisung des Referenzbereiches eines neuen Knotens wird dem Knoten zumeist zuerst ein bestimmter Wert aus dem gesamten Adressraum zugewiesen, anhand dessen ihm dann seine Position im Overlay-Netzwerk, sowie die ihm zugehörigen Verbindungen zu anderen Knoten zugeordnet werden können. Die Besonderheit dieser Klasse an Verfahren ist, dass aufgrund der sich hinter dem Overlay-Netzwerk verbergenden Graphenstruktur bestimmte Garantien, beispielsweise in Bezug auf die Anzahl an zu kontaktierenden Knoten je Lookup-Anfrage an das System, gegeben werden können.

In den allermeisten Fällen werden strukturierte Routing-Verfahren für die Implementierung einer DHT eingesetzt; dabei werden die semantikfreien Referenzen als n-Bit Hash des Bezeichners eines bestimmten Datums erzeugt, ebenso auch die Referenzen für die Knoten als Hash beispielsweise ihrer IP-Adresse und einiger Zusatzinformationen. Auf diese Art und Weise soll einerseits sichergestellt werden, dass keine Kollision für zwei verschiedene Datenobjekte auftritt und andererseits, dass die Menge an gespeicherten Objekten je Knoten in etwa gleich verteilt wird; je nach Verteilung der zuständigen Referenzbereiche. Der Adressraum fast aller strukturierter Lookup-Verfahren ist somit die Menge aller n-Bit-Ganzzahlen, für ein bestimmtes fest gewähltes $n \in \mathbb{N}$, das im Falle einer DHT durch die Hash-Funktion vorgegeben wird.

Dem gegenübergestellt werden andererseits sogenannte unstrukturierte Verfahren, die meist keine oder nur eine äußerst geringfügige Menge an Routinginformationen zu den im System enthaltenen Knoten besitzen und keine besondere Netzwerkstruktur in Form eines (mehr oder minder) komplexen Overlay-Netzwerkes aufbauen. Das Overlay-Netzwerk wird hier, wenn überhaupt, ad-hoc erzeugt, indem ein Knoten bei seinem Beitritt mit anderen bereits vorhandenen Knoten verbunden wird, allerdings ohne dabei eine bestimmte Struktur unter den Knoten untereinander aufzubauen. Mögliche Ansätze für unstrukturierte Routing/Lookup-Verfahren finden sich beispielsweise im (un)kontrollierten Fluten des Netzwerks mit der Anfrage, so beispielsweise im anfänglichen Gnutella-Netzwerk, der Nutzung einer bestimmten Heuristik zur Auswahl des/der nächsten gewählten Knoten(s) oder einem völlig zufällig gewählten Durchlauf des vorhandenen Netzwerkes finden (so bei Moors und Risson [MR07]). Andere Systeme, wie beispielsweise das für die illegale Verbreitung von Musik verwendete System Napster, nutzen hingegen stärkere Hierarchien mit einem oder mehreren zentralen Registerknoten; wiewohl dann berechtigterweise in Frage gestellt werden kann, ob die Definition eines eigentlich symmetrisch gearteten P2P-Systems so noch gegeben ist.

Beispiele für einzelne Verfahren beider Kategorien werden in Kapitel 3 kurz angerissen; für die weitergehende Beschäftigung mit den Verfahren sei hier auf die vorhandene Literatur verwiesen.

2.3.1. Erörterung der Verfahrenstypen

Schon allein aufgrund ihrer Strukturiertheit besteht für die strukturierten Verfahren quasi ein Zwang, innerhalb ihrer Routing-Verfahren ausschließlich semantikfreie Referenzen zu verwenden, während den unstrukturierten Verfahren, zumindest in der Theorie, beide Optionen (semantisch/semantikfrei) offen stehen. Die Praxis zeigt allerdings, dass hier in den meisten Fällen lediglich semantikfreie Referenzen eingesetzt werden; eine prominente Ausnahme davon bildet das Freenet-System [CSWH01], das allerdings im Gegensatz zu nahezu allen anderen strukturierten Verfahren keine DHT implementiert.

Interessanterweise lässt sich, wie schon von Moors und Risson [MR07] angemerkt wurde, ein Trend in Richtung eine strukturellen Unterfütterung unstrukturierter Verfahren feststellen, beispielsweise durch die Einführung von hierarchisch höhergestellten Überknoten/Superknoten, die jeweils eine bestimmte Anzahl an Blättern verwalten und Lookup-Anfragen ihrer Blätter somit nur an andere Überknoten/Superknoten weiterleiten müssen. Das Vorgehen ist insofern verständlich, als beispielsweise in der „klassischen“ Ausführung des Gnutella-Systems durch die dort eingesetzte Technik des Flutens eine große Menge an unnötiger Netzwerklast erzeugt wird, die zumindest vermindert werden sollte. Für andere Systeme, insbesondere im Falle Napsters, problematisch war die Tatsache, dass mit dem einen zentralen Registerknoten ein einzelner Ausfallpunkt (engl.: single point of failure) gegeben war, nach dessen (gerichtlich erwirkter) Abschaltung das ganze System nicht mehr funktionsfähig war.

Die daraufhin von den Autoren geäußerte Kritik an der klassischen Einteilung der Verfahren in strukturierte und unstrukturierte scheint daher berechtigt. Dennoch lässt sich, je nach Verwendung einer semantikfreien oder semantischen Referenzierung, eine gewisse Neigung zu einer stärkeren oder respektive schwächeren Strukturierung der für die Verfahren aufgebauten Overlay-Netzwerke erkennen, da die zusätzlich aufgebaute Struktur der „unstrukturierten“ Verfahren zumeist deutlich einfacher gestrickt ist, als die der „strukturierten“ und dementsprechend beispielsweise nur schwächere Garantien bezüglich der Anzahl an zu kontaktierenden Knoten für einen Lookup geben kann.

Es lohnt sich daher also durchaus noch, den Aspekt der Strukturiertheit zumindest im Hinterkopf zu behalten; allerdings besteht der weitaus größere Unterschied mittlerweile tatsächlich darin, ob semantische Referenzen verwendet werden, oder nicht und damit effektiv auch, ob das System eine Art DHT implementiert oder nicht.

In der von Chawathe et al. verfassten Arbeit zur Optimierung des Gnutella-Netzwerks [CRB+03] wurden von den Autoren zudem einige Schwachstellen strukturierter DHT-basierter Looup-Verfahren aufgezeigt: So sind einige der strukturierten Verfahren häufig anfällig für einen Zusammenbruch des Systems bei größeren unvorhergesehenen Knotenausfällen oder -abgängen, bieten, wie bereits erwähnt, keinerlei Möglichkeit zur Suche nach bestimmten Schlüsselwörtern und missachten die Tatsache, dass oft nur nach „populären“, d.h. häufig gesuchten und replizierten Dateien/Daten gesucht wird und somit die Garantie, jedes im System enthaltene Datum jederzeit finden zu können, eher unwichtig erscheint. Da File-Sharing-Systeme von allen diesen drei Punkten betroffen sind, ist vom Einsatz einer DHT für ein solches System nach Meinung der Autoren eher abzuraten. Soweit ist der Einschätzung der Autoren zuzustimmen, allerdings finden sich, wie im Falle dieser Arbeit, aber trotz aller dieser Einwände dennoch Einsatzmöglichkeiten für strukturierte Lookup-Verfahren, eben wenn die Referenzen als Schlüsselwerte einer DHT bereits in numerischer Form vorliegen und der Wechsel an dem System zugehöriger Peers nur äußerst sporadisch auftritt. Erwähnenswert ist hierbei, dass mit *Kademlia* (beschrieben von Maymounkov und Mazières [MM02]) bereits ein strukturiertes Lookup-Verfahren existiert, das insbesondere auf die Unterstützung häufiger Knotenzu- und -abgänge ausgelegt ist.

Die Auswirkungen transienter, sprich: nur für kürzere Zeit im Netzwerk befindlicher, Knoten auf andere strukturierte Overlay-Netzwerke sind meist aber gravierender und benötigen daher eine zusätzliche Behandlung; in solchen Fällen geschieht es häufig, dass Routing-Einträge einzelner Knoten auf nicht mehr existente Nachbarn verweisen und eine über einen solchen Knoten versandte Lookup-Anfrage dann ins Leere läuft. Zur Korrektur fehlerhafter Routingeinträge wurden deshalb oft zusätzliche, die Verfahren um eine Stabilisierung erweiternde Datenstrukturen und Algorithmen angegeben, die einerseits die bestehenden Routinginformationen je Knoten erweitern und auf einen Knotenausfall hin die vorhandenen Routingeinträge anpassen. Die Anpassung erfolgt dabei entweder aktiv, indem die bestehenden Informationen über die Nachbarn eines Knotens regelmäßig im Hintergrund überprüft und ggf. angepasst werden (so u.a. bei *Chord*[SMK+01]) oder passiv, indem etwa bei einem Ausfall, beim Abgang eines Knotens oder bei einem beim Empfang einer Nachricht die bestehenden Routinginformationen überprüft und ggf. angepasst werden (letzter Fall findet sich bei *Kademlia* [MM02]).

Ein weitaus grundlegenderes Problem besteht, wie beispielsweise von Zhang et al. [ZGG05] beschrieben, darin, dass die Unterschiede zwischen dem Overlay-Netzwerk eines Lookup-Verfahrens und dem darunterliegenden, physischen Netzwerk meist beträchtlich sind und daher ein Schritt/Hop im Overlay-Netzwerk häufig mehrere Schritte/Hops im darunterliegenden Netzwerk entspricht; unter Umständen tritt in einem großen System sogar der Fall auf, dass zwei im Overlay-Netzwerk benachbarte Knoten, die mittels des Internet Protocols (IP) miteinander kommunizieren, gar in unterschiedlichen autonomen Systemen (AS) gelegen sind. Solche Fälle sollten tunlichst vermieden oder zumindest in ihrer Anzahl beschränkt werden, da sich sonst die für einen Lookup benötigte Antwortzeit allein durch die entstehende Netzwerklatenz in die Länge zieht; insbesondere vor dem Hintergrund, dass ein Lookup mittels strukturierter Verfahren fast ausschließlich in $O(log(N))$ Schritten/Hops (für $N \in \mathbb{N}$ Knoten) ausgeführt wird. Auf das Latenz-Problem wurde bereits recht früh von *Plaxton et al.* [PRR97] eingegangen; sowohl Zhang et al. [ZGG05], als auch Ren et al.

[RGJZ04] beschreiben jeweils zwei unterschiedliche konkrete Lösungsansätze, um die Topologie des Overlay-Netzwerks an die der physisch gegebenen Verbindungen anzupassen und damit die Latenzen, d.h. effektiv die Antwortzeit je Lookup, zu reduzieren.

2.3.2. Metriken

Zur Bewertung eines Lookup-Verfahrens hinsichtlich seiner Effektivität bestehen mehrere mögliche Metriken, mit welchen sich bestimmte Eigenschaften eines Verfahrens messbar erfassen lassen. Besonders häufige Verwendung finden in diesem Rahmen sowohl die Menge der je Knoten zu speichernden Nachbarn im Overlay-Netzwerk, effektiv die Größe der Overlay-Routingtabelle je Knoten, sowie die Anzahl an Schritten/Hops je auszuführendem Lookup bzw. die Länge eines für einen Lookup zurückgelegten Pfades, wie beispielsweise in [SMK+01], [RD01] oder [RFH+01]; wobei zu beachten ist, dass diese Größen, insbesondere für strukturierte Verfahren, meist nur als Abschätzung des schlechtesten Falls in O-Notation angeben. Von ebenfalls nicht zu unterschätzender Relevanz zur Messung der Leistungsfähigkeit eines Lookup-Verfahrens ist die Anzahl an auszutauschenden Nachrichten beim Beitritt eines Knotens zum Netzwerk, die stark mit der Anzahl an Nachbarn je Knoten korreliert; vorwiegend für Systemen mit einer größeren Anzahl hochtransienter Knoten ist ein hoher Wert hier eher ungünstig. Auch hier wird häufig lediglich eine grobe Abschätzung der genauen Anzahl angegeben, da diese von Fall zu Fall unter Umständen stark variieren kann. Logischerweise ergibt sich zwischen diesen beiden Größen, der Anzahl an Nachbarn je Knoten und der Anzahl an nötigen Schritten/Hops je Lookup, die Möglichkeit einer Abwägung, sodass einerseits Verfahren mit einer großen Anzahl an Nachbarn und einer kleinen Anzahl an Schritten/Hops oder im umgedrehten Verhältnis bestehen.

Andere, eher praktisch geartete Ansätze finden sich beispielsweise in der bereits erwähnten Messung bestehender Latenzen zwischen einzelnen Knoten, bzw. der Messung der für einen Lookup im Durchschnitt benötigten Zeit im Allgemeinen, sowie in der Messung der durchschnittlichen Last je Lookup auf die einzelnen Knoten, in Form der empfangenen und zu sendenden Nachrichten je Knoten. Alternativ dazu können die oben beschriebenen Abschätzungen für die Anzahl an Schritten/Hops bzw. die Anzahl an Nachbarn exakt ausgemessen werden.

Innerhalb dieser Arbeit wird vor allem die für einen Lookup benötigte Zeit als hauptsächlich relevante Metrik eines effektiven Lookup-Verfahrens betrachtet, da ein intransientes, nahezu statisches System als Grundlage für die Messungen vorausgesetzt ist; die Abschätzungen für den zu haltenden Zustand je Knoten, wie auch die durchschnittliche Länge eines Lookup-Pfades werden allerdings ebenfalls beschrieben und miteinbedacht.

3. Verwandte Arbeiten

Wie der vorige Abschnitt bereits zu erkennen gab, ist das der Arbeit zugrundeliegende Lookup-Problem kein grundsätzlich neues, sondern wurde vor ungefähr zwanzig Jahren bereits intensiv diskutiert und beleuchtet; hauptsächlich im Hinblick auf mögliche Ansätze zur Entwicklung neuer oder zur Verbesserung bereits bestehender Routing-Verfahren. Darüber hinaus wurden im Nachgang zur Entwicklung und Beschreibung neuer, zumeist strukturierter Verfahren, mit den Verfahren einhergehende weitere Probleme, wie beispielsweise die Absicherung eines laufenden Systems gegen Angreifer in kleinerer oder größerer Zahl oder der mögliche, effektive Einsatz von Caching-Strategien in einem solchen System (siehe dazu u.a. *Rao et al.*[RCFB07]) behandelt.

Einen guten und gleichzeitig kompakten Überblick über die bisherige Forschungsarbeit zur Lösung des Lookup-Problems, inklusive bestehender Verfahren sowohl zur einfachen Auflösung von semantikfreien Schlüsselwerten, beispielsweise im Rahmen einer DHT, als auch zu Ansätzen zur Suche nach bestimmten, semantische Informationen beinhaltender Schlüsselwörter, beispielsweise in einem Dateiaustauschsystem, bieten Moors und Risson in dem von ihnen verfassten *RFC 4981* [MR07]. Von den Autoren werden dabei neben den einzelnen Verfahren auch die zum allgemeinen Verständnis der zugrundeliegenden Thematik notwendigen Grundlagen erklärt und auf eine durchaus umfangreiche Menge an weiterführender Literatur verwiesen. Innerhalb dieser Arbeit wird daher an mehreren Stellen auf den *RFC 4981* Bezug genommen werden.

In weitaus kleinerem Umfang werden von *Lua et al.* [LCP+05] einige bekannte Lookup-Verfahren beschrieben und miteinander verglichen, wobei auch hier auf weitergehende Problemstellungen und eine große Menge verwandter Arbeiten eingegangen wurde.

Hinsichtlich bestehender Lookup-Verfahren sei im Allgemeinen noch einmal auf die in Abschnitt 2.3 ausführlicher erörterte Unterscheidung zwischen Strukturiertheit und Unstrukturiertheit verwiesen. Das wohl prominenteste und heutzutage am häufigsten genutzte Lookup-Verfahren ist der vom DNS (Domain Name System) [Moc87] angebotene Verzeichnisdienst, welches allerdings stark hierarchisch aufgebaut ist und daher für P2P-Systeme eher ungeeignet ist. Mögliche Ansätze zur Umsetzung des DNS auf Basis einer DHT wurden jedoch bereits vorgeschlagen (so u.a. von *Cox et al.* [CMM02]).

Im bereits mehrfach erwähnten Bereich des Filesharings entstanden, wie bereits in Abschnitt 2.3 beschrieben, die meisten der heute existierenden unstrukturierten Lookup-Verfahren; prominentester Vorreiter bezüglich dieser Klasse an Verfahren war Napster, das allerdings hauptsächlich zum urheberrechtswidrigen Austausch von Musikdateien genutzt wurde und daher zunächst ein schlechtes Licht auf P2P-Systeme im Allgemeinen warf. Indes bleibt aufgrund des Einsatzes eines Zentralregisters fraglich, ob hier überhaupt von einem P2P-System gesprochen werden kann oder nicht, da für ein solches eigentlich, wie in Abschnitt 2.1 erwähnt, eine halbwegs symmetrische Rollenverteilung unter den beteiligten Peers vorausgesetzt wird; jedoch existieren auch genügend andere, zumeist unstrukturierte Verfahren, die eine ähnliche Komponente nutzen.

Weitere Nutzer unstrukturierter Lookup-Verfahren finden sich mit Gnutella [Kir03], welches rein dezentral gearbeitet und Lookups ursprünglich durch ein Fluten des Netzwerks ausgeführt hat. Mittlerweile existieren einige Optimierungen des Lookup-Verfahrens, beispielsweise von Chawathe et al. [CRB+03] oder in Form der Gnutella2 genannten Weiterentwicklung des Systems [VAI+08].

Ebenfalls ein unstrukturiertes Lookup-Verfahren nutzend, jedoch deutlich ausgeklügelter aufgebaut ist das Freenet-System [CSWH01], das im Gegensatz zur von Gnutella verwendeten Breitensuche eine Art Tiefensuche zum Lookup eines bestimmten Datums einsetzt und gleichzeitig aber, gegensätzlich zu der Mehrheit aller anderer unstrukturierter Lookup-Verfahren, ein semantikfreies Referenzierungskonzept auf Basis von Hashwerten nutzt.

Weitere Beispiele für den Einsatz unstrukturierter Lookup-Verfahren finden sich etwa im ursprünglichen BitTorrent-Protokoll [PGES05], im FastTrack-Protokoll [LKR06] oder auch in einer anfänglichen Implementierung des Kommunikationsdienstes Skype [GD05]; wiewohl für BitTorrent bereits Erweiterungen mittels DHTs/struktureller Verfahren existieren.

Gewissermaßen als eine Art Ergänzung zu oder Antwort auf die aufkommende Popularität von Peer-to-Peer-Systemen im Filesharing-Bereich wurden bereits vorhandene Ansätze zum Einsatz und zur Entwicklung strukturierter Lookup-Verfahren aufgegriffen und zu einer beträchtlichen Anzahl konkreter Verfahren weiterentwickelt. Vorarbeit zu solchen Verfahren findet sich bereits vor dem eigentlichen Aufkommen strukturierter Lookup-Verfahren [PRR97], [LNS96].

Auf Basis der Arbeit von *Plaxton et al.* [PRR97] wurden schließlich zwei Verfahren entwickelt, die sich aufgrund ihrer vergleichbaren Grundlegung ähneln: Pastry [RD01] und Tapestry [ZKJ+01]. Beide Verfahren basieren, ähnlich dem Longest-Prefix-Matching, auf dem Ansatz, die gesuchte Referenz ziffernweise (für eine Zifferngröße $2^b, b \in \mathbb{N}$) iterativ aufzulösen, indem stets der benachbarte Knoten gewählt wird, der die Referenz eine Ziffer genauer, oder sofern nicht vorhanden: mit gleicher Genauigkeit, auflöst. Nach Wahl des Parameters b ergibt sich dementsprechend auch die Anzahl der Nachbarn je Knoten, sowie die Anzahl an Schritten/Hops je Lookup-Anfrage; mittels der genauen Festsetzung von b kann also eine Gewichtung der beiden Metriken angegeben werden.

Als eine Art Archetyp aller auf einem ringförmigen Adressraum aufbauenden Verfahren kann das von *Stoica et al.* konzipierte Chord-Verfahren [SMK+01] begriffen werden; mitunter eines der bekanntesten Verfahren dieser Klasse und daher eine große Inspirationsquelle für zahlreiche weitere Verfahren. Chord nutzt zur effektiven Weiterleitung einer Lookup-Anfrage eine Menge von $\log(n)$ (für $n \in \mathbb{N}$ Bits) Zeigern (auch als „Finger“ bezeichnet), die jeweils auf nahe und weit entfernte Nachbarknoten auf dem Ring zeigen. Jeder Knoten im Chord-Ring bekommt dazu eine der Referenzen auf dem Ring zugewiesen und ist dann für alle Referenzen zwischen seinem direkten Vorgänger auf dem Ring und sich selbst verantwortlich. Es wird schließlich der, aus Sicht des aktuell die Anfrage bearbeitenden Knotens unmittelbarste Vorgänger der gesuchten Referenz mit der Weiterleitung der Nachricht beauftragt, welcher wiederum nach dem gleichen Muster verfährt, bis der eigentlich für die gesuchte Referenz zuständige Knoten gefunden ist. Für eine ausführlichere Darstellung des Chord-Verfahrens sei hier auf Abschnitt 4.1 verwiesen.

Zu Chord existieren bereits zahlreiche Weiterentwicklungen, die das bestehende Verfahren entweder erweitern oder aber schon stärker anpassen und die Grundstruktur des Overlay-Netzwerkes verändern. Der ersten Kategorie zugerechnet werden können Verfahren, wie das für mobile Netze optimierte MR-Chord-Verfahren von *Woungang et al.* [WTL+15], die Nutzung der jeweiligen Nachbarn der

Nachbarn eines Chord Knotens zur Leistungsoptimierung von *Bin et al.* [BFMJ08], sowie eine von *Wu et al.* [WLW08] vorgeschlagene Erweiterung des Chord-Protokolls durch bidirektionale Zeiger und latenzsenkende Maßnahmen, mitsamt vielen weiteren hier nicht erwähnten Arbeiten.

Als ein deutlich weitgehender vom ursprünglichen Chord-Verfahren abweichendes, jedoch ebenfalls einen Ring als Grundstruktur des Adressraums nutzendes Verfahren ist *Koorde*, entwickelt von *Kaashoek und Karger* [KK03], das eine Art Approximation eines de-Bruijn-Graphen [De 46] auf besagtem ringförmigem Adressraum darstellt und deshalb für jeden enthaltenen Knoten jeweils nur genau drei Verweise auf unmittelbare Nachbarknoten abspeichern muss, um eine Referenz ähnlich leistungsstark wie mit dem Chord-Verfahren auf den verantwortlichen Knoten auflösen zu können. Unter Nutzung erweiterter Routinginformationen in $O(\log(N))$ benötigt Koorde sogar merklich weniger Schritte zur Auflösung einer Referenz als Chord und bietet dazu noch eine gewisse Absicherung gegen größere unvorhergesehene Ausfälle mehrerer im Netzwerk enthaltener Knoten; eine genauere Beschreibung des Verfahrens ist dem Abschnitt 4.2 zu entnehmen.

Andere, an de-Bruijn-Graphen anlehnende Verfahren finden sich etwa im Distance-Halving-Netzwerk von *Naor und Wiederer* [NW03], welches zwar nicht ausdrücklich einen de-Bruijn-Graphen als Grundstruktur für das Overlay-Netzwerk nutzt, wohl aber große Ähnlichkeiten mit einem solchen besitzt, sowie im von *Fraigniaud und Gauron* [FG06] entwickelten D2B-Netzwerk. Weiterhin existiert mit Viceroy [MNR02] ein weiteres, auf eine gradminimierte Graphenstruktur aufbauendes Lookup-Verfahren, mit einer nötigen Anzahl an je Knoten zu speichernden Verweise auf unmittelbare Nachbarknoten in $O(1)$; hier jedoch auf Grundlage des sogenannten Butterfly-Netzwerks.

Ein weiteres, erwähnenswertes Overlay-Netzwerk mit zugehörigem Lookup-Verfahren ist das Content-Adressable Network, im folgenden mit CAN abgekürzt, wie es von *Ratnasamy et al.* [RFH+01] beschrieben wurde. Im Gegensatz zu den vorig genannten Verfahren findet hier aber kein eindimensionaler Raum an (numerischen) Referenzen Verwendung, sondern der Adressraum besteht aus einem d-dimensionalen Torus, von dem je ein bestimmter Bereich genau einem Knoten zugewiesen wird. Jeder Knoten muss zur Gewährleistung der vollen Funktionalität Informationen über genau $O(d)$ Nachbarn halten; ein Lookup benötigt im Schnitt $O(d(n^{\frac{1}{d}}))$ Schritten/Hops.

Wieder einen anderen Ansatz wählen *Maymounkov und Mazières* mit *Kademlia* [MM02]; hier wird statt einer geometrischen Topologie (z.B. Ring/Torus) eine Abstandsmetrik auf Basis des bitweisen XOR-Operators verwendet. Auf gewisse Art und Weise entsteht dadurch ebenfalls eine Art Ringstruktur, jedoch ohne eine eindeutig gerichtete Ordnung der Referenzen. Zur Auflösung einer Referenz besitzt jeder Knoten eine Anzahl sogenannter „Buckets“, die eine festlegbare Anzahl an Nachbarn innerhalb eines bestimmten Distanzbereiches nach der XOR-Metrik beinhalten und im Falle einer zu suchenden Referenz im Distanzbereich des „Buckets“ parallel Anfragen an eine ebenfalls festlegbare Anzahl im „Bucket“ enthaltener Knoten sendet. Darüber hinaus nutzt Kademlia eine Art passives Update der Routingeinträge eines Knotens, wodurch das Verfahren u.a. für hochdynamische Netzwerke optimiert werden soll: Wird eine Nachricht von einem bisher nicht bekannten Knoten empfangen, wird dieser, sofern noch Platz vorhanden, im zugehörigen „Bucket“ einsortiert oder als Ersatz für einen nicht mehr erreichbarer Knoten im „Bucket“ verwendet.

Auf einem gänzlich anderen Ansatz aufbauend entwickelt wurden Verfahren, denen sogenannte Skip-Graphen, eine Erweiterung sogenannter Skip-Listen, zugrunde liegen; beispielsweise *Aspnes und Shah* [AS07] oder auch *Harvey et al.* [HDJ+02] beschreiben solche Ansätze. Grundlegende Unterschiede zu den vorig vorgestellten Verfahren finden sich hier im Aufbau der Routingtabellen:

Statt eine, nach Bestimmung der Referenz eines Knotens deterministisch aufbaubare Graphenstruktur zu verwenden, nutzen diese Verfahren mehrere doppelt verkettete Listen, die in der Dichte ihrer Besetzung durch einzelne Knoten steig abnehmen und anhand dieser Besetzungsdichte in mehrere Ebenen geordnet sind; insgesamt existieren genau $\lceil \log(N) \rceil$ Listen, für $N \in \mathbb{N}$ die Anzahl der Knoten im Netzwerk. Eine Anfrage wird dann zuerst an die oberste Liste weitergeleitet und dann schrittweise an die darunterliegenden Listen weitergeleitet, bis die gesuchte Referenz aufgelöst. Als notwendige Erweiterung zur Vermeidung von Problemen bei Knotenausfällen besitzt ein Skip-Graph aber mehrere Listen je Ebene, wobei jeder Knoten in genau einer Liste jeder Ebene enthalten ist, wie dies bei Skip-Listen nicht der Fall ist; dort besteht jede Ebene dann aus mehreren Listen, die jeweils eine Teilmenge aller im Netzwerk enthaltener Knoten beinhaltet und miteinander verbindet.

Was die bisher beschriebenen strukturierten Lookup-Verfahren/Overlay-Netzwerke vereint, ist die Tatsache, dass jeweils eine Anzahl von $O(\log(N))$ Schritte/Hops für einen Lookup in einem Netzwerk bestehend aus genau N Knoten benötigt wird. Dem gegenüber steht eine weitere Menge an Verfahren, die die Auflösung einer Referenz innerhalb von nur $O(1)$ Schritte/Hops im Overlay-Netzwerk bewältigen, allerdings mit dem Nachteil einer Anzahl an zu speichernden Routing-Informationen in einer Größenordnung in $O(N)$. Beispiele solcher Verfahren finden sich etwa mit dem von *Leong et al.* beschriebenen *EpiChord*-Verfahren [LLD06], das dem Namensgeber Chord zumindest im Ansatz ähnelt, sowie mit 1h-Calot bei *Tang et al.* [TBC+05]; ein Vergleich dreier Verfahren dieser Klasse hinsichtlich deren Leistungsfähigkeit findet sich schließlich bei *Monnerat und Amorim* [MA09].

Zur besseren Übersichtlichkeit ist eine Auflistung der Eigenschaften einer Auswahl aus den genannten Verfahren in Tabelle A.1 angegeben; die Verfahren lassen sich dort hinsichtlich der Anzahl an je Auflösungsvorgang benötigten Schritten, wie auch des Speicheraufwandes an Routing-Informationen für einen einzelnen Knoten vergleichen.

Aufgrund der Tatsache, dass einem Knoten in einem strukturierten Lookup-Verfahren etwa die Rolle einer Art primitiven Routers zukommt, scheint die Verwendung solcher Verfahren für den Einsatz beim Aufbau und der Unterhaltung von Multicast-Bäumen in einem Netzwerk nicht verwunderlich; entsprechend bald wurden entsprechende Erweiterungen entwickelt und beschrieben, beispielsweise von *Zhuang et al.*[ZZJ+01] mit *Bayeux* oder mit *Scribe* von *Castro et al.*[CDKR02]. Mittels des Bezeichners einer Multicast-Gruppe wird im System, beispielsweise durch Hashing des Bezeichners oder der Verwendung zufälliger Bezeichner, ein Wurzelknoten für den entsprechenden Multicast-Baum der jeweils bezeichneten Gruppe ausgemacht, der dann den Ausgangspunkt für die Weiterleitung einer Nachricht an alle Mitglieder der Gruppe bildet. Unter Nutzung der durch die darunterliegenden Lookup-Verfahren bereitgestellten Routing-Informationen werden nun Pfade von den einzelnen Gruppenmitgliedern zum Wurzelknoten, bzw. auch in gegensätzlicher Richtung innerhalb des Overlay-Netzwerks aufgebaut, die dann zum Weiterleiten der Nachrichten als Pfade benutzt werden; die genaue Einrichtung der Pfade unterscheidet sich jedoch von Verfahren zu Verfahren.

4. Beschreibung der verwendeten Lookup-Verfahren

In Abschnitt 2.2 wurde das Lookup-Problem bereits kurz umrissen und mögliche Klassen an Verfahren zur Umsetzung einer effektiven Lösung des Problems grob skizziert. Die für die in dieser Arbeit durchgeführten Analysen verwendeten Lookup-Verfahren sollen nun im folgenden genauer beschrieben und erörtert werden, um einerseits ein grobes Verständnis derselben zu erlangen und mögliche Stärken und Schwächen der Verfahren herauszuarbeiten; insbesondere bezogen auf den Einsatz innerhalb eines kleinen Adressraums, d.h. einer Anzahl an Bits $n \leq 64$. Standardmäßig von den Verfahren verwendet werden Adressräume mit $n \geq 128$.

4.1. Chord

Von allen älteren strukturierten Lookup-Verfahren besitzt das von *Stoica et al.* entwickelte Chord-Verfahren [SMK+01] die größte Popularität; die Arbeit von *Stoica et al.* ist im Übrigen eine der meistzitierten in diesem Bereich. Auf Grundlage konsistenten Hashings bietet Chord eine durchaus effektive und gleichzeitig einfach verständliche Implementierung einer DHT, die neben einer logarithmischen Anzahl an Schritten/Hops je Lookup für große, vergleichsweise dicht besetzte Adressräume zudem eine halbwegs gleichförmige Aufteilung des Adressraumes auf die einzelnen Knoten aufweist, somit eine gute Verteilung der Netzwerklast auf die einzelnen Knoten bietet und gleichzeitig eine gute Skalierbarkeit, zumindest im Rahmen eines P2P-Systems, besitzt.

4.1.1. Grundkonzept

Chord nutzt zur Umsetzung der DHT als Adressbereich die Menge aller n-Bit-Ganzzahlen, angeordnet auf einem Ring (d.h. Berechnungen erfolgen jeweils modulo 2^n) in im Uhrzeigersinn aufsteigender Reihenfolge, „beginnend“ bei 0 und „endend“ bei $2^n - 1$; wiewohl nach $2^n - 1$ ein Umbruch auf die 0 erfolgt. Die im Netzwerk enthaltenen Knoten, wie auch die auf den Knoten gespeicherten Datenobjekte erhalten zur eindeutigen Identifizierbarkeit eine der auf dem Ring liegenden Ganzzahlen als Referenz zugewiesen, mittels derer dann sowohl der für ein bestimmtes Datenobjekt zuständige Knoten, als auch die Nachbarn eines Knotens ermittelt werden können. Zur Erzeugung der Referenzen wird die Berechnung eines n-Bit-Hashwertes der entsprechenden Bezeichner der Datenobjekte (z.B. der Dateiname) und Knoten (z.B. die IP-Adresse) empfohlen; eigentlich lassen sich die Referenzen aber auch gleichverteilt zufällig aus dem bestehenden Adressraum wählen, sofern die entsprechende Zuordnung zwischen Referenz und Referenzobjekt irgendwie sichergestellt werden kann.

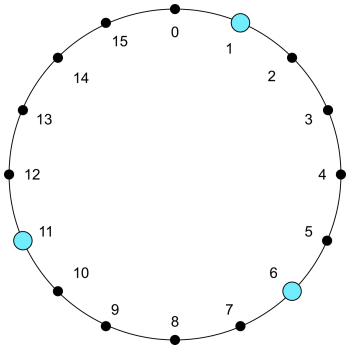


Abbildung 4.1.: Beispiel eines Chord-Rings für $n = 4$ Bit mit 3 Knoten

Um nun eine eindeutige Zuständigkeit aller Knoten für einen bestimmten Bereich des Adressraumes ohne Überschneidungen gewährleisten zu können, wird jedes Datenobjekt dem Knoten zugeordnet, dessen Referenz der Referenz des Objektes auf dem Ring unmittelbar nachfolgt oder gleicht. Demnach entspricht der Verantwortungsbereich eines Knotens dem Bereich an Referenzen zwischen der Referenz des unmittelbaren Vorgängerknotens auf dem Ring (exklusiv) und der Referenz des Knotens selbst (inklusiv). Der einer Referenz unmittelbar zuvorkommende Knoten auf dem Ring wird im folgenden als Vorgänger bezeichnet, der einer Referenz auf dem Ring unmittelbar nachfolgende Knoten entsprechend als Nachfolger; zur Auflösung einer gegebenen Referenz auf ein Datenobjekt muss jetzt also der Nachfolger dieser Referenz ermittelt werden. Im Beispiel von 4.2 ist Knoten 3 (mit Referenz 11) der Nachfolger von Knoten 2 (mit Referenz 6) und wiederum Knoten 2 der Vorgänger von Knoten 3; Knoten 3 ist somit für den Referenzbereich von $(6, 11]$, effektiv also für den Bereich $[7, 11]$ als Speicherknoten verantwortlich, da Knoten 3 für alle Referenzen in diesem Bereich der direkte Nachfolger ist.

Wie von *Stoica et al.* [SMK+01] beschrieben böte bereits eine solche Ringstruktur mit jeweils einem Zeiger je Knoten auf den jeweiligen Vorgänger bzw. Nachfolger die Möglichkeit, Referenzen auf den für die Referenz entsprechend zuständigen Knoten aufzulösen, indem der Ring über die Zeiger, beispielsweise auf den Nachfolger so lange traversiert würde, bis der zuständige Knoten gefunden wäre; allerdings mit einer durchschnittlichen Anzahl von $O(N)$ Schritten/Hops je Lookup in einem Netzwerk mit N Knoten. Dementsprechend wird diese Variante lediglich als eine Art Absicherung verwendet, für den Fall, dass ein Großteil der Nachbarn eines Knotens nicht mehr erreichbar sein sollten, wobei Chord für genau solche Fälle die Garantie geben muss/gibt, dass der für jeden Knoten gespeicherte Zeiger auf seinen Nachfolger immer korrekt ist.

In Normalfall allerdings nutzt Chord je Knoten eine Art „Routingtabelle“ mit $O(n)$ Einträgen, wobei aufgrund der mehrfachen Verwendung eines bestimmten Knotens in mehreren Einträgen eine effektive Größe dieser Tabelle in $O(\log(N))$ angegeben werden kann; $n \in \mathbb{N}$ bezeichne hierbei die Länge der ganzzahligen Referenzen auf dem Ring in Bits, $N \in \mathbb{N}$ die Anzahl im System befindlicher Knoten. Jeder Eintrag, im folgenden nach der Konvention von *Stoica et al.* als „Finger“ bezeichnet, der Routingtabelle eines Knotens verweist dabei auf einen Knoten in einem bestimmten Entfernungsbereich von diesem Knoten, wobei die Entfernung mit steigendem Index des Fingers zunimmt; konkret verweist der i-te Finger eines Knotens mit Referenz `ref` auf den Nachfolger der Referenz $\texttt{ref} + 2^{i-1} \mod 2^n$, wobei die Indizierung üblicherweise bei 1 beginnt und bis n reicht. Zu jedem mit einem Finger „getroffenen“ Knoten werden aus Gründen der Praktikabilität zusätzlich die Zeiger auf dessen Vorgänger und Nachfolger, sowie, im Falle einer gleichverteilt zufälligen

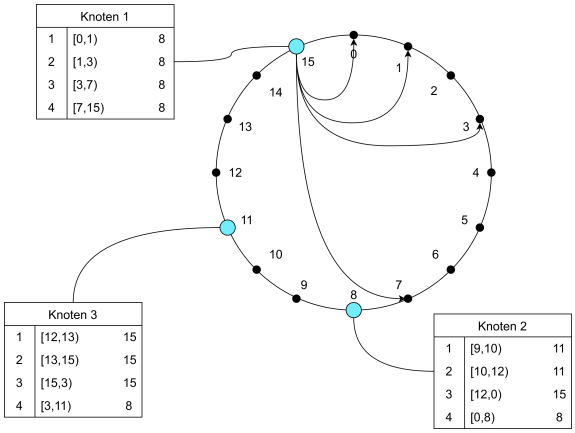


Abbildung 4.2.: Ein Chord-Ring für $n = 4$ Bit mit Routingtabellen

Wahl der Referenzen, deren Referenzen abgespeichert; eine Übersicht über den Inhalt eines Fingers findet sich in Tabelle 4.1, mit node als Bezeichner für die Referenz des den Finger speichernden Knotens.

Zum besseren Verständnis sei hier ein anschauliches Beispiel in Abbildung 4.2 gegeben; insbesondere sind hier zur besseren Verständlichkeit beispielhaft die von Knoten 1 ausgehenden Finger als Zeiger auf die jeweiligen Anfangswerte der Finger-Intervalle eingezeichnet. Betrachte den Knoten 2 mit der Referenz ref = 8: Dieser besitzt aufgrund der zugrundeliegenden 4-Bit-Referenzen eine Routingtabelle mit genau 4 Einträgen/Fingern, von denen im Schaubild sowohl die Referenz des für den Eintrag zuständigen Knotens, als auch das vom Finger abgedeckte Intervall mitangegeben ist. Da das Intervall für den ersten Eintrag mit der Referenz $\texttt{ref} + 2^{i-1} \mod 2^n = 8 + 2^0 \mod 2^4 = 9 \mod 16 = 9$ beginnt, wird der unmittelbare Nachfolger von 9 eingetragen; dieser ist Knoten 3, unter der Referenz 11. Das Intervall des zweiten Fingers beginnt mit der Referenz $8 + 2 \mod 16 = 10$, deren unmittelbarer Nachfolger ebenfalls der Knoten 3 unter der Referenz 11 ist; entsprechend ergibt sich der zweite Eintrag der Routingtabelle. Für den dritten und vierten Finger wird auf gleiche Art und Weise verfahren, wiewohl der unmittelbare Nachfolger für 12 nun nicht mehr Knoten 3, sondern Knoten 1 unter der Referenz 15 ist (da $11 < 12 \leq 15$); entsprechendes gilt dann auch für den vierten Finger, mittels welchem Knoten 6 auf sich selbst verweist, da Knoten 6 unmittelbarer Nachfolger des Startwertes dessen Intervalles 0 ist.

Wert	**Definition**
finger[i].start	$\texttt{node} + 2^{i-1}$
finger[i].interval	[finger[i].start, finger[i+1].start)
finger[i].node	Der Nachfolger der Referenz finger[i].start
finger[i].successor	Der unmittelbare Nachfolger von finger[i].node
finger[i].predecessor	Der unmittelbare Vorgänger von finger[i].node

Tabelle 4.1.: Aufbau eines Finger-Eintrages nach *Stoica et al.* [SMK+01]

Als grundlegendes Prinzip für die effektive Auflösung einer gegebenen Referenz auf einen im Netzwerk enthaltenen Knoten versucht das Chord-Verfahren, sich dem für die aufzulösende Referenz zuständigen Knoten mit jedem Schritt/Hop so weit wie möglich zu nähern; der Algorithmus folgt somit dem „Greedy"-Prinzip und findet das zu einer Referenz zugehörige Datum, sofern dieses im System vorhanden ist, in endlicher Zeit.

4.1.2. Das Auflösungsverfahren konkret

Algorithmus 4.1 findClosestPreceedingFinger für Chord, nach [SMK+01]

```
1: procedure NODE.FINDCLOSESTPRECEEDINGFINGER(ref)
2:     for i = m,...,1 do
3:         if finger[i].node ∈ (node, ref) then
4:             return finger[i]
5:         end if
6:     end for
7:     return node
8: end procedure
```

Algorithmus 4.2 findPredecessor für Chord, nach [SMK+01]

```
1: procedure NODE.FINDPREDECESSOR(ref)
2:     N ← node
3:     while ref ∉ (N.node, N.successor] do
4:         N ← N.findClosestPreceedingFinger(ref)
5:     end while
6:     return N
7: end procedure
```

Algorithmus 4.3 findSuccessor für Chord, nach [SMK+01]

```
1: procedure NODE.FINDSUCCESSOR(ref)
2:     N ← findPredecessor(ref)
3:     return N.successor
4: end procedure
```

Die Auflösung einer Referenz r erfolgt dann dinglich unter Zuhilfenahme der einzelnen „Finger" wie folgt: Zuerst wird der unmittelbare Vorgänger von r ermittelt, dessen Nachfolger der für r zuständige unmittelbare Nachfolger von r sein muss. Zu diesem Zweck prüft der die Auflösung einleitende Knoten zuerst, ob er selbst bereits der gesuchte unmittelbare Vorgänger von r ist und sucht, sofern dies nicht der Fall ist, nach dem Finger in seiner Routingtabelle, der r am nächsten zuvorkommt. Schließlich übernimmt der Knoten dieses Eintrages die weitere Suche nach dem Nachfolger von r, indem er wieder zuerst prüft, ob dieser Knoten der direkte Vorgänger von r ist und fährt dann fort, wie vom initiierenden Knoten begonnen wurde, bis schließlich der unmittelbare Vorgänger von r gefunden wurde, dessen Nachfolger der gesuchte Knoten ist. Eine Pseudocode-Implementierung der beiden zur Auflösung einer Referenz nötigen Methoden `findPredecessor` und `findSuccessor` ist in

Algorithmus 4.2 bzw. 4.3 angegeben, wobei auf die konkrete Beachtung einer bestimmten entfernten Aufrufsemantik verzichtet wurde; ebenfalls enthalten ist auch die von *Stoica et al.* verwendete Hilfsmethode `findClosestPreceedingFinger`, angegeben in Algorithmus 4.1.

Beispielhaft skizziert werden soll etwa die Auflösung der Referenz 7 durch Knoten 3 in dem in Abbildung 4.3 dargestellten, das Netzwerk aus Abbildung 4.2 erweiternden Chord-Netzwerk: Zuerst prüft Knoten 3, ob er der unmittelbare Vorgänger von Referenz 7 ist, was hier aber offensichtlich nicht der Fall ist; der unmittelbare Nachfolger von Knoten 3 ist Knoten 1 und $7 \notin [11, 15)$ offensichtlicherweise. Knoten 3 bestimmt daraufhin den nächsten, möglicherweise nicht unmittelbaren in seiner Routingtabelle enthaltenen Vorgänger der Referenz 7, welcher in diesem Fall Knoten 1 unter der Referenz 15 ist. Man beachte, dass Knoten 2 (Referenz 8) zwar das Intervall $[3, 11)$ von Finger 4 des Knotens 3 abdeckt, in welchem die 7 ja enthalten ist, Knoten 2 allerdings kein Vorgänger von 7 aus Sicht von Knoten 3 ist, da $8 > 7$. Knoten 3 kontaktiert nun Knoten 1 und lässt sich von diesem den nächsten ihm bekannten Vorgänger der gesuchten Referenz 7 ausgeben; dieser ist Knoten 1. Schließlich kontaktiert Knoten 3 Knoten 1, welcher Knoten 3 darüber informiert, dass er der gesuchte unmittelbare Vorgänger von 7 ist und dessen Nachfolger, Knoten 2 der für die Referenz 7 zuständige Knoten ist. Nach erfolgreicher Ermittlung des für die Referenz zuständigen Knotens würde nun die das Chord-Netzwerk nutzende, eine Schicht höher verortete Software den Knoten kontaktieren und die Existenz des durch die Referenz referenzierten Datums abfragen und anschließend weitere Dinge unternehmen; allerdings ist dieser Vorgang nicht Bestandteil eines Chord-Lookups.

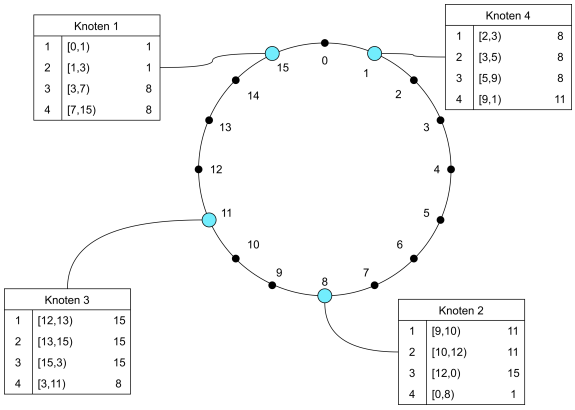


Abbildung 4.3.: Ein weiterer Chord-Ring für $n = 4$ Bit mit Routingtabellen

4.1.3. Knotenbeitritte

Naheliegend ist, dass die von den Knoten angebotenen Funktionen zur Bestimmung des unmittelbaren Vorgängers bzw. Nachfolgers einer Referenz ebenfalls dazu genutzt werden kann, die Routingeinträge eines dem bereits bestehenden Chord-Netzwerk beitretenden Knotens zu initialisieren; entsprechendes wurde von den Entwicklern des Chord-Verfahrens ebenfalls vorgeschlagen. Zur Initialisierung der eigenen Routingeinträge kontaktiert ein beitretender Knoten also zunächst einen bereits im System enthaltenen Knoten, im folgenden als Hilfsknoten bezeichnet, mit dessen Hilfe er dann zuerst seinen unmittelbaren Nachfolger bestimmt und diesen anschließend kontaktiert, damit dieser einerseits dem nun neu beigetretenen Knoten Informationen über seinen ehemaligen

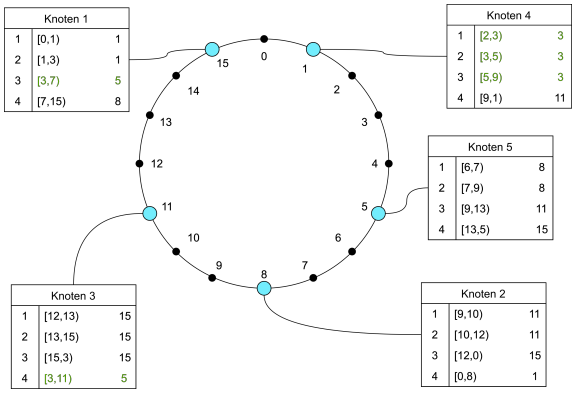


Abbildung 4.4.: Der Chord-Ring aus Abbildung 4.3 nach dem Einfügen des Knotens 5

unmittelbaren Vorgänger übermittelt, welcher der unmittelbare Vorgänger des neu beigetretenen Knotens ist und andererseits dieser den neu beigetretenen Knoten als seinen neuen unmittelbaren Vorgänger einträgt. Anschließend werden mithilfe des Hilfsknotens die Finger des neuen Knotens initialisiert, indem der Hilfsknoten die jeweiligen unmittelbaren Nachfolger der Anfangswerte jedes Finger-Intervalls bestimmt und an den neuen Knoten übermittelt. Sind alle Einträge vorhanden, signalisiert der neue Knoten sein Dasein allen Knoten im Netzwerk, deren Finger-/Routingtabellen durch den Beitritt des neuen Knotens möglicherweise angepasst werden müssen. Effektiv handelt es sich dabei um alle möglichen Vorgänger der Referenz $\texttt{ref} - 2^{i-1} \mod 2^n$, jeweils für jeden Finger mit $1 \leq i \leq n$, da ab hier der Finger i überhaupt erst auf den neuen Knoten verweisen könnte, sowie dem Knoten, dessen eigene Referenz dem Wert $\texttt{ref} - 2^{i-1} \mod 2^n$ gleicht; letztere Kategorie bildet insbesondere in kleineren Adressräumen einen zu beachtenden Randfall, der, sofern unbeachtet, zu fehlerhaften Routingeinträgen führen kann. Der vom neu beigetretenen Knoten ermittelte zu benachrichtigende Knoten für Eintrag i prüft schließlich, ob der genannte Eintrag anzupassen ist, passt diesen, wenn nötig, an und benachrichtigt seinen unmittelbaren Vorgänger über den anzupassenden Eintrag, mit welchem dieser in gleicher Weise verfährt, wie sein Benachrichtiger. So wird die Nachricht schließlich so lange im Chord-Ring weitergereicht, bis alle anzupassenden Einträge angepasst worden sind.

Auch in diesem Fall bietet sich die Verwendung eines Beispiels zum besseren Verständlichmachen des hier beschriebenen Vorganges an: Betrachtet werden soll dazu die Abbildung 4.4, die eine Erweiterung des Chord-Rings aus Abbildung 4.3 um einen fünften Knoten an der Position 5 im Ring darstellt; die in den bereits vor dem Einfügen im Ring enthaltenen Knoten geänderten Einträge sind dabei zur besseren Übersichtlichkeit blassgrün gefärbt. Bei seinem Beitritt erzeugt Knoten 5 zunächst die ihm auf dem Chord-Ring zugewiesene Referenz, diesenfalls die 5, kontaktiert anschließend einen der bereits im Chord-Netzwerk/-Ring enthaltenen Knoten, beispielsweise Knoten 3, welcher nun den unmittelbaren Nachfolger der Referenz 5 bestimmt; in diesem Falle wäre das Knoten 2 unter der Referenz 8. Knoten 5 kontaktiert schließlich Knoten 2 und ermittelt so seinen unmittelbaren Vorgänger, Knoten 4 unter der Referenz 1, im Falle des genannten Beispiels, und benachrichtigt damit gleichzeitig seinen unmittelbaren Nachfolger (Knoten 2) über seinen Beitritt zum Chord-Netzwerk als dessen neuer unmittelbarer Vorgänger. Knoten 2 trägt daraufhin Knoten 5 als seinen neuen unmittelbaren Vorgänger ein (im Bild nicht dargestellt). Ist Knoten 5

an der richtigen Stelle im Ring eingefügt, lässt Knoten 5 nun die unmittelbaren Nachfolger der Anfangsreferenzen seiner Finger ermitteln und trägt die entsprechenden Knoten dann in seine Routingtabelle ein; das Ergebnis ist in Abbildung 4.4 ebenfalls einsehbar.

Schließlich bestimmt der nun initialisierte Knoten 5 diejenigen Knoten, deren Routingtabelle aufgrund seines Beitritts angepasst werden müssen, indem, wie obig beschrieben, der unmittelbare Vorgänger der Referenz $\texttt{ref} - 2^{i-1}$ für jeden Finger $1 \leq i \leq n$ bestimmt und benachrichtigt wird. Beispielsweise für Finger 1 ist somit der unmittelbare Vorgänger der Referenz $5 - 2^0 = 5 - 1 = 4$ zu bestimmen; dieser ist Knoten 4. Knoten 4 wird deshalb kontaktiert und prüft dann, ob eine Aktualisierung seines ersten Fingers nötig ist, erkennt die Notwendigkeit, da Knoten 5 tatsächlich der neue unmittelbare Nachfolger des Anfangswertes 2 seines ersten Fingerintervalles ist und ändert den Eintrag entsprechend. Anschließend leitet er die Nachricht an seinen unmittelbaren Vorgänger, hier Knoten 1 weiter, welcher wiederum prüft, ob sein erster Finger aktualisiert werden muss, was hier jedoch nicht der Fall ist. Derweil bestimmt Knoten 5 schon den im zweiten Finger betroffenen Knoten, d.h. den unmittelbaren Vorgänger der Referenz $5 - 2^1 = 5 - 2 = 3$ (bzw. einen auf der Referenz 3 liegenden Knoten, sofern vorhanden), was wiederum Knoten 4 ist und verfährt, wie schon für den ersten Finger beschrieben; ebenso die kontaktierten Knoten 4 und 1. Erst beim dritten Finger geschieht eine „echte“ Weitergabe der neuen Informationen: Knoten 5 ermittelt den unmittelbaren Vorgänger der Referenz $5 - 2^2 = 5 - 4 = 1$, wobei hier der oben erwähnte Randfall auftritt, da ein Knoten mit der Referenz 1 im Chord-Ring enthalten ist. Somit wird dann nicht der unmittelbare Vorgänger der Referenz 1, sondern der die Referenz selbst „besetzende“ Knoten, diesenfalls Knoten 4, kontaktiert, welcher daraufhin seinen dritten Finger auf eine notwendige Anpassung überprüft, diese, da deren Notwendigkeit besteht, durchführt und die Nachricht anschließend an seinen Vorgänger, Knoten 1, weiterleitet. Knoten 1 prüft nun ebenfalls, ob eine Anpassung seines dritten Fingers aufgrund des Beitritts von Knoten 5 nötig geworden ist und führt die in der Tat notwendige Anpassung durch; anschließend leitet er die Nachricht wiederum an seinen unmittelbaren Vorgänger, Knoten 3 weiter. Auch dieser prüft wiederum seinen dritten Finger auf die Notwendigkeit einer Anpassung, die in diesem Falle aber nicht nötig ist und verwirft die Nachricht. Schließlich bestimmt Knoten 5 noch den ersten ggf. im vierten Finger betroffenen Knoten, den unmittelbaren Vorgänger der Referenz $5 - 2^3 \mod 16 = 5 - 8 \mod 16 = 13$, effektiv also Knoten 3 und benachrichtigt ihn entsprechend. Dieser wiederum passt seinen vierten Finger aufgrund des Beitritts von Knoten 5 an und leitet die Nachricht an seinen unmittelbaren Vorgänger, Knoten 8, weiter, der jedoch

Algorithmus 4.4 joinNetwork für Chord, nach [SMK+01]

```
 1: procedure NODE.JOINNETWORK(helper)
 2:     node.node ← initReference()
 3:     if helper == null then
 4:         for i = 1,...,n do
 5:             finger[i] ← node
 6:         end for
 7:         predecessor ← node
 8:     else
 9:         initFingertable(helper)
10:         updateOthers()
11:     end if
12: end procedure
```

keine Anpassung vornehmen muss. Auf diese Art und Weise wurde der Knoten 5 nun in den bereits bestehenden Chord-Ring eingefügt, wobei hier nun einfach nachzuprüfen ist, dass die so entstandenen Einträge korrekt sind.

Eine algorithmische Darstellung des Beitrittsvorgangs eines Knotens ist in 4.4 beschrieben und in angepasster Form *Stoica et al.* [SMK+01] entnommen; die ergänzenden Algorithmen sind ebenfalls in Anhang B.1 beschrieben. Infolge des Beitritts eines neuen Knotens müssen zuletzt seitens der die gespeicherten Daten verwaltenden Anwendung alle auf dem Nachfolger des neu beigetretenen Knotens gespeicherten Daten, für welche der neu beigetretene Knoten fortan zuständig ist, an jenen übertragen werden, um weiterhin eine korrekte Auflösung einer der davon betroffenen Referenzen zu ermöglichen. Das Chord-Programm sendet dazu beispielsweise eine Nachricht an die die Daten verwaltende Anwendung, welche sich dann um die Übertragung aller betroffenen Daten an den neuen Besitzer kümmert; jedoch ohne eine direkte Beteiligung des Chord-Programmes.

4.1.4. Knotenaustritte

Auch wenn von den Autoren nicht ausdrücklich beschrieben worden, verhält sich der kontrollierte Austritt eines Knotens aus einem bestehenden Chord-Netzwerk in etwa gespiegelt zu dessen Beitritt und soll in einer von vielen möglichen Ausführungen an dieser Stelle nur kurz umrissen werden. Zunächst aktualisiert der betreffende Chord-Knoten sowohl den Eintrag des unmittelbaren Vorgängers seines unmittelbaren Nachfolgers mit dem seines unmittelbaren Vorgängers und den Eintrag des unmittelbaren Nachfolgers seines unmittelbaren Vorgängers mit seinem unmittelbaren Nachfolger; kurz: Der austretende Knoten stellt den Zustand des Ringes her, der nach seinem Austritt bestehen muss. Anschließend signalisiert der austretende Knoten allen möglichen, ihn selbst als Nachbarknoten in einem Finger gespeichert haltenden Knoten seinen Austritt und die notwendige Anpassung des Fingers auf den unmittelbaren Nachfolger des austretenden Knotens. Ist der Knoten von keinem Knoten im Ring mehr in irgendeiner Art und Weise als Nachbar eingespeichert, werden schließlich alle von diesem ausgetretenen Knoten gespeicherten Daten an den, nun ehemaligen, unmittelbaren Nachfolger übermittelt, der nun für die damit verbundenen Referenzen zuständig ist; anschließend kann der Knoten seine Ausführung beenden.

Demgegenüber existiert eine zahllose Menge an möglichen unkontrollierten Austrittsvorgängen, alle jedoch mit dem Ergebnis eines dann teilweise oder gänzlich funktionsunfähigen Chord-Netzwerks.

4.1.5. Erste Anpassungen des Chord-Verfahrens

Wie sich nun feststellen lässt, ist das eben beschriebene Verfahren zum Beitritt eines neuen Knotens in ein bereits bestehendes Chord-Netzwerk zwar grundsätzlich zum Erreichen dieses Zieles geeignet, jedoch nur dann auf effiziente Art und Weise, wenn lediglich eine geringe Anzahl neuer Knoten zu einem bestimmten Zeitpunkt dem Netzwerk beitritt; ein gleichzeitiger Beitritt vieler Knoten hingegen erzeugt eine vergleichsweise große Netzwerklast, die zu zahlreichen Verzögerungen führt, begründet darin, dass bei jedem Knotenbeitritt eine in $O(\log(n))$ liegende Anzahl an Nachrichten zu versenden ist. Darüber hinaus könnten sich ebenfalls Probleme mit während dem Beitritt ausgeführten Auflösungsanfragen ergeben, die dann gegebenenfalls länger im Netzwerk unterwegs sind oder in manchen Fällen sogar gar nicht beantwortet werden können. Um dann ebenfalls eine große Anzahl an Beitritten innerhalb kürzerer Zeit zu ermöglichen, wurde von den Autoren bereits

ein hierfür besser geeignetes Beitrittsverfahren beschrieben, dass auf eine sofortige Einrichtung aller Finger im beitretenden Knoten verzichtet und dort lediglich den unmittelbaren Nachfolger des Knotens abspeichert, während die restlichen Einträge im Laufe der Zeit durch regelmäßige Abfragen gesetzt werden. Dadurch wird zwar die Anzahl der je Beitritt zu sendenden Nachrichten drastisch reduziert und somit weiterhin eine Funktionalität der Lookup-Funktionalität garantiert, jedoch zum Preis einer, zumindest für einen bestimmten Zeitraum vorherrschenden größeren Anzahl an benötigten Schritten je Anfrage.

Hierfür gestaltet sich auch die algorithmische Ausgestaltung ein wenig einfacher: Im Falle eines Beitritts ermittelt der beitretende Knoten durch den gewählten Hilfsknoten lediglich seinen unmittelbaren Nachfolger auf dem Chord-Ring und belässt seinen unmittelbaren Vorgänger leer. Alle weiterzuleitenden Auflösungsanfragen werden ausschließlich an den unmittelbaren Nachfolger weitergeleitet, bis alle Finger korrekt eingetragen sind; dinglich also ein Verhalten gleich dem, wenn alle Finger auf den unmittelbaren Nachfolger zeigen würden. Mittels wiederkehrenden Anfragen an den unmittelbaren Nachfolger über dessen unmittelbaren Vorgänger wird jeweils abgeprüft, ob der dem Knoten bekannte unmittelbare Vorgänger weiterhin auf den richtigen Knoten verweist, oder ob ein anderer Knoten dem Netzwerk beigetreten ist, welcher diesen Posten nun einnimmt. In diesem Zuge benachrichtigt der Knoten seinen unmittelbaren Nachfolger gleichzeitig über seine Existenz als dessen unmittelbarer Vorgänger, für den Fall einer hier nötigen Anpassung des Verweises. Schließlich wählt jeder Knoten, ebenfalls in gewissem Zeitabstand wiederkehrend, einen seiner Finger zur Überprüfung auf Aktualität aus und passt den Finger an, sofern nötig. Eine kleine, jedoch nicht unbedingt notwendige Verbesserung ergäbe sich darüber hinaus durch die Benachrichtigung des unmittelbaren Vorgängers des beitretenden Knotens über dessen Beitritt als dessen unmittelbarer Nachfolger; hierdurch würde auch in diesem Fall eine durchgängige Auffindbarkeit der vom beigetretenen Knoten verwalteten Daten gewährleistet.

Auch für die hier beschriebenen Verfahren ist eine algorithmische Beschreibung zum besseren Verständnis in den Algorithmen 4.5, 4.6 und 4.7, sowie in 4.8 angegeben; auf ein Beispiel wurde hier aufgrund der Ähnlichkeit zum Standardverfahren allerdings verzichtet.

Algorithmus 4.5 Angepasster Join-Algorithmus für Chord, nach [SMK+01]

```
 1: procedure NODE.JOINNETWORK(helper)
 2:     node ← initReference()
 3:     if helper != null then
 4:         successor ← helper.findSuccessor(node.node)
 5:         predecessor = nil
 6:     else
 7:         successor ← node
 8:         predecessor ← node
 9:     end if
10: end procedure
```

Eine weitere, recht einfach gehaltene Verbesserung des Beitrittsvorgangs besteht darin, den beigetretenen Knoten zunächst die gespeicherten Nachbarn beispielsweise seines unmittelbaren Nachfolgers oder eines anderen auf dem Ring in dessen Nähe liegenden Knotens übernimmt, da diese seinen eigenen Einträgen möglicherweise ähneln und diese dann, beispielsweise unter Zuhilfenahme der beschriebenen Stabilisierungsalgorithmen, mit der Zeit berichtigt.

Algorithmus 4.6 Stabilisierungsalgorithmus für Chord, nach [SMK+01]

```
1: procedure NODE.STABILIZE
2:     cand ← successor.predecessor
3:     if cand.node ∈ (node, successor.node) then
4:         successor ← cand
5:     end if
6:     successor.notify(node)
7: end procedure
```

Algorithmus 4.7 Benachrichtigung des Nachfolgers, nach [SMK+01]

```
1: procedure NODE.NOTIFY(other)
2:     if (predecessor == nil) ∨ (other.node ∈ (predecessor, node)) then
3:         predecessor ← other
4:     end if
5: end procedure
```

Unbeachtet geblieben ist bisher zudem die Behandlung von Knotenausfällen, d.h. von unvorhergesehenen Knotenabgängen ohne Benachrichtigung der im restlichen Netzwerk verbliebenen Knoten; feststellbar sind solche Ausfälle zumindest durch das Ausbleiben einer Antwort des besagten Knotens. Die Wahrscheinlichkeit eines solchen Ausfalls ist, je nach Art des verwendeten Systems, mehr oder minder relevant, bedarf aber dennoch einer angemessenen Betrachtung, da, sofern der Fall unbehandelt bleibt, die Möglichkeit einer Partitionierung des Systems zumindest in besonders ungünstigen Fällen denkbar ist; in anderen kann zumindest die Auflösung bestimmter Referenzen nicht mehr eindeutig garantiert werden. Von den Autoren wurde hierzu die Aufrechterhaltung einer Liste der nächsten $O(\log(N))$ unmittelbaren Nachfolgern eines Knotens vorgeschlagen, welche dem Knoten im Falle eines Ausfalls des ersten unmittelbaren Nachfolgers immerhin alternative Möglichkeiten zur Weiterleitung einer Anfrage bieten und somit die Anzahl an benötigten Schritten je Auflösung in $O(\log(N))$ mit hoher Wahrscheinlichkeit erhalten. Angeführt wird zudem, dass eine Größe der Liste in $O(\log(N))$ selbst bei einer Ausfallwahrscheinlichkeit von $\frac{1}{2}$ ausreicht, um das Netzwerk stabil zu halten. Bestimmten lässt sich der Faktor $O(\log(N))$ beispielsweise über das Verhältnis des Abstandes eines Knotens und seinem Nachfolger zum gesamten Adressraum.

4.1.6. Zusammenfassung

Zusammenfassend besitzt das grundlegende Chord-Verfahren folgende Eigenschaften:
Eine Auflösung einer gegebenen Referenz wird mit einem benötigten Aufwand an Schritten/Hops in $O(\log(N))$, sowie der Kenntnis einer Anzahl an Nachbarn im Chord-Netzwerk in $O(\log(N))$, mit hoher Wahrscheinlichkeit geleistet; $N \in \mathbb{N}$ beschreibt die Anzahl der im System enthaltenen Knoten. Insbesondere die Menge der je Anfrage zu kontaktierenden Knoten, d.h. die Anzahl an Schritten/Hops, ergibt sich aus der Tatsache, dass die Entfernung zwischen dem die Anfrage ausführenden Knoten und dem eigentlichen unmittelbaren Vorgänger der gesuchten Referenz mit jedem Schritt, d.h. mit jedem aufgerufenen Knoten, mindestens halbiert wird, wodurch sich für eine anfängliche Entfernung der gesuchten Referenz zum unmittelbaren Vorgänger derselben kleiner 2^s für $s < n; s \in \mathbb{N}$ eine Anzahl von maximal m Schritten bis zum Auffinden des Vorgängers ergibt. Da der Ring jedoch lediglich an N Stellen besetzt ist, was eine Zuständigkeit für $\frac{2^s}{N}$ Referenzen je

Algorithmus 4.8 Periodisches Update der Finger, nach [SMK+01]

```
1: procedure NODE.FIXFINGERS(other)
2:     i ← random index ∈ 1, ...n
3:     finger[i].node ← node.findSuccessor(finger[i].start)
4: end procedure
```

Knoten unter Annahme einer einwandfreien Gleichverteilung der Knoten auf dem Ring bedeutet, genügt eine Verminderung der verbleibenden Entfernung zwischen dem Ausgangsknoten und dem derzeitig betrachteten Knoten auf höchstens $\frac{2^s}{N}$, sprich eine Anzahl in $O(\log(N))$ Schritten zur Auflösung einer Referenz; insbesondere da $\frac{2^s}{N} \leq \frac{2^n}{M}$. Genauere Ausführungen zum Beweis dieser Tatsache sind hierbei *Stoic et al.*[SMK+01] zu entnehmen. Für den Beitritt eines weiteren Knotens zum bestehenden Chord-Netzwerk wurde von den Autoren ein Aufwand an auszutauschenden Nachrichten in $O(\log(N)^2)$ mit großer Wahrscheinlichkeit veranschlagt, welcher sich aus der Tatsache begründet, dass einerseits bereits zur Bestimmung der Finger des beigetretenen Knotens etwa $O(\log(N))$ Nachrichten auszutauschen sind und zur Aktualisierung der Finger aller durch den Beitritt des Knotens betroffenen Knoten eine Menge von $O(\log(N))$ Knoten kontaktiert werden muss, die die Nachrichten wiederum etwa $O(\log(N))$ mal weiterreichen.

4.2. Koorde

Aufbauend auf dem von *Chord* verwendeten Ansatz einer ringförmigen Anordnung des Adressraumes wurde von *Kaashoek und Karger* [KK03] eine gradminimierte Alternative für den Aufbau eines Overlay-Netzwerkes auf Basis sogenannter de-Bruijn-Graphen entwickelt, wobei ein Koorde-Netzwerk dabei in der Lage ist, mittels einer Anzahl von $O(1)$ Nachbarn je Knoten eine Referenz in einer ähnlichen Anzahl an Schritten/Hops wie Chord aufzulösen. Koorde stellt somit eine andere Art an Erweiterung bzw. Optimierung des Chord-Verfahrens, nicht hinsichtlich einzelner Aspekte des Verfahrens, sondern hinsichtlich der Grundidee hinter dem Verfahren selbst dar.

4.2.1. De-Bruijn-Graphen

Bereits erwähnt worden sind die nach dem niederländischen Mathematiker Nicolaas Govert de Bruijn benannten de-Bruijn-Graphen als Anlehnung für die Grundstruktur des aufzubauenden Overlay-Netzwerks, dessen Aufbau zunächst recht einfach beschrieben werden kann: Ein de-Bruijn-Graph ist definiert über eine Menge an Symbolen Σ, ähnlich der theoretischen Informatik auch als eine Art Alphabet zu sehen, mit Kardinalität $|\Sigma| = k$; wobei die Knoten eines solchen Graphen durch alle Wörter einer bestimmten Länge $l \in \mathbb{N}$ über Σ, sprich: durch alle Elemente von Σ^l, dargestellt werden. Ein jeder in einem solchen de-Bruijn-Graphen enthaltener Knoten, dargestellt durch $v \in \Sigma^k$, besitzt dabei Kanten zu solchen Knoten, deren Wort jeweils dem um eine Stelle nach links verschobenen Wort v mit einem beliebigen, an der durch die Verschiebung frei gewordenen letzten Stelle eingefügten Symbol aus Σ entspricht, wobei das erste Symbol in v durch die Verschiebung verschwindet; jeder Knoten besitzt daher sowohl genau k ausgehende, als auch k eingehende Kanten [De 46; Pro11]. Verwendet man nun $\Sigma = \{0, 1\}$ mit $l \in \mathbb{N}$ beliebig gewählt, so bilden der Menge der Knoten des dabei resultierenden de-Bruijn-Graphen jeweils genau die

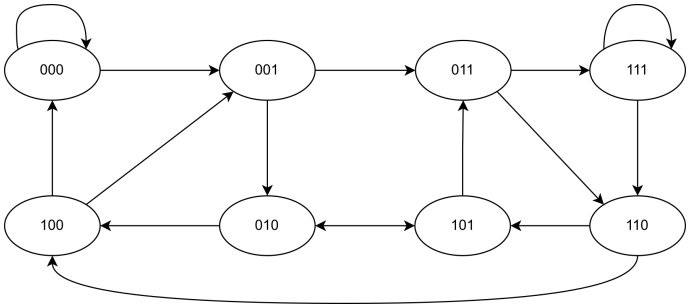


Abbildung 4.5.: Der de-Bruijn-Graph mit 2 Symbolen und Wortlänge 3

Menge aller l-Bit-Ganzzahlen ab, mit der Eigenschaft, dass jeder Knoten $v \in \Sigma^k$ nun genau zwei ausgehende Kanten zu den Knoten $(v << 1) \mod 2^l$ und $(v << 1) + 1 \mod 2^l$, effektiv also den Knoten $2v \mod 2^l$ und $(2v + 1) \mod 2^l$ besitzt. Für fest gewählte k und l ergibt sich allgemein, dass jeder im resultierenden de-Bruijn-Graphen enthaltene Knoten $v \in [0, k^l)$ gerichtete Kanten auf alle Knoten der Menge $N_v = \{x \in [0, k^l); x = ((v << 1) + z) \mod k^l, z \in \Sigma\}$ besitzt; ein konkretes Beispiel für einen de-Bruijn-Graphen mit $k = 2$ und $l = 3$ findet sich in Abbildung 4.5. De-Bruijn-Graphen besitzen darüber hinaus die Eigenschaft, in jedem Fall sowohl einen eulerschen, als auch einen hamiltonschen Kreislauf zu besitzen und werden in der Bioinformatik unter anderem zur Analyse von Genomsequenzen eingesetzt; für Koorde dienen sie jedoch einem gänzlich anderen Zweck.

4.2.2. Grundstruktur des Koorde-Rings

Koorde kombiniert nun das Konzept eines de-Bruijn-Graphen mit dem Ansatz eines ringförmigen Adressraums, wie dieser beispielsweise im Chord-Verfahren vorliegt, indem zwar die für die Ringstruktur nötigen Zeiger je Knoten auf dessen jeweiligen unmittelbaren Vorgänger bzw. Nachfolger beibehalten wird, jedoch die „eigentliche“ Routinginformationen, die sog. Finger, nun durch die unmittelbaren Nachbarn des Knotens im aus dem Adressraum konstruierten de-Bruijn-Graphen, d.h. dem de-Bruijn-Graphen für $k = 2$ und $l = n$, die Anzahl an Bits für die Referenzen ersetzt. Im aus Sicht des de-Bruijn-Routings bestmöglichen Fall ist der Ring dann voll besetzt und jeder Knoten `ref` verweist mit jeweils einem Zeiger auf den Knoten unter der Referenz $2\texttt{ref} \mod 2^n$ und $2\texttt{ref} + 1 \mod 2^n$. Zur Navigation innerhalb des de-Bruijn-Graphen lässt sich schließlich die Tatsache verwenden, dass die Referenzen der Nachbarn eines Knotens eben einer einmaligen bitweisen Verschiebung seiner Referenz nach links entsprechen, wobei von links dann jeweils ein Bit, je nach Wahl 0 oder 1, nachgeschoben wird: Ähnlich der bei *Pastry*[RD01] und *Tapestry*[ZKJ+01] vorhandenen schrittweisen Präfix- bzw. Suffixanpassung wird hier nun bitweise die gesuchte Referenz von der Referenz des suchenden Knotens ausgehenden von rechts in den aktuell betrachteten Referenzwert nachgeschoben, indem der de-Bruijn-Graph entsprechend traversiert wird; einzige unter der Nutzung der Bit-Shifts und ohne eine Betrachtung möglicher, bereits vorhandener Ähnlichkeiten zwischen dem Suffix der Referenz des aktuell betrachteten Knotens ergäbe sich so ein laufzeitlicher Aufwand je Auflösungsvorgang in $O(n)$. Nutzt man eine solche eventuell auftretende Übereinstimmung aus, verkleinert sich der Aufwand je Auflösungsvorgang ebenfalls. Eine algorithmische Beschreibung des Verfahrens für vollständige de-Bruijn-Graphen ist in Algorithmus 4.9 zu sehen.

Algorithmus 4.9 Trivialer Lookup in einem vollständigen de-Bruijn-Graphen nach [KK03]

```
1: procedure NODE.LOOKUP(target, kshift)
2:     if node == target then
3:         return node
4:     else
5:         Bit ← topBit(kshift)
6:         kshift ← (kshift << 1)   mod 2^n
7:         return node.neighbor[Bit].lookup(target, kshift)
8:     end if
9: end procedure
```

In diesem Fall lohnt sich ebenfalls die Betrachtung eines Beispiels zur besseren Verständlichkeit des Beschriebenen: Sei $k = 2$, $l = 5$; gesucht ist die Referenz $s = 21$, binär $s = 10101_2$, der suchende Ausgangsknoten besitze die Referenz `ref` $= 13$, binär 01101_2. Im naiven Ansatz beginnt Knoten 13 nun, die Bits von s schrittweise von rechts nach links in seine eigene Referenz zu schieben und dabei den Graphen anhand seiner Kanten zu traversieren: Zuerst wird das erste Bit, eine 1, nachgeschoben; die Anfrage wird also an den Knoten mit der Referenz $11011_2 = 27$, ein Nachbar von 13 übermittelt. Dieser fährt dann mit dem nächsten Bit, einer 0 fort und landet bei $10110_2 = 22$. Dieser wiederum schiebt das nächste Bit, eine 1 hinein und landet bei $01101_2 = 13$. Von dort aus wird dann auf gleiche Art und Weise zu $11010_2 = 26$ und schließlich zum Zielknoten mit der gesuchten Referenz 10101 vermittelt; das Ergebnis dann anschließend an den Sucher zurück übermittelt. Aus diesem Beispiel wird bereits ersichtlich, dass auch ein kürzerer Weg möglich ist, da die Weiterleitung in den ersten drei Schritten gewissermaßen im Kreis verläuft und so unnötigen Aufwand verursacht; ermittelt der suchende Knoten zu Beginn aber die größtmögliche Anzahl an übereinstimmenden Bits seines Suffixes mit dem Präfix der gesuchten Referenz, werden Kreisläufe vermieden und im Beispiel lediglich die letzten beiden Schritte ausgeführt.

4.2.3. Koorde-Lookup

Leider entspricht die Realität aber nur selten dem für eine bestimmte Sache optimalen Ideal; so auch in diesem Fall, da, wie allgemein ersichtlich sein sollte, die meisten solcher Ringe nur äußerst spärlich besetzt sind und dementsprechend so gut wie nie alle zum Aufbau eines vollständigen de-Bruijn-Graphen nötigen Referenzen mit Knoten besetzt sind. Um die Eigenschaften des de-Bruijn-Graphen dennoch nutzbar zu machen, wird der de-Bruijn-Graph innerhalb des Netzwerks deshalb nur so weit wie im Einzelfall möglich angenähert, indem jeder Knoten für alle ihm unmittelbar nachfolgenden Referenzen als für diese verantwortlicher de-Bruijn-Knoten auftritt, gewissermaßen also diese „fehlenden“ Knoten ersetzt. Zur Auflösung einer Referenz, wieder im naiven Ansatz, wird also wieder die gesuchte Referenz bitweise in die Referenz des auflösenden Knotens nachgeschoben und daran anlehnend das Netzwerk traversiert, wobei bei jeder Weiterleitung jetzt darauf zu achten ist, dass der aktuell weiterleitende Knoten auch wirklich der für die aktuell betrachtete imaginäre Referenz (bzw. imaginärer Knoten in [KK03]; gemeint ist damit die sich nach einer bestimmten Anzahl bitweiser Verschiebungen ergebende Referenz) zuständige Knoten ist oder nicht. Letzterer Fall tritt vor allem dann auf, wenn der Knoten, von dem der aktuell betrachtete Knoten die Anfrage empfangen hat (d.h. der Vorgänger im de-Bruijn-Graph) zwar für die vorige imaginäre Referenz zuständig war, der jetzige Knoten aber aufgrund der ungleichmäßigen Verteilung der Knoten auf

Algorithmus 4.10 Trivialer Lookup in einem Koorde-Ring nach [KK03]

```
1:  procedure NODE.LOOKUP(target, imag ,kshift)
2:      if target ∈ (node.node, node.successor.node] then
3:          return node
4:      else if imag ∈ (node.node, node.successor.node] then
5:          Bit ← topBit(kshift)
6:          kshift ← kshift << 1   mod 2^n
7:          imag ← (imag << 1   mod 2^n) + Bit
8:          return node.neighbor.lookup(target, imag, kshift)
9:      else
10:         return node.successor.lookup(target, imag, kshift)
11:     end if
12: end procedure
```

dem Ring nicht mehr für dessen imaginäre Nachbarreferenz verantwortlich ist; hier muss die Nachricht dann an den unmittelbaren Nachfolger des aktuell behandelnden Knotens weitergeleitet werden, ohne dabei eine weitere bitweise Verschiebung durchzuführen, da der Nachfolger oder einer dessen Nachfolger der für den imaginären Knoten zuständige Knoten ist. Von diesem aus wird dann die Suche wie für den de-Bruijn-Graph beschrieben weiter fortgesetzt, bis der für die Referenz zuständige Knoten, hier wie bei Chord auch der unmittelbare Nachfolger der Referenz, gefunden ist; wobei sich, wie auch im vollständigen de-Bruijn-Graphen, eine nötige Anzahl von $O(n)$ Schritten/Hops je Anfrage ergibt. Eine algorithmische Darstellung des Verfahrens findet sich hierfür in 4.10. Zu beachten ist für diesen Fall im Übrigen auch, dass jeder Knoten unter der Referenz `ref` nun nur noch einen Zeiger auf den Vorgänger seines jeweiligen ersten de-Bruijn-Nachbarn $2\texttt{ref} \mod 2^n$ besitzt, da der Vorgänger des zweiten de-Bruijn-Nachbarn $2\texttt{ref} \mod 2^n$ mit äußerst großer Wahrscheinlichkeit offensichtlicherweise dem des ersten de-Bruijn-Nachbarn entspricht und somit lediglich redundant Speicher verbrauchen würde; die bitweise Verschiebung erfolgt jedoch immer noch mit beiden möglichen Werten 0 und 1.

Problematisch ist an dieser Art Verfahren jedoch die Tatsache, dass die Anzahl nötiger Schritte/Hops je aufzulösender Referenz unangenehm hoch ist; je nach Größe des Adressraums ist eine deutlich zu große Menge an Nachrichten auszutauschen, die sich mit dem Beitritt oder Abgang weiterer

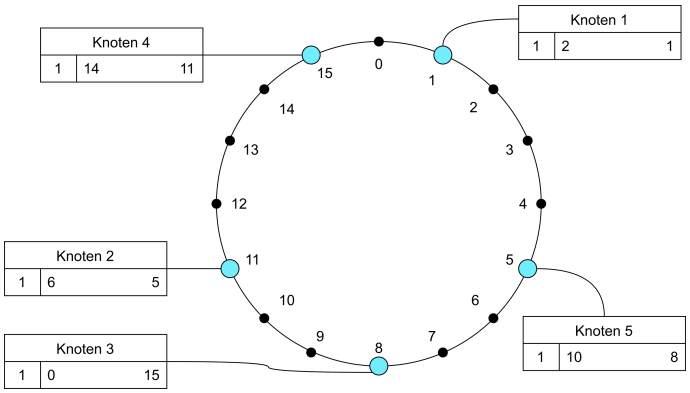


Abbildung 4.6.: Ein beispielhafter Koorde-Ring mit $n = l = 4$ Bits

Algorithmus 4.11 Verbesserter Lookup für Koorde

```
1: procedure NODE.LOOKUP(key)
2:     i, imag ← findLongestSuffix(node.node, key)
3:     kshift ← (key << i)  mod 2^n
4:     return node.lookup(key, imag, kshift)
5: end procedure
```

Knoten nur unmerklich verkleinert. Um dem zu begegnen, wurde von den Autoren eine geringfügige Änderung des Verfahrens vorgeschlagen, welche die Anzahl an nötigen Schritten je Lookup mit großer Wahrscheinlichkeit auf eine Anzahl in $O(\log(N))$ reduziert: Statt den imaginären Zwischenknoten, der für die weitere Suche verwendet wird, anfangs auf den Wert der Referenz des suchenden Knotens zu setzen, wird nun aus dem Referenzbereich, für welchen der die Suche auslösende Knoten zuständig ist, diejenige Referenz als Startwert ausgewählt, die den längstmöglichen Suffix unter allen Referenzen bilden kann, der mit dem längsten Präfix der gesuchten Referenz übereinstimmt. Da die Knoten im Durchschnittsfall jeweils für einen Bereich von $\frac{2^n}{N}$ Referenzen als deren unmittelbarer Vorgänger verantwortlich sind, unterscheidet sich jeder Knoten im Durchschnittsfall von seinem unmittelbaren Nachfolger erst an dem Bit an der Stelle $\log(\frac{2^n}{N}) = \log(2^n) - \log(N) = n - \log(N)$; alle vorigen Bits sind somit frei wählbar, d.h. die letzten (geordnet von links nach rechts) $O(n - \log(N))$ Bits können mit den ersten $O(n - \log(N))$ Bits der gesuchten Referenz besetzt werden und es verbleiben genau $O(\log(N))$ zu traversierende Bits. Genau dieser Algorithmus liegt dem eigentlichen Koorde-Verfahren zugrunde; eine algorithmische Beschreibung liegt in Algorithmus 4.11 vor, hier unter Nutzung des bereits vorig beschriebenen Koorde-Lookup-Algorithmus.

Ganz besonders auch in diesem Fall eignet sich auch hier die Formulierung eines Beispiels zu besseren Anschaulichkeit des Verfahrens; zugrunde liegt wieder das in Abbildung 4.6 beschriebene Koorde-Netzwerk; hier soll nun, von Knoten 1 unter der Referenz 1 ausgehende, die Referenz $13 = 1101_2$ auf den für diese Referenz zuständigen Knoten im Netzwerk aufgelöst werden. Unter Nutzung des nicht-trivialen Ansatzes zur Initialisierung des ersten imaginären Knotens bestimmt Knoten 1 zuerst die von ihm de-Bruijn-Vorgänger verwaltete Referenz mit dem längsten, mit einem Präfix der gesuchten Referenz übereinstimmenden Suffix; in diesem Fall ist das die Referenz $3 = 0011_2$, da hier die letzten beiden Bits mit den ersten beiden Bits der gesuchten Referenz übereinstimmen. Zuerst prüft Knoten 1, ob er selbst bereits der unmittelbare Vorgänger der gesuchten Referenz ist, was hier jedoch nicht der Fall ist. Da der aktuell gewählte imaginäre de-Bruijn-Knoten 3 jedoch in seinen Zuständigkeitsbereich fällt, schiebt er die Referenz des imaginären Knotens um eine Stelle nach links und setzt an der letzten, jetzt „frei“ gewordenen Stelle das nächste, in diesem Fall das dritte, Bit der gesuchten Referenz ein und erhält damit als neuen imaginären Knoten den Wert $\texttt{imag} = 0110_2 = 6$; anschließend leitet er diese Informationen nun an seinen de-Bruijn-Nachbarn, sich selbst, weiter (d.h. hier: startet z.B. einen rekursiven Aufruf). Wieder bei Knoten 1 prüft dieser abermals, ob die gesuchte Referenz in seinem Zuständigkeitsbereich liegt, was hier aber nicht der Fall ist. Da auch der imaginäre Knoten außerhalb seines Zuständigkeitsbereiches liegt, leitet er die Anfrage direkt an seinen unmittelbaren Nachfolger, Knoten 5, weiter. Dieser prüft wiederum, ob er der für diese Referenz zuständige Knoten ist und geht dann, da dies nicht der Fall ist, wie Knoten 1 vor und prüft, ob der imaginäre Knoten $\texttt{imag} = 6$ in seinen Zuständigkeitsbereich fällt. Da dies der Fall ist, schiebt er den Wert des imaginären Knotens wieder eine Stelle weiter und leitet den entstehenden Wert mitsamt Anfrage dann an seinen de-Bruijn-Nachbarn, Knoten 3, weiter; nun ist

`imag` = $1101_2 = 13$ und `kshift` = 0000_2. Knoten 3 ist aber weder für die gesuchte Referenz, noch für den imaginären Knoten zuständig, also schickt er seine Anfrage weiter an Knoten 2. Dieser ist schließlich für beide zuständig, kennt dementsprechend auch den unmittelbaren Nachfolger der Referenz 13, Knoten 4, beendet die Suche und gibt das Ergebnis dann an Knoten 1 zurück. Verfolgt man den „eigentlichen" Pfad der Anfrage in einem imaginären, vollständigen de-Bruijn-Graphen, kann man gut erkennen, was genau hier passiert: Die Anfrage wandert von Referenz 3 zu 6, indem von Knoten 1 auf Knoten 5 gewechselt wird. Von dort aus wandert die Anfrage weiter zu Referenz 13, indem von Knoten 5 über Knoten 3 auf Knoten 2 gewechselt wird.

4.2.4. Knotenbeitritt

Wie beim bereits behandelten Chord-Verfahren wird auch dieser Lookup-Algorithmus zur Initialisierung der jeweiligen Nachbarn eines Knotens bei dessen Beitritt verwendet, wiewohl nun nur genau 3 Zeiger je Knoten zu speichern sind. Zwar wurde von *Kaashoeck und Karger* keine genaue algorithmische Ausformulierung des Beitrittsvorgangs eines Knotens in ein bestehendes Koorde-Netzwerk angegeben, sondern lediglich auf die mögliche Anpassung und Nutzung des für Chord genutzten Beitrittsvorganges hingewiesen; allerdings gestaltet sich besonders die Aktualisierung der durch den Betritt des neuen Knotens in ihrem de-Bruijn-Zeiger betroffenen Knoten als aufwändiger, sodass sich eine genauere Betrachtung dieses Falls lohnt. Zu beachten ist, dass aufgrund der ringförmigen Anordnung des Adressraums und insbesondere des damit einhergehenden Umbruchs an der größtmöglichen Referenz möglicherweise nicht nur ein, sondern zwei betroffene Knoten in zwei verschiedenen Abschnitten des Adressraums liegen; zudem muss nicht der imaginäre de-Bruijn-Knoten gesucht werden, der die Referenz des beigetretenen Knotens als de-Bruijn-Nachbar referenziert, sondern der, welcher die Referenz des unmittelbaren Nachfolgers des neu beigetretenen Knotens referenziert. Der genaue Beitrittsvorgang ist daher mit Algorithmus 4.12 beschrieben und deckt diese Fälle alle ab: Zuerst werden die Vorgänger der durch den Beitritt möglicherweise betroffenen imaginären Knoten, sowohl mit als auch ohne Umbruch, berechnet und anschließend kontaktiert. Sollte der beigetretene Knoten in der Tat auch der neue de-Bruijn-Nachbar des kontaktierten Knotens sein, ändert er seinen de-Bruijn-Zeiger entsprechend ab und leitet die Nachricht an seinen unmittelbaren Vorgänger weiter, der dann damit genauso verfährt, bis alle anzupassenden de-Bruijn-Zeiger angepasst worden sind; eine algorithmische Ausformulierung dazu findet sich in Anhang B.2.

Ebenfalls für Koorde anwendbar ist das für Chord entwickelte Stabilisierungsverfahren, hier allerdings nur mit einem zu aktualisierenden Zeiger; für die Einbindung der zur Absicherung gegen spontane Knotenausfälle verwendeten Nachfolger-Liste indes ergeben sich hier andere Herausforderungen. Zwar ist die Verwendung einer solchen Liste an Nachfolgern durchaus hilfreich, um den für die korrekte Beantwortung einer Anfrage nötigen Zeiger auf den jeweiligen unmittelbaren Nachfolger eines Knotens korrekt zu halten, nicht abwegig; allerdings ist zur Aufrechterhaltung der de-Bruijn-Zeiger eine andere Vorgehensweise nötig. Intuitiverweise ginge man davon aus, dass eine Art Liste aller unmittelbaren Vorgänger dieses Zeigers ausreichend wäre, um mit Ausfällen umgehen zu können, allerdings gibt das Verfahren leider keine Garantie auf die allzeitige Richtigkeit der Zeiger auf den Vorgänger eines Knotens, wodurch eine solche ggf. inkorrekt wäre. Abhilfe schaffen lässt sich hier durch die Anfrage an einen, dem eigenen de-Bruijn-Nachbarn $2\texttt{ref} \mod 2^n$ um einen Faktor in $O(\log(N))$ vorgehenden imaginären Knoten, bzw. eigentlich an dessen Vorgänger, über dessen Liste an $O(\log(N))$ Nachfolgern, die wiederum den gesuchten Vorgänger von $2\texttt{ref} \mod 2^n$,

Algorithmus 4.12 Algorithmus zum Beitritt eines Knotens zu einem Koorde-Rings

```
 1: procedure NODE.JOINNETWORK(helper)
 2:     node ← initReference()
 3:     if helper == null then
 4:         successor ← node
 5:         predecessor ← node
 6:         neighbor ← node
 7:     else
 8:         successor ← helper.lookup(node.ref)
 9:         predecessor = successor.predecessor
10:         notify(predecessor)
11:         pointer ← (node.node << 1)  mod 2^n
12:         neighbor = helper.findPredecessor(pointer)
13:         updateOthers()
14:     end if
15: end procedure
```

sowie alle seine nächsten $O(\log(N))$ Vorgänger, diesmal mit Korrektheitsgarantie, beinhaltet und anschließend als solche Liste verwendet werden kann. Einher damit geht jedoch ein zusätzlicher Speicheraufwand je Knoten in $O(\log(N))$, was den eigentlichen Vorteil eines Speicheraufwandes in $O(1)$, den Koorde gegenüber Chord besaß, zunichtemacht; ebenfalls problematisch ist die sich dadurch erhöhende Anzahl an auszutauschenden Nachrichten je Knotenbeitritt.

4.2.5. Ansätze zur Leistungsoptimierung

Um diesen Faktor an zusätzlichem Aufwand nutzbar zu machen, wird von den Autoren nach Erörterung des Problems die Nutzung eines de-Bruijn-Graphen mit einer Basis in $O(\log(N))$, d.h. mit $k = \log(N)$ vorgeschlagen. In der Umsetzung eines solchen Graphen auf einem ringförmigen Adressraum besäße jeder Knoten unter der Referenz `ref` einen Zeiger auf den unmittelbaren Vorgänger der Referenz $k \cdot \texttt{ref} \mod k^l$, sowie auf dessen k unmittelbare Nachfolger, um Ausfallsicherheit zu gewährleisten; für k bietet sich ein Wert mit $k = 2^x$ an. Der je Knoten zu haltende Zustand eines solchen Koorde-Rings wäre in einer Größenordnung von $O(k)$ verortet; wählt man jedoch $k \approx \log(N)$, so ergibt sich einerseits ein zu haltender Zustand in $O(\log(N))$, dafür aber lediglich eine Anzahl an zu besuchenden Knoten je aufzulösender Referenz in $O(\frac{\log(N)}{\log(\log(N))})$, verglichen mit Chord also ein deutlicher Zugewinn an Effizienz. In Hinblick auf die Verwendung kleiner Adressräume scheint eine wirkliche Effizienz dieser Verbesserung im Hinblick auf den dafür nötigen zusätzlichen Aufwand, insbesondere aufgrund der großen Menge an auszutauschenden Nachrichten beim Beitritt eines neuen Knotens, bzw. unter Nutzung des Stabilisierungsalgorithmus im Hinblick auf die benötigte Zeit, zwar fraglich; für größere Adressräume ist ein Zugewinn an Effizienz hier aber als äußerst wahrscheinlich anzunehmen.

4.3. Mögliche allgemeine Optimierungen der Verfahren

Oftmals zu beobachten ist, dass die Zuteilung des Adressraums aufgrund einer ungünstigen Wahl der die Position der Knoten jeweils bestimmenden Referenzen mehr oder minder stark unausgeglichen wirkt und manche Knoten daher lediglich einen verhältnismäßig kleinen, andere Knoten einen verhältnismäßig großen Bereich an Referenzen zu verwalten haben, es dementsprechend zu einer ungleichmäßigen Verteilung der Anzahl an je Knoten verwalteter Referenzen mit ggf. zugehörigen Daten, wie auch der Arbeitslast an Knoten je Auflösung kommt. Lohnenswert ist eine solche Überlegung vor allem aufgrund der Tatsache, dass für die hier betrachteten Fälle kleine Referenzen und damit einhergehend kleine Adressräume, mit $n \leq 64$ Bit, verwendet werden, welche zumeist spärlicher besetzt sind, allgemein auch nur dünner zu besetzen sind, als große Adressräume und eine gleichmäßige Aufteilung des Adressraums auf die einzelnen Knoten somit nicht unbedingt zu erwarten ist. Stattdessen erfolgt häufig die Aufteilung des Adressraumes häufig zufällig, aufgrund der dünnen Besetztheit in den meisten Fällen unregelmäßig und bis zu einem gewissen Grad ungleich, wodurch die für die Verfahren erarbeiteten Garantien nur verzerrt erfüllt/wiedergegeben werden. Tatsächlich trifft eine diese Aussage vornehmlich zunächst auf Adressräume mit $n < 48$ Bit zu, da hier, ausgehend von einem Speicherkontingent je Knoten von etwa einem Terabyte, eine Besetzung der einzelnen Ringe mit etwa 1000 Knoten bereits ausreichen sollte; unter der Verwendung konsistenten Hashings jedoch wird sich die Anzahl an nötigen Knoten für eine effektive Besetzung des gesamten Adressraums aufgrund des als „Geburtstagsparadoxon“ bekannten Phänomens maßgeblich verringern, wodurch nun ebenfalls auch für Referenzen einer Länge von beispielsweise $n = 64$ Bit eine deutlich geringere Anzahl an Knoten nötig wird, um den Adressraum voll zu besetzen.

Allgemein bietet sich in der Aufrechterhaltung einer gleichmäßigen Verteilung der je Knoten verwalteten Referenzbereiche also den Vorteil einer gleichmäßigen Lastverteilung je Lookup: Für Chord im Speziellen könnte eine laufzeitmäßig vorteilhafte Auswirkung darin bestehen, dass so einerseits die jeweils durch die einzelnen Finger referenzierten Nachbarknoten jetzt ebenfalls deutlich gleichmäßiger besetzt sind, d.h. der Unterschied an tatsächlichem Speicheraufwand je Knoten auch für kleinere Systeme ausgeglichen; ein Knoten verwaltet dann mit hoher Wahrscheinlichkeit lediglich einen Bereich der Größe $\frac{2^n}{N}$, $\frac{2^n}{2N}$ oder $\frac{2^{n-1}}{N}$. Durch diese gleichmäßige Verteilung der Knoten und die damit einhergehende einheitliche Größe der je Knoten verwalteten Teilbereiche ergibt sich zudem auch ein laufzeitlicher Vorteil für Chord, da so auch die Länge der Pfade je Auflösungsanfrage, ausgehenden vom die Suche anstoßenden Knoten als Wurzelknoten halbwegs ausgeglichen ist und keine überaus großen Entartungen/Unregelmäßigkeiten mehr aufweist; ähnlich einem $n - ren$m ausgeglichenen Suchbaum, ausgehend vom suchenden Knoten. Hinzuzufügen ist, dass durch eine bessere Abschätzung des Wertes von $\log(N)$ gegebenenfalls die Anzahl der je Knoten zu verwaltenden Finger zumindest ein wenig dynamischer gestaltet werden kann, da ja immer nur eine Anzahl in $O(\log(N))$ Fingern besetzt ist; gegebenenfalls ergeben sich hierdurch zumindest kleinere laufzeitmäßige, definitiv aber auf den Speicherplatzverbrauch bezogene Vorteile.

Für das standardmäßige, nicht-angepasste Koorde-Verfahren ergäbe sich mit einer solchen Aufteilung immerhin die Möglichkeit einer einfacheren Abschätzung des Wertes für $\log(N)$, mit welchem sich etwa die anfänglich für den imaginären Knoten eines Lookups gewählte Referenz einfacher abschätzen ließe; der laufzeitliche Zugewinn ist hier jedoch eher marginal. Möglicherweise von vorteilhafter Auswirkung auf die für eine Auflösung benötigte Zeit bestehen ggf. in der genaueren Treffsicherheit bei der Suche nach dem de-Brujin-Nachbarn des in einem Lookup aktuell verwendeten

imaginären Knotens, wobei der genaue Leistungszugewinn hier ebenfalls eher gering ausfallen wird; eine weit bessere Optimierung bestünde beispielsweise in der Einrichtung eines zweiten de-Brujin-Nachbarn beispielsweise von der mittig im von Knoten verwalteten Bereich an imaginären Knoten liegenden Referenz.

Vorgestellt wurde ein solches Verfahren zur Einrichtung eines in großem Maße gleichmäßig verteilten Adressraums beim Aufbau des Overlay-Netzwerks, beispielsweise von *Naor und Wiederer*[NW03], als unerlässliche Grundlage zur Initialisierung eines von denselben Autoren beschriebenen Distance-Halving-Netzwerks, mit dem vergleichsweise einfachen Grundgedanken einer jeweiligen Aufteilung des größten im Netzwerk enthaltenen Referenzbereiches in zwei gleich große Teile; wobei einer der Teile dem bisherigen Besitzer zufällt und der andere Teil dem neuen Knoten. Gewissermaßen entspricht diese Art der Aufteilung des Adressraums etwa der Aufteilung bei Beitritt eines Knotens in einem 1-dimensionalen CAN [RFH+01] - als nichts anderes kann ein solcher Ring ja betrachtet werden - und erfolgt konkret, wie folgt: Dem ersten dem Netzwerk beitretenden Knoten wird dabei zunächst der gesamte verfügbare Adressraum unterstellt; alle weiteren Knoten ermitteln bei ihrem Beitritt dann an $c \cdot \log(N)$ unterschiedlichen Positionen jeweils den Abstand zum rechten und linken Nachbarn dieser Position und werden dann im größten gefundenen Segment mittig eingefügt, wobei eine Abschätzung für $\log(N)$ beispielsweise aus der Größe des vom zum Beitritt gewählten Hilfsknoten verwalteten Referenzbereichs ermittelt werden kann. Um diese Anpassung im Beitrittsvorgang nun für Chord und Koorde nutzbar zu machen, muss sowohl die Ermittlung der Größe eines Segments, als auch die Ermittlung der zu wählenden Position des neuen Knotens auf dem Ring geringfügig angepasst werden: Statt $c \cdot \log(N)$ Positionen auf ihre Segmentgröße abzufragen, wird nun für jede der gegebenen Positionen jeweils der unmittelbare Nachfolger bestimmt und anschließend die Größe des Bereiches zwischen diesem Knoten und dessen unmittelbarem Nachfolge zurückgegeben. Der beitretende Knoten wählt schließlich den größten Bereich aus und platziert sich in der Mitte dieses Bereiches; Nachfolger und Vorgänger des Knotens wurden dann bereits ermittelt und könnten im beigetretenen Knoten sofort initialisiert werden. Für eine weitere Beschäftigung mit diesem Verfahren ist eine algorithmische Beschreibung mit Algorithmus 4.13 angegeben.

Besonderen Nutzen besäße eine solche Aufteilung des Adressraums allerdings für die auf einem de-Bruijn-Graphen mit Ausgangsgrad $k = \log(N)$ aufbauende Optimierung des Koorde-Verfahrens, bei welcher insbesondere eine gute Möglichkeit zur Abschätzung der Größe $\log(N)$ mittels weniger auszutauschender Nachrichten, möglicherweise sogar in $O(1)$, möglich wäre. Eine genauere Abschätzung für $k = \log(N)$ ermöglicht aufgrund des nun genauer vorliegenden Wertes für die Anzahl an je Knoten zu speichernden Nachbarn in diesem Fall eine Verbesserung der Aktualität der je Knoten gespeicherten Nachbarn und dementsprechend möglicherweise auch eine Verbesserung der tatsächlich benötigten Zeit je Auflösung. Da nach bisheriger Kenntnis keine Implementierung oder Auswertung einer solchen Kombination des angepassten Koorde-Verfahrens für einen Grad in $O(\log(N))$ mit einer gleichmäßigen Verteilung der Knoten auf dem Koorde-Ring existiert, wäre eine Analyse des möglichen Zeitzugewinns je Lookup für eine solche Implementierung durchaus lohnenswert.

Alternativ zur verbesserten Variante des Koorde-Verfahrens für den Einsatz in kleinen Adressräumen ebenfalls von großer Bedeutung sind die vorig bereits kurz erwähnten One/Two-Hop-Verfahren, von denen verglichen mit $\log(N)$-Koorde eine weitere Verkürzung der nötigen Laufzeit je Auflösung zu erwarten ist und deren vergleichsweise große Menge an zu speichernden Nachbarverweise in $O(N)$,

Algorithmus 4.13 Initialisierung der Position eines Knotens auf einem Ring, bei gleichmäßiger Aufteilung des Adressraums

```
1:  procedure NODE.INITREFERENCE(helper)
2:      if helper == null then
3:          node.node ← getRandomNBitNumber(n)
4:      else
5:          size ← (helper.successor - helper.node)   mod 2^n
6:          logn ← ⌈log(2^n / size)⌉
7:          inode ← helper
8:          for 1,...,c·logn do
9:              number ← getRandomNBitNumber(n)
10:             newNode ← helper.findSuccessor(number)
11:             if size < ((newNode.successor - newNode.node)   mod 2^n) then
12:                 inode ← newNode
13:                 size ← ((inode.successor - inode.node)   mod 2^n)
14:             end if
15:         end for
16:         node.node ← (inode.node + size/2)   mod 2^n
17:     end if
18: end procedure
```

bzw. $O(\sqrt{N})$ aufgrund der spärlichen Besetztheit des Adressraums nicht allzu große Auswirkungen auf die allgemeine Leistungsfähigkeit dieser Verfahren haben sollte. Insbesondere ein Vergleich dieser Verfahren mit $\log(N)$-Koorde scheint durchaus angebracht und sinnvoll.

5. Vergleich und Analyse der verwendeten Verfahren

Um nun die Leistungsfähigkeit strukturierter Auflösungsverfahren für kleine Adressräume und somit auch für kleine bzw. kurze Referenzen zu ermitteln, wurden einzelne ausgewählte Verfahren, die in Kapitel 4 bereits ausführlicher beschrieben wurden, gegeneinander und zur Ermittlung des tatsächlich durch die Kommunikation der einzelnen Knoten untereinander entstehenden Zusatzaufwandes auch gegen eine Implementierung des Systems mit zentralem Indexknoten, ähnlich dem Aufbau des anfänglichen Napster-Netzwerks, verglichen. Der Vergleich der einzelnen Verfahren erfolgt hauptsächlich hinsichtlich der für die Auflösung einer im System gespeicherten Referenz benötigten Zeit, unter zusätzlicher Betrachtung der nötigen Anzahl an zu speichernden Nachbarknoten je Knoten im System; die Auswirkungen entsprechender Anpassungen der Verfahren sind ebenfalls beschrieben, soweit diese ausgewertet wurden.

5.1. Aufbau der Versuche

Für die Messungen selbst wurden die zu messenden Verfahren auf einem bereits bestehenden Kommunikationsgrundgerüst aufbauend, inklusive der Unterstützung aller nötigen Funktionalitäten, implementiert und anschließend mittels eines mehrere, ebenfalls selbst erstellte Beispieldatensätze in Form von Befehlslisten ausführenden Clients ausgewertet, welcher zu einer solchen Auswertung die Listen jeweils strikt sequenziell ausführt und während dieser Ausführung für jede Auflösung einer Referenz die benötigte Zeit misst, aufsummiert und anschließend durch die Anzahl an erfolgten Auflösungen teilt, um so einen Durchschnittswert der benötigten Zeit zu erhalten. Eine kurze genauere Beschreibung der einzelnen Listentypen ist in Unterabschnitt 5.1.1 gegeben; die Anzahl an Befehlen je ausgeführter Liste ist in der jeweiligen Auswertung mitangegeben. Unterschieden wurden bei den Messungen dann sowohl die Anzahl an im System enthaltener Knoten, sowie die Größe des Adressraums, bzw. die Länge einer Referenz in Bits, um einen entsprechend vielseitigen Vergleich zu bieten; insbesondere die Zunahme des Unterschieds in der durchschnittlich benötigten Zeit je Lookup zwischen der zentralen und allen dezentralen Varianten bei steigender Knotenanzahl ist hierbei als maßgeblich für die Bestimmung der Leistungsfähigkeit eines Verfahrens in diesem Fall zu erachten.

Beginnend bei einem Netzwerk von fünf Knoten wird die Anzahl der Knoten mit jeder Vergrößerung verdoppelt, bis eine Anzahl von 2000 Knoten im Netzwerk erreicht ist; die einzige Ausnahme davon findet sich in einem Sprung von 20 auf 50 Knoten, statt der erwarteten Verdopplung auf 40 Knoten. Abstufungen der Größe des Adressraums, somit auch in der Größe der einzelnen verwendeten Referenzen sind mit jeweils 16, 24, 32 und 64 Bit gewählt worden und damit deutlich kleiner gewählt, als dies bei einer sonst standardmäßig verwendeten Länge von 128 bzw. 160 Bit der Fall wäre. Für die tatsächliche Umsetzung der Messungen mit den einzelnen Adressräumen muss in den

echt verteilten Systemen Chord und Koorde zusätzlich die als sogenanntes „Geburtstagsparadoxon“ bekannte Wahrscheinlichkeit der Wahl derselben Position auf dem Ring innerhalb des Adressraumes durch zwei verschiedene Knoten bedacht werden; eine solche Doppelbesetzung hätte für die korrekte Funktionsfähigkeit des Systems nach der Initialisierung aller Knoten weitreichend negative Folgen und würde ein Funktionieren des Systems in den allermeisten Fällen verhindern. Berechnet man für die gegebenen Adressraumgrößen jedoch die konkreten Eintrittswahrscheinlichkeiten eines solchen Falles für die unterschiedlichen Netzwerkgrößen, so stellt man fest, dass ein solches Problem lediglich bei Verwendung eines 16-Bit-Adressraums, kombiniert mit einer Netzwerkgröße größer 213 Knoten, mit einer Wahrscheinlichkeit größer $0,5$ auftritt, weshalb daher auf eine Auswertung der verteilten Verfahren in einem solchen Adressraum mit jeweils 256, 512 und 1024 Knoten verzichtet werden kann.

Eine Übersicht über die Eigenschaften des zur Durchführung der Messungen verwendeten physischen Rechners findet sich in Anhang A.2.

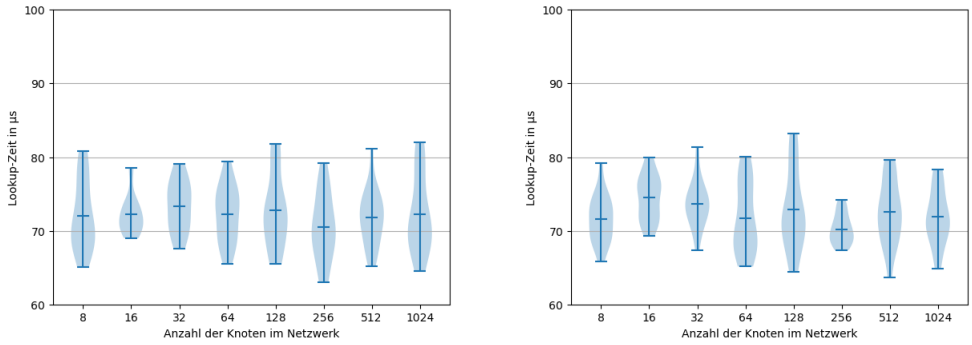
5.1.1. Verwendete Datensätze

Begründet durch die Tatsache, dass für den Betrieb eines verteilten Speichersystems neben reinen Lookup-Operationen zudem auch Operationen zum Einfügen, Entfernen und gegebenenfalls auch zum Verschieben einer Referenz, beispielsweise im Falle eines bereits vollen physischen Speichers auf einem bestimmten Knoten, anbietet, wurden auch diese Fälle in die Tests miteingebracht und durch die Anlage von vier verschiedenen Befehlstypen (Auflösung/Lookup, Einfügen/Insert, Entfernen/Remove, Verschieben/Update) abgedeckt. Zur konkreten Messung der Effizienz einzelner Lookup-Verfahren wurden daher Listen mit unterschiedlichen Kombinationen und Auftrittswahrscheinlichkeiten der beschriebenen Befehle erstellt, die jeweils ein bestimmtes Szenario repräsentieren sollen und getrennt voneinander ausgewertet wurden, wiewohl die Listen zum Teil sequenziell hintereinander ausgeführt werden können und auch wurden. Die genaue Aufschlüsselung der Listen in die einzelnen enthaltenen Befehlstypen mitsamt deren Auftrittswahrscheinlichkeit finden sich in Tabelle 5.1; zu beachten ist dabei, dass Listen des Typs „Init“ nur zur Initialisierung eines Systems mit einer bestimmten Anzahl an Referenzen verwendet wurden und, mangels enthaltener Lookup-Befehle, nicht zur Analyse genutzt werden können.

Je nach gewähltem Adressraum wurde die Größe der verwendeten Befehlsliste angepasst, sodass die Listen für den 16-Bit-Adressraum beispielsweise „nur“ 20.000 Befehle enthalten, um ein Aus- oder Überreizen des gesamten Adressraums zu vermeiden, während die Listen des 24-Bit-Adressraumes jeweils 40.000 Befehle enthalten. Für die beiden verbliebenen Adressräume der Größe 32 und

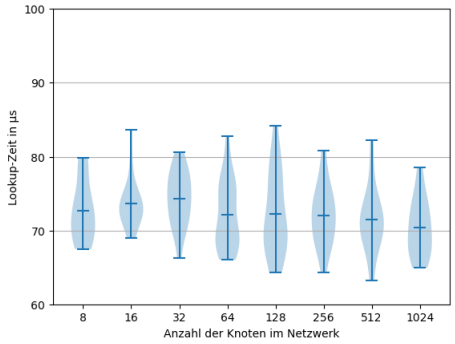
Listen-Kategorie	Lookup	Insert	Remove	Update
Only-Lookup	1	0	0	0
Init	0	1	0	0
Changing	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	0
Lookup	$\frac{2}{3}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
Equal	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

Tabelle 5.1.: Aufschlüsselung der Befehlslisten nach dem Anteil des jeweiligen Befehlstyps

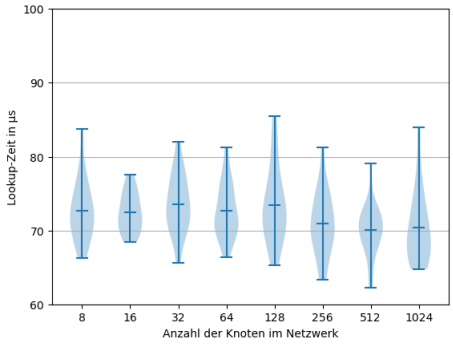


(a) Only-Lookup-Datensatz

(b) Changing-Datensatz



(c) Lookup-Datensatz



(d) Equal-Datensatz

Abbildung 5.1.: Auswertung der zentralisierten Variante für eine Adressraumgröße von $n = 64$ Bit

64 Bit wurden jeweils Listen mit 50.000 enthaltenen Befehlen erzeugt; eine größere Anzahl an abzuarbeitenden Befehlen je Liste wäre im gegebenen zeitlichen Rahmen nicht umsetzbar gewesen.

5.2. Ergebnisse

5.2.1. Zentralisierte Variante

Zuerst erfolgt und angegeben sind die Messungen der durchschnittlich für die Auflösung einer Referenz benötigten Zeit in μs mittels eines zentralen Indexknotens, für die jeweiligen Adressraumgrößen aufgeschlüsselt und in die einzelnen Befehlslistenarten unterteilt in der Abbildung 5.1, sowie für alle anderen Adressräume in Anhang B.3 angegeben. Begründet ist diese, vielleicht seltsam anmutende Aufteilung der Präsentation der Messergebnisse damit, dass für die beiden verteilten Verfahren Messungen außerhalb des 64-Bit-Adressraumes aufgrund des zeitlichen Rahmens dieser Arbeit leider nicht möglich gewesen sind, Messungen für die anderen Adressräume im Rahmen der zentralisierten Variante aber schon erfolgt sind und dem Leser daher nicht vorenthalten werden sollen, jedoch der Übersichtlichkeit wegen in den Anhang verlegt wurden.

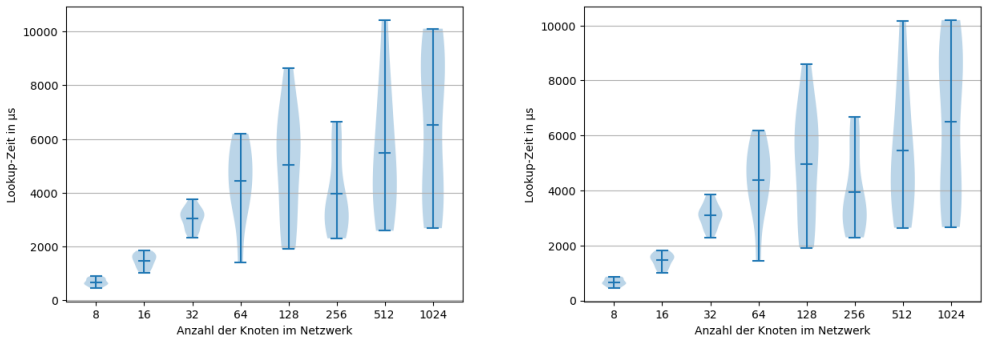
Ferner liegen für jede mögliche Kombination an Netzwerkgröße, Datensatzart und Adressraumgröße gleich zwei Datensätze der jeweils für die Adressraumgröße typischen Länge vor, die, für die zentralisierte Variante, je zwanzigmal, für Chord und Koorde jeweils mindestens zehnmal, im Falle der beiden kleinsten Netzwerkgrößen ebenfalls zwanzigmal ausgemessen wurden.

Da die zentralisierte Verwaltung der Zuordnung einer im System verwendeten Datenreferenz zu einem im System enthaltenen Knoten als mögliche Topologie in der Aufteilung des Arbeitsaufwands zur Auflösung einer solchen Referenz ein laufzeitmäßiges Optimum darstellt, werden die für dieses Verfahren gemessenen Zeitwerte im folgenden als Grundlage für alle weiteren Vergleiche dienen, anhand derer die Effizienz der beiden anderen Verfahren gemessen werden kann. Im Rahmen dieses Verfahrens erwartbar und im Ergebnis dann auch messbar ist ein stets konstanter Verlauf der benötigten Zeit je Auflösung über alle eingesetzten Netzwerkgrößen, gänzlich unabhängig von der Anzahl an im Netzwerk befindlichen Knoten, wie sich ein solcher denn dann auch, immerhin jedoch mit einigen Schwankungen, in den Messungen widerspiegelt; die Konstantheit des Aufwandes ist also recht gut ersichtlich.

Als Mittelwert für diese Größe für einen 64-Bit-Adressraum und über alle Netzwerkgrößen hinweg lässt sich grob eine Zeitspanne von etwa $70-75\mu s$ je Auflösungsvorgang angeben; allgemein bewegt sich dieser Wert etwa zwischen $65\mu s$ und $85\mu s$, wobei hauptsächlich sowohl kleinere Abweichungen nach oben und unten zu beobachten sind. Die zusätzlichen Messwerte der anderen Adressräume in Anhang B.3 bestätigen die offensichtlich erwartbare Konstantheit des zentralisierten Verfahrens, sowie die Plausibilität der für den 64-Bit-Adressraum gemessenen Werte ebenfalls.

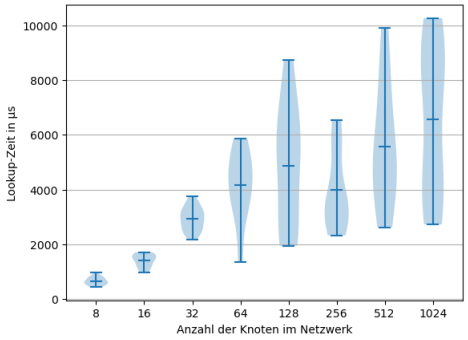
5.2.2. Chord

Im Gegensatz zur vorig betrachteten zentralisierten Variante zur Auflösung der Referenz, mit einem erwartbaren und gemessenen Laufzeitaufwand in $O(1)$ ist für Chord hier eine deutliche Steigerung zu erwarten, mit einer benötigten Anzahl an Schritten/Hops in $O(\log(N))$ für Netzwerkgröße $N \in \mathbb{N}$, gegenüber dem konstanten Aufwand beim zentralen Indexknoten in $O(1)$; diese Steigerung ist dann auch zu beobachten und erreicht hier schon für die kleinste Netzwerkgröße etwa eine Verzehnfachung der aufzuwendenden Zeit und ist zumindest allgemein als eher unangenehm zu bewerten. Vorteilhaft gegenüber der vorigen Variante ist für Chord jedoch die Tatsache, dass aufgrund der dezentralen Verwaltung der Zuordnung einzelner Referenzen zum jeweils zuständigen Knoten kein zentraler Ausfallpunkt im System mehr existiert und das Netzwerk, zumindest im dafür angepassten Modus, im Falle eines Knotenausfalls seine vollständige Funktionalität beibehält, abgesehen von den auf dem ausgefallenen Knoten gespeicherten Daten. Zumindest im groben ersichtlich ist dieser logarithmische Anstieg des Aufwandes je Auflösung einer Referenz in den in Abbildung 5.2 mittels einzelner Schaubilder dargestellten Messergebnisse, jeweils mit der Verteilung der einzelnen Messwerte, den Extrema als Grenzen, sowie dem arithmetischen Mittelwert über alle Messwerte angegeben; die logarithmische Skalierung an der y-Achse ist hierfür zu beachten. Für eine Netzwerkgröße von 8 und 16 Knoten wurden die Messungen jeweils in zwanzigfacher, für alle anderen Netzwerkgrößen in zehnfacher Wiederholung ausgeführt, wobei auch hier wieder dieselben Datensätze wie zur Ausmessung des zentralisierten Verfahrens verwendet wurden. Jeder Datensatz wurde dabei mithilfe eines einzelnen, im Chord-Ring enthaltenen Server-Knotens zuerst ausgemessen, anschließend wurde über alle gemessenen Werte der arithmetische Mittelwert gebildet und dieser dann für jeden Durchlauf eines Datensatzes gespeichert.

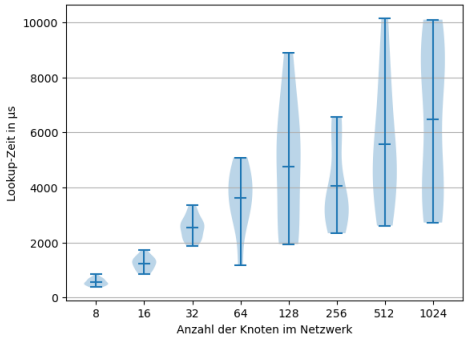


(a) Only-Lookup-Datensatz

(b) Changing-Datensatz



(c) Lookup-Datensatz



(d) Equal-Datensatz

Abbildung 5.2.: Auswertung des Chord-Verfahrens für eine Adressraumgröße von $n = 64$ Bit

Auffällig ist auf den Schaubildern einerseits die deutliche Zunahme in der Bandbreite der gemessenen Werte mit Zunahme der Netzwerkgröße und andererseits die mit dieser einhergehenden Schwankung zwischen den einzelnen Werten für die unterschiedlichen Netzwerkgrößen; insbesondere die Werte für 32, 64 und 128 Knoten stechen durch eine merkliche Überhöhung hervor, während die darauffolgenden Werte für 256 und 512 Knoten durch eine merkliche Unterbietung des erwarteten Mittelwertes auffallen. Während geringfügige Schwankungen, wie in Abbildung 5.1 noch durch Schwankungen im Ablauf des Programmes auf dem ausführenden Prozessor, beispielsweise in Bezug auf Scheduling und Caching zurückzuführen sind, ist bei Schwankungen in einer derartigen Größenordnung, mit einer derartigen Zunahme eine zusätzliche Quelle vonnöten; diese findet sich dann auch in der genauen Einteilung des Adressraumes auf die einzelnen Knoten und der damit einhergehenden Struktur der Fingertabellen der einzelnen Knoten. Da jede der Messungen je Datensatz mittels eines im Chrod-Ring enthaltenen, beliebig, jedoch fest gewählten (Knoten-ID: 1) Server-Knotens durchgeführt wurde, hängt die genaue Ausgestaltung der Messwerte stark mit der Position dieses Knotens, bzw. der Positionierung der anderen im Ring enthaltenen Knoten zusammen: Sind die Knoten auf dem Ring halbwegs gleichmäßig verteilt, ergibt sich eine nahezu optimale Situation für den Server-Knoten, da so die Mehrheit der Anfragen wirklich in $\log(N)$ Schritten, zuzüglich eines kleinen konstanten Faktors, beantwortet werden. Sind die Knoten jedoch ungleichmäßig auf dem Ring verteilt und steht der Server-Knoten an einer ungünstigen Stelle,

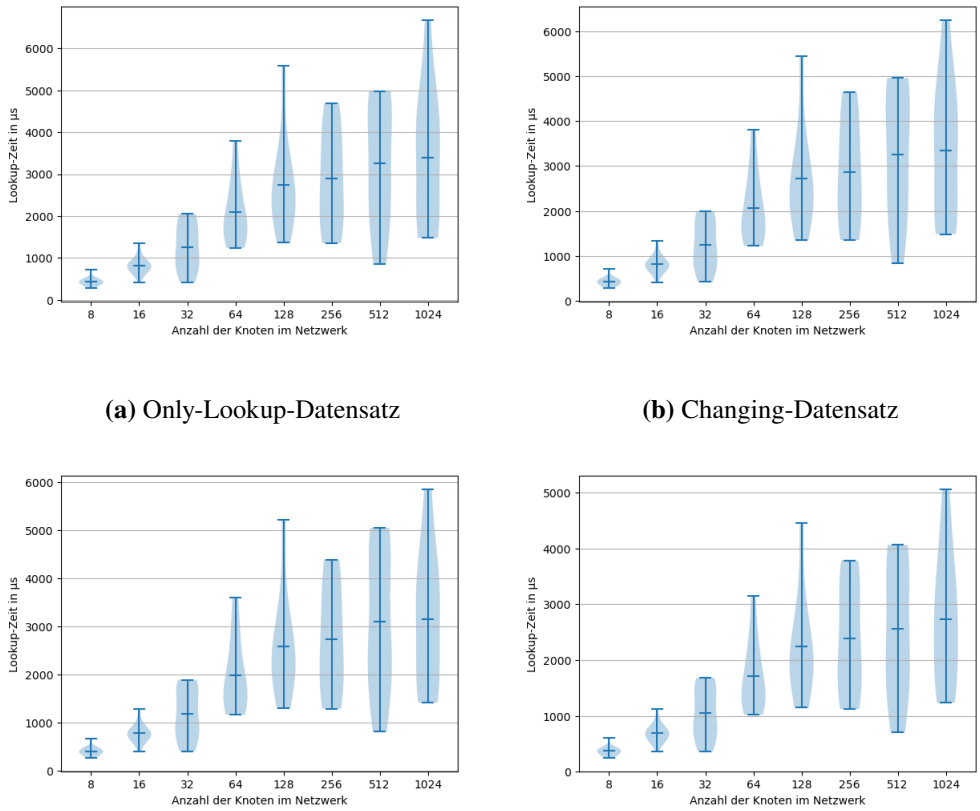
sodass dessen Fingertabelle dünner als erwartbar, d.h. mit weniger als $\log(N)$ Einträgen besetzt ist, kann sich die Anzahl an Schritten je Anfrage deutlich erhöhen. Zur bildlichen Anschauung ist es hier hilfreich, sich das Chord-Netzwerk nicht als Ring, sondern als den Graphen vorzustellen, den besagter Ring ja nur zum besseren Verständnis der Einteilung des Adressraums abstrahiert; die Kanten zwischen den Knoten bilden dabei die einzelnen Finger. Ausgehend vom Server-Knoten als Wurzelknoten lässt sich nun eine Art Baum für die mit Chord durchgeführte, gerichtete Tiefensuche innerhalb des Graphen angeben, wobei die Pfadlänge je Suchvorgang bei einer gleichmäßigen Verteilung des Adressraumes auf die einzelnen Knoten halbwegs ausgeglichen sein sollte, während bei einer ungleichmäßigen Verteilung die Pfadlänge ausufern kann; hierin liegt also der eigentliche Grund für die Schwankungen.

Mithin ein weiterer, jedoch recht geringer Einflussfaktor ist die Anzahl der vom Server-Knoten verwalteten Referenzen und die damit einhergehende Wahrscheinlichkeit einer trivialen Auflösung der Referenz auf den Server-Knoten selbst oder dessen unmittelbarem Nachfolger; ein solcher Fall tritt jedoch lediglich mit einer Wahrscheinlichkeit in $O(\frac{1}{N})$ auf und ist somit nur für besonders kleine Netzwerkgrößen relevant. Zur Erklärung des drastischen Sprunges zwischen den Werten für eine Netzwerkgröße von 16 und 32 Knoten (bzw. 32 und 64 Knoten bei Koorde; vergleiche Abbildung 5.3) beispielsweise könnte diese Tatsache zumindest angeführt werden; weitaus größeren Einfluss haben hier selbstverständlich die bei den Messungen aufgetretenen Schwankungen.

5.2.3. Koorde

Gegenüber dem zentralen Indexknoten besitzt Koorde ebenfalls den Nachteil einer größeren benötigten Laufzeit, mit einer Anzahl an Schritten/Hops in $O(\log(N))$ je Anfrage und teilt mit Chord den Vorteil einer dezentralen Verwaltung der zu speichernden Zuordnung der Referenzen zum jeweils verantwortlichen Knoten, besitzt gegenüber Chord aber den Vorteil einer deutlich geringeren Anzahl an je Knoten zu speichernden Nachbarn in $O(1)$ (Chord: $O(\log(N))$). Bezogen auf die tatsächlich benötigte Antwortzeit je Auflösungsanfrage könnten sich für Koorde indes sowohl Vorteile bezüglich der Anzahl an zu untersuchenden möglichen Nachbarn zur Weiterleitung der Anfrage - Chord besitzt davon zumeist etwa $\log(N)$ je Knoten, während Koorde lediglich zwei dergleichen besitzt - ergeben, als auch ein Nachteil darin, dass bei einer Weiterleitung der Anfrage über den de-Bruijn-Nachbarn eines Knotens gegebenenfalls eine weitere Weiterleitung an dessen unmittelbaren Nachfolger aufgrund der Unstimmigkeiten zwischen der Größe des Raumes der möglichen de-Bruijn-Nachbarn des weiterleitenden Knotens und der Größe des Raumes der vom de-Bruijn-Nachbarn verwalteten imaginären de-Bruijn-Knoten ergeben; welcher der beiden Faktoren überwiegt, gilt es herauszufinden.

Die Messungen erfolgten ausschließlich auf dem Rechner mit den in Tabelle A.2 angegebenen Eigenschaften, wobei bis zu den Messungen bis zu einer Netzwerkgröße von einschließlich 512 Knoten 4 Threads in den beiden Kommunikationskomponenten verwendet wurden, während zum Ausmessen der letzten Netzwerkgröße von 1024 Knoten dort lediglich ein Thread verwendet wurde; hierbei aber ohne sichtlich negativen Einfluss auf die Messergebnisse; in allen anderen Aspekten erfolgte die Messung in gleicher Art und Weise, wie bei dem zuerst ausgemessenen Chord-Verfahren. Ebenfalls wie für Chord ist auch in den Schaubildern für Koorde ein mit dem Anstieg der Netzwerkgröße einhergehender logarithmisch gearteter Anstieg der real benötigten durchschnittlichen Zeit zur Auflösung einer Referenz zu entnehmen; das Verfahren arbeitet



(a) Only-Lookup-Datensatz

(b) Changing-Datensatz

(c) Lookup-Datensatz

(d) Equal-Datensatz

Abbildung 5.3.: Auswertung des Koorde-Verfahrens für eine Adressraumgröße von $n = 64$ Bit

also erwartungsgemäß performant, mit einer die Laufzeit bestimmenden Anzahl an Schritten je Auflösungsvorgang in $O(\log(N))$; in anschaulichen Schaubildern dargestellt finden sich die Messwerte in 5.3.

Trotz dieser Übereinstimmung in der erwartbaren Größenordnung der benötigten Zeit je Auflösungsanfrage in Abhängigkeit von der Anzahl an im Netzwerk enthaltenen Knoten finden sich merkliche Unterschiede für alle ausgemessenen Werte für Koorde gegenüber den Messwerten für Chord; grob gesagt ist das Koorde-Verfahren hier überraschenderweise etwa doppelt so schnell, wie das Chord-Verfahren, einhergehend mit einem deutlich verringerten Speicheraufwand an zu speichernden Nachbarn je Knoten. Weiterhin bemerkbar ist die auch hier auftretende Aufwandssteigerung gegenüber dem zentralisierten Verfahren; hier jedoch gegenüber Chord lediglich mit einer Verfünf- bis Versechsfachung der für eine Auflösung notwendigen Zeit zu veranschlagen.

Ein bereits erwähnter Ansatz zur Erklärung dieser Tatsache findet sich darin, dass der zur Weiterleitung der Nachricht benötigte Aufwand bei Koorde signifikant kleiner ist, als bei Chord, erklärt diese Beobachtung aber nur teilweise, da es sich dabei ja lediglich um eine lokale Operation auf einem Speicherfeld (engl. Array) des jeweiligen Knotens handelt. Der wesentlich höhere, laufzeitliche, wie kommunikative Aufwand ergibt sich unter anderem auch dadurch, dass für die Implementierung des Chord-Verfahrens, insbesondere des Routings einer Anfrage, ein rein iterativer Ansatz verwendet

wurde, weshalb auch lediglich eine virtuelle und keine wirkliche Weiterleitung der Nachricht stattfindet und im Rahmen dieses iterativen Vorgehens entweder die benötigten Informationen für die einzelnen Finger, wie dargestellt in Tabelle 4.1, lokal auf dem Knoten gespeichert werden, was mit jeder Änderung dieser Informationen durch den Beitritt eines Knotens und der damit möglicherweise einhergehenden Änderung des unmittelbaren Vorgängers bzw. Nachfolgers eines solchen Fingers einen erheblichen zusätzlichen Kommunikationsaufwand je Knotenbeitritt bedeutet, oder alternativ diese Informationen mit jedem Zugriff auf den Finger erneut abgefragt werden. In letzterem Fall erhöht sich der Aufwand zur Laufzeit stark, wodurch die eben beobachtete Verdopplung der Zeit, da nun nicht mehr eine, sondern zwei Nachrichten je Weiterleitung gesendet werden müssen. Als Ausgleichsmaßnahme wäre hier eine Verschiebung des Kommunikationsaufwandes hin zur Initialisierung des Netzwerkes zu prüfen oder allgemein eine rekursive Implementierung des Chord-Verfahrens, abweichend von der Beschreibung in [SMK+01].

Wie beim Chord-Verfahren bereits beobachtet, treten auch hier mit zunehmender Netzwerkgröße zunehmende Schwankungen in den Messwerten auf; die Bandbreite an gemessenen Werten vergrößert sich. Ähnlich wie für das Chord-Verfahren, jedoch nicht auf genau gleiche Art und Weise, hängt diese Tatsache wiederum mit der Gleichmäßigkeit der Aufteilung des Adressraumes zusammen, wobei der Zusammenhang hier jedoch unmittelbarer sichtbar wird, da die Anzahl an zu besuchenden Knoten je Auflösungsanfrage nun mit der Anzahl an bereits durch ein Präfix der gesuchten Referenz besetzten Suffix-Bits des ersten, von Server gewählten imaginären Knotens abhängt. Dabei gilt: Je größer der Bereich an vom Server verwalteten imaginären Knoten, heißt: Je größer der Bereich zwischen dem Server-Knoten und seinem unmittelbaren Nachfolger, desto mehr Bits können auf diese Art und Weise anfänglich besetzt werden, desto weniger Bits sind noch schrittweise aufzulösen und desto weniger Knoten sind für einen Auflösungsvorgang zu besuchen. Für eine zufällige oder pseudozufällige Wahl der Position eines Knotens auf dem Koorde-Ring ergeben sich hier entsprechend je nach der konkreten Wahl der Knoten vorteilhaftere oder weniger vorteilhafte Anordnungen der Knoten; ein Ansatz zur Verminderung dieser Schwankungen besteht in der bereits in Abschnitt 4.3 abgesprochenen, gleichmäßigen Verteilung des Adressraumes auf die einzelnen Knoten, wobei die wirkliche Wirksamkeit einer solchen Anpassung noch zu überprüfen ist.

Weiterhin feststellbar ist eine leichte, jedoch bemerkbare Verminderung der je Auflösungsanfrage benötigten Zeit für den Equal-Datensatz in Abbildung 5.3d, sprich: den Datensatz mit dem geringsten Anteil an solchen Anfragen. Der Lookup-Datensatz scheint dabei ein ähnliches Verhalten offenzulegen; für die anderen beiden Datensätze ist das Verfahren dagegen ein wenig schlechter performant. Erklärbar ist diese Beobachtung möglicherweise durch die in diesen beiden Datensätzen vorhandene Auslagerung bereits gespeicherter Referenzen auf einen anderen Knoten als Speicherort, wodurch die Wahrscheinlichkeit einer trivialen Auflösung einer Referenz, da diese bereits von Server-Knoten gespeichert wird, zwar leicht ansteigt, jedoch nicht ausreichend stark genug, um diesen großen Unterschied zu erklären. Zu beachten ist insbesondere auch die Tatsache, dass genau das gleiche Phänomen unter Verwendung des Chord-Verfahrens nur in deutlich abgeschwächter Ausprägung, jedoch bei genauem Hinsehen merkbar, auftritt; es handelt sich dabei also vermutlich um ein implementierungsspezifisches Detail, das, gepaart mit der genannten Erklärung zu eben dieser Beobachtung führt.

6. Zusammenfassung

Auch wenn die Zusammenhänge der mit der Auflösung einer in einem verteilten System verwendeten Referenz auf ein in diesem System gespeicherten Datum verbundenen Herausforderungen bereits vor längerer Zeit betrachtet, erörtert und mit konkreten Lösungsansätzen in Form bestimmter Algorithmen und Verfahren weitgehend überwunden sind, soll mit dieser Arbeit trotzdem der Versuch eines Beitrags zu einem Teilgebiet dieses doch recht umfangreich ausgestalteten Sachzusammenhanges unternommen werden; eine besondere Gewichtung sollte dabei auf die Betrachtung der Gegebenheiten bei der Verwendung kleiner Referenzen, beziehungsweise kleiner Adressräume gelegt werden. Zunächst wurden die allgemeinen Hintergründe zur Thematik, insbesondere das für die Auflösung zentrale Lookup-Problem erklärt und erörtert, sowie mit diesen einhergehenden Eigenschaften bestimmter Lösungsansätze benannt und beschrieben. Unter der Betrachtung der bisher ausgearbeiteten Lösungsansätze dieses Problems wurden schließlich zwei konkrete Verfahren, Chord und Koorde, ausgewählt, im Detail beschrieben und erklärt, sowie implementiert; anschließend wurde zuerst auf möglicherweise auftretende Probleme bei der Verwendung kleiner Referenzen mit Lookup-Verfahren dieser Art allgemein eingegangen und bezogen auf die verwendeten Verfahren anschließend auf mögliche Verbesserungsvorschläge eingegangen.

Kapitel 5 bietet schließlich einen Vergleich beider Verfahren miteinander und gegenüber einem einen zentralen Indexknoten verwendenden, laufzeitlich optimal arbeitenden Verfahren; hier zeigt sich einerseits ein, wenn auch erwartbarer, so jedoch ungünstig gesteigerter Aufwand der beiden dezentralen Verfahren gegenüber dem zentralen Verfahren, jedoch unter dem Zugewinn höherer Ausfallsicherheit. Im Vergleich beider dezentraler Verfahren bewährt sich Koorde durch eine gegenüber Chord deutlich verminderte, beinahe halbierten aufzuwendenden Zeitspanne je Auflösungsanfrage, die aber nur in Teilen auf Eigenheiten der Verfahren selbst zurückzuführen ist, zumindest durch eine geschicktere Implementierung des Chord-Verfahrens um einen nicht unerheblichen Faktor vermindert werden könnte; anzunehmen ist jedoch, dass das Koorde-Verfahren aber auch dann noch wegen seines gegenüber Chord geringeren Speicheraufwandes an unmittelbaren Nachbarn in $O(1)$ und der damit einhergehenden lokalen laufzeitlichen Besserstellung sichtlich zu bevorzugen ist. Zudem durch die Messungen bestätigt wurde die bereits zuvor geäußerte Annahme einer negativen Auswirkung der ungleichmäßigen Aufteilung des vorhandenen Adressraumes auf die einzelnen Knoten, die sich hierbei durch eine große Bandbreite an gemessenen Werten erkennbar macht. Bis auf deren Beschreibung, mitsamt einer kurzen algorithmischen Beschreibung eines Verfahrens zur gleichverteilten Einteilung des Adressraumes konnte aufgrund der zeitlichen Vorgaben jedoch leider keiner der vorgestellten Verbesserungsansätze beider Verfahren weiter behandelt oder gar implementiert werden.

6.1. Ausblick

Wiewohl mit dieser Arbeit eine Grundlage zur Analyse zweier strukturierter Verfahren zur Auflösung von Datenreferenzen in P2P-Systemen mit kleinen Adressräumen gelegt wurde, gilt es dennoch, weitere Ansätze zur effizienten Lösung dieses Problems zu untersuchen.

Aufgrund ihrer laufzeitmäßig besonders leistungsfähigen Arbeitsweise scheinen beispielsweise die sogenannten One- oder Two-Hop-Verfahren als besonders geeignete Kandidaten für eine weitergehende Untersuchung, da mit einem solchen, eine Auflösung in $O(1)$ Schritten/Hops für $N \in \mathbb{N}$ Knoten ausführenden Verfahren in etwa die Effizienz der optimalen zentralisierten Variante erreicht wird und aufgrund des allgemein nicht allzu dicht besetzbaren, kleinen Adressraums ein Ausufern der nötigen Anzahl an zu speichernden Nachbarn verhindert werden kann. Die Analyse, mitsamt Vergleich solcher One-/Two-Hop-Verfahren konnte innerhalb dieser Arbeit zwar aufgrund zeitlicher Vorgaben nicht umgesetzt werden, ist aber gerade im Hinblick auf deren Leistungsfähigkeit stark zu empfehlen. Im Falle eines, wie in Abschnitt 4.3 vorgestellten, halbwegs gleichmäßig auf die einzelnen Knoten aufgeteilten Adressraumes wären zudem weitere Optimierungen im Hinblick auf eine Reduzierung der Größe der zu speichernden Anzahl an Nachbarn, gepaart mit einer freien wählbaren Anzahl an zur Auflösung notwendigen Schritten denkbar.

Allgemein empfehlenswert ist ebenfalls die Umsetzung der im bereits erwähnten Abschnitt 4.3 vorgestellten Verbesserungen, mitsamt einer Auswertung von deren Leistungsfähigkeit; insbesondere ist ein besonderes Augenmerk auf die Verknüpfung des für de-Bruijn-Graphen mit Grad $k = \log(N)$ angepassten Koorde-Verfahrens mit der Einrichtung eines gleichmäßig aufgeteilten Adressraums zu legen, welche unter den mit einer Anzahl von $O(\log(N))$ Nachbarn arbeitenden Verfahren wohl die größte Leistungsfähigkeit besitzt.

Noch unbeachtet geblieben sind Abwandlungen halb-zentralisierter Verfahren, die mehrere zentrale Indexknoten zu Verwaltung der Zuordnung von Referenzen zu den die Daten besitzenden Knoten nutzen, wie auch allgemein unstrukturierte Verfahren; wobei zumindest letztere aufgrund ihres großen Kommunikationsaufwands und ihrer schwachen Garantien eher ungeeignet erscheinen. Gegebenenfalls ergäbe sich aber, zumindest in der Verknüpfung dieser beiden Verfahrenstypen, ebenfalls Möglichkeiten zum effizienten Einsatz in kleinen Adressräumen.

Ansätze zum Einsatz von Caching Verfahren oder einer Replikation einiger Daten auf mehreren Knoten wurden bisher ebenfalls nicht beachtet, erscheinen im Hinblick auf einen Einsatz in kleinen Adressräumen aber eher ungeeignet; insbesondere wieder aufgrund der dünnen Besetztheit dieser Adressräume und der damit einhergehenden Kürze der für die Auflösung zurückzulegenden Pfade innerhalb des Netzwerks. Der zusätzliche Aufwand an Speicher und zur Kommunikation einzusetzender Nachrichten würde sich aller Vermutung nach mit dem zusätzlichen Aufwand der One-Hop-Verfahren aufwiegen, welche hier mit deutlich höherer Leistungsfähigkeit einsetzbar wären.

Literaturverzeichnis

[AS07] J. Aspnes, G. Shah. „Skip Graphs“. In: *ACM Trans. Algorithms* 3.4 (Nov. 2007), 37–es. ISSN: 1549-6325. DOI: 10.1145/1290672.1290674. URL: https://doi.org/10.1145/1290672.1290674 (zitiert auf S. 25, 61).

[BFMJ08] D. Bin, W. Furong, J. Ma, L. Jian. „Enhanced Chord-Based Routing Protocol Using Neighbors' Neighbors Links“. In: *22nd International Conference on Advanced Information Networking and Applications - Workshops (aina workshops 2008)*. 2008, S. 463–466. DOI: 10.1109/WAINA.2008.53 (zitiert auf S. 25).

[BKK+03] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica. „Looking up data in P2P systems“. en. In: *Communications of the ACM* 46.2 (Feb. 2003), S. 43–48. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/606272.606299. URL: https://dl.acm.org/doi/10.1145/606272.606299 (besucht am 29. 12. 2022) (zitiert auf S. 13, 17).

[CDKR02] M. Castro, P. Druschel, A.-M. Kermarrec, A. Rowstron. „Scribe: a large-scale and decentralized application-level multicast infrastructure“. In: *IEEE Journal on Selected Areas in Communications* 20.8 (2002), S. 1489–1499. DOI: 10.1109/JSAC.2002.803069 (zitiert auf S. 26).

[CMM02] R. Cox, A. Muthitacharoen, R. T. Morris. „Serving DNS Using a Peer-to-Peer Lookup Service“. In: *Peer-to-Peer Systems*. Hrsg. von P. Druschel, F. Kaashoek, A. Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 155–165. ISBN: 978-3-540-45748-0 (zitiert auf S. 23).

[CRB+03] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, S. Shenker. „Making Gnutella-like P2P Systems Scalable“. In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '03. Karlsruhe, Germany: Association for Computing Machinery, 2003, S. 407–418. ISBN: 1581137354. DOI: 10.1145/863955.864000. URL: https://doi.org/10.1145/863955.864000 (zitiert auf S. 21, 24).

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, T. W. Hong. „Freenet: A Distributed Anonymous Information Storage and Retrieval System“. In: *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. Hrsg. von H. Federrath. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 46–66. ISBN: 978-3-540-44702-3. DOI: 10.1007/3-540-44702-4_4. URL: https://doi.org/10.1007/3-540-44702-4_4 (zitiert auf S. 20, 24).

[De 46] N. G. De Bruijn. „A combinatorial problem“. In: *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam* 49.7 (1946), S. 758–764 (zitiert auf S. 25, 37).

[FG06] P. Fraigniaud, P. Gauron. „D2B: A de Bruijn based content-addressable network“. In: *Theoretical Computer Science* 355.1 (2006). Complex Networks, S. 65–79. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2005.12.006. URL: https://www.sciencedirect.com/science/article/pii/S0304397505009163 (zitiert auf S. 25).

[GD05] S. Guha, N. Daswani. *An experimental study of the skype peer-to-peer voip system*. Techn. Ber. Cornell University, 2005 (zitiert auf S. 24).

[HDJ+02] N. J. Harvey, J. Dunagan, M. Jones, S. Saroiu, M. Theimer, A. Wolman. *SkipNet: A Scalable Overlay Network with Practical Locality Properties*. Techn. Ber. MSR-TR-2002-92. Note: The technical report previously numbered MSR-TR-2002-92, Ä Polynomial-time Rescaling Algorithm for Solving Linear Programs‚ïs now numbered MSR-TR-2003-09. Dez. 2002, S. 38. URL: https://www.microsoft.com/en-us/research/publication/skipnet-a-scalable-overlay-network-with-practical-locality-properties/ (zitiert auf S. 25).

[HP] V. Hilt, S. Previdi. *IRTF Peer-to-Peer Research Group (P2PRG) [CONCLUDED]*. Website. last checked: 23.03.2023. URL: https://irtf.org/concluded/p2prg (zitiert auf S. 17).

[Kir03] P. Kirk. *Gnutella - A Protocol for a Revolution*. Website. last checked: 23.03.2023. 2003. URL: https://rfc-gnutella.sourceforge.net/ (zitiert auf S. 13, 24).

[KK03] M. F. Kaashoek, D. R. Karger. „Koorde: A Simple Degree-Optimal Distributed Hash Table“. In: *Peer-to-Peer Systems II*. Hrsg. von M. F. Kaashoek, I. Stoica. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, S. 98–107. ISBN: 978-3-540-45172-3 (zitiert auf S. 3, 25, 37, 39, 40, 61).

[LCP+05] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim. „A survey and comparison of peer-to-peer overlay network schemes“. In: *IEEE Communications Surveys & Tutorials* 7.2 (2005), S. 72–93. DOI: 10.1109/COMST.2005.1610546 (zitiert auf S. 19, 23).

[LKR06] J. Liang, R. Kumar, K. W. Ross. „The FastTrack overlay: A measurement study“. In: *Computer Networks* 50.6 (2006). Overlay Distribution Structures and their Applications, S. 842–858. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2005.07.014. URL: https://www.sciencedirect.com/science/article/pii/S1389128605002446 (zitiert auf S. 24).

[LLD06] B. Leong, B. Liskov, E. D. Demaine. „EpiChord: Parallelizing the Chord lookup algorithm with reactive routing state management“. In: *Computer Communications* 29.9 (2006). ICON 2004, S. 1243–1259. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2005.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0140366405003750 (zitiert auf S. 26).

[LNS96] W. Litwin, M.-A. Neimat, D. A. Schneider. „LH*—a Scalable, Distributed Data Structure“. In: *ACM Trans. Database Syst.* 21.4 (Dez. 1996), S. 480–525. ISSN: 0362-5915. DOI: 10.1145/236711.236713. URL: https://doi.org/10.1145/236711.236713 (zitiert auf S. 24).

[MA09] L. Monnerat, C. Amorim. „Peer-to-Peer Single Hop Distributed Hash Tables“. In: *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. 2009, S. 1–8. DOI: 10.1109/GLOCOM.2009.5425764 (zitiert auf S. 26).

[MM02] P. Maymounkov, D. Mazières. „Kademlia: A Peer-to-Peer Information System Based on the XOR Metric“. In: *Peer-to-Peer Systems*. Hrsg. von P. Druschel, F. Kaashoek, A. Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 53–65. ISBN: 978-3-540-45748-0 (zitiert auf S. 21, 25, 61).

[MNR02] D. Malkhi, M. Naor, D. Ratajczak. „Viceroy: A Scalable and Dynamic Emulation of the Butterfly“. In: *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*. PODC '02. Monterey, California: Association for Computing Machinery, 2002, S. 183–192. ISBN: 1581134851. DOI: 10.1145/571825.571857. URL: https://doi.org/10.1145/571825.571857 (zitiert auf S. 25, 61).

[Moc87] P. Mockapetris. *Domain names - concepts and facilities*. RFC 1034. Nov. 1987. DOI: 10.17487/RFC1034. URL: https://www.rfc-editor.org/info/rfc1034 (zitiert auf S. 23).

[MR07] T. Moors, J. Risson. *Survey of Research towards Robust Peer-to-Peer Networks: Search Methods*. RFC 4981. Sep. 2007. DOI: 10.17487/RFC4981. URL: https://www.rfc-editor.org/info/rfc4981 (zitiert auf S. 18–20, 23).

[NW03] M. Naor, U. Wieder. „A Simple Fault Tolerant Distributed Hash Table“. In: *Peer-to-Peer Systems II*. Hrsg. von M. F. Kaashoek, I. Stoica. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, S. 88–97. ISBN: 978-3-540-45172-3 (zitiert auf S. 3, 25, 45).

[PGES05] J. Pouwelse, P. Garbacki, D. Epema, H. Sips. „The Bittorrent P2P File-Sharing System: Measurements and Analysis“. In: *Peer-to-Peer Systems IV*. Hrsg. von M. Castro, R. van Renesse. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, S. 205–216. ISBN: 978-3-540-31906-1 (zitiert auf S. 24).

[Pro11] S. Prohaska. *De Bruijn Graph and Sequence*. Presentation. 2011. URL: https://www.bioinf.uni-leipzig.de/Leere/SS11/GraphenNetzwerke_SP/VL05.pdf (zitiert auf S. 37).

[PRR97] C. G. Plaxton, R. Rajaraman, A. W. Richa. „Accessing nearby copies of replicated objects in a distributed environment“. In: *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*. 1997, S. 311–320 (zitiert auf S. 21, 24, 61).

[RCFB07] W. Rao, L. Chen, A. W.-C. Fu, Y. Bu. „Optimal Proactive Caching in Peer-to-Peer Network: Analysis and Application“. In: *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*. CIKM '07. Lisbon, Portugal: Association for Computing Machinery, 2007, S. 663–672. ISBN: 9781595938039. DOI: 10.1145/1321440.1321533. URL: https://doi.org/10.1145/1321440.1321533 (zitiert auf S. 23).

[RD01] A. Rowstron, P. Druschel. „Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems“. In: *Middleware 2001*. Hrsg. von R. Guerraoui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 329–350. ISBN: 978-3-540-45518-9 (zitiert auf S. 22, 24, 38, 61).

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker. „A Scalable Content-Addressable Network“. In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), S. 161–172. ISSN: 0146-4833. DOI: 10.1145/964723.383072. URL: https://doi.org/10.1145/964723.383072 (zitiert auf S. 22, 25, 45, 61).

[RGJZ04] S. Ren, L. Guo, S. Jiang, X. Zhang. „SAT-Match: a self-adaptive topology matching method to achieve low lookup latency in structured P2P overlay networks“. In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 2004, S. 83–. DOI: 10.1109/IPDPS.2004.1303022 (zitiert auf S. 22).

[SMK+01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan. „Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications“. In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 2001), S. 149–160. ISSN: 0146-4833. DOI: 10.1145/964723.383071. URL: https://doi.org/10.1145/964723.383071 (zitiert auf S. 3, 21, 22, 24, 27–30, 33–37, 54, 61, 63, 64).

[TBC+05] C. Tang, M. J. Buco, R. N. Chang, S. Dwarkadas, L. Z. Luan, E. So, C. Ward. „Low Traffic Overlay Networks with Large Routing Tables“. In: *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '05. Banff, Alberta, Canada: Association for Computing Machinery, 2005, S. 14–25. ISBN: 1595930221. DOI: 10.1145/1064212.1064216. URL: https://doi.org/10.1145/1064212.1064216 (zitiert auf S. 26, 61).

[VAI+08] M. Vapa, A. Auvinen, Y. Ivanchenko, N. Kotilainen, J. Vuori. „Optimal Resource Discovery Paths of Gnutella2“. In: *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*. 2008, S. 546–553. DOI: 10.1109/AINA.2008.89 (zitiert auf S. 24).

[WLW08] Y.-C. Wu, C.-M. Liu, J.-H. Wang. „Enhancing the Performance of Locating Data in Chord-Based P2P Systems“. In: *2008 14th IEEE International Conference on Parallel and Distributed Systems*. 2008, S. 841–846. DOI: 10.1109/ICPADS.2008.88 (zitiert auf S. 25).

[WTL+15] I. Woungang, F.-H. Tseng, Y.-H. Lin, L.-D. Chou, H.-C. Chao, M. S. Obaidat. „MR-Chord: Improved Chord Lookup Performance in Structured Mobile P2P Networks“. In: *IEEE Systems Journal* 9.3 (2015), S. 743–751. DOI: 10.1109/JSYST.2014.2306147 (zitiert auf S. 24).

[ZGG05] H. Zhang, A. Goel, R. Govindan. „Improving lookup latency in distributed hash table systems using random sampling“. In: *IEEE/ACM Transactions on Networking* 13.5 (2005), S. 1121–1134. DOI: 10.1109/TNET.2005.857106 (zitiert auf S. 21).

[ZKJ+01] B. Y. Zhao, J. Kubiatowicz, A. D. Joseph et al. „Tapestry: An infrastructure for fault-tolerant wide-area location and routing“. In: (2001) (zitiert auf S. 24, 38).

[ZZJ+01] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, J. D. Kubiatowicz. „Bayeux: An Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination“. In: *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV '01. Port Jefferson, New York, USA: Association for Computing Machinery, 2001, S. 11–20. ISBN: 1581133707. DOI: 10.1145/378344.378347. URL: https://doi.org/10.1145/378344.378347 (zitiert auf S. 26).

A. Allgemeine Zusatzinformationen

Name	Topologie	#Hops/Lookup	Anzahl Nachbarn	Referenz
Chord	Ring	$O(\log(N))$	$O(\log(N))$	[SMK+01]
Koorde	Ring/ de-Bruijn-Graph	$O(\log(N))$	$O(1)$	[KK03]
Kademlia	Ring/XOR	$O(\log(N))$	$O(\log_{2^b}(N))$	[MM02]
Pastry	PRR	$O(\log_{2^b}(N))$	$O(\log_{2^b}(N) \cdot 2^b)$	[RD01]
Viceroy	Butterfly	$O(\log(N))$	$O(1)$	[MNR02]
CAN	d-dimensionaler Torus	$O(dN^{\frac{1}{d}})$	$O(d)$	[RFH+01]
1h-/2h-Calot	Ring	$O(1)$	$O(\sqrt{N})/O(N)$	[TBC+05]
Skip-Graph	Skip-Lists	$O(\log(N))$	$O(\log(N))$	[AS07]

Tabelle A.1.: Vergleich mehrerer Lookup-Verfahren hinsichtlich deren Eigenschaften

Anmerkung: Mit PRR wird hierbei das von *Plaxton et al.* [PRR97] vorgestellte Konzept einer effektiven Suche nach einem bestimmten Knoten in einem P2P-System nahe liegenden Datenobjekt bezeichnet; auf genau diesem Suchkonzept basieren die damit bezeichneten Verfahren.

Prozessortyp	Kerne	RAM	Netzwerkadapter 1	Netzwerkadapter 2
Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz	8	8 GB	RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller	QCA9377 802.11ac Wireless Network Adapter

Tabelle A.2.: Eigenschaften des zur Messung verwendeten physischen Rechners

B. Zusätzliche Algorithmen und Schaubilder

B.1. Zum Beitrittsvorgang bei Chord (vgl. Unterabschnitt 4.1.3)

Nachfolgend angegeben sind die für den unangepassten Beitrittsvorgang eines Knotens zu einem Chord notwendigen, in der Beschreibung aus Platzgründen aber ausgelassenen Algorithmen:

Algorithmus B.1 Algorithmus zur Initialisierung der Finger, nach [SMK+01]

```
1:  procedure NODE.INITFINGERTABLE(helper)
2:      for i = 1,...,n do
3:          finger[i].start ← (node.node +2^(i−1))  mod 2^n
4:      end for
5:      finger[1] ← helper.findSuccessor(finger[1].start)
6:      node.successor ← finger[1].node
7:      node.predecessor ← finger[1].node.predecessor
8:      node.successor.predecessor ← node
9:      for i = 1,...,n do
10:         if finger[i+1].start ∈ (node, finger[i].node] then
11:             finger[i+1] ← finger[i]
12:         else
13:             finger[i+1] ← helper.findSuccessor(finger[i+1].start)
14:         end if
15:     end for
16: end procedure
```

Algorithmus B.2 Algorithmus zur Benachrichtigung bereits im Chord-Ring enthaltener Knoten, nach [SMK+01]

```
1:  procedure NODE.UPDATEOTHERS
2:      for i = 1,...,n do
3:          pre = findPredecessor((node.node - 2^(i−1))  mod 2^n)
4:          pre.updateFinger(node, i)
5:      end for
6:  end procedure
```

Anmerkung zur Abweichung in Zeile 2 in Algorithmus B.3: In [SMK+01] wird hier als Bedingung `new.node ∈ [node, finger[i].node)` angegeben; für den Fall `finger[i].node = node`, der bei der Initialisierung des zweiten Knotens im Netzwerk auftritt, ist diese Bedingung jedoch trivialerweise

Algorithmus B.3 Algorithmus zur Aktualisierung eines Fingers nach einem Knotenbeitritt, nach [SMK+01]

```
1: procedure NODE.UPDATEFINGER(new, i)
2:     if new.node ∈ [finger[i].start, finger[i].node) then
3:         finger[i] ← new
4:         predecessor.updateFinger(new, i)
5:     end if
6: end procedure
```

immer erfüllt und neue Knoten werden dann entsprechend als Finger eingetragen, obwohl diese gar nicht den vom Finger zu referenzierenden Knoten entsprechen. Zur Absicherung gegen solche Fälle wurde der Algorithmus daher geringfügig angepasst.

B.2. Zu Koorde

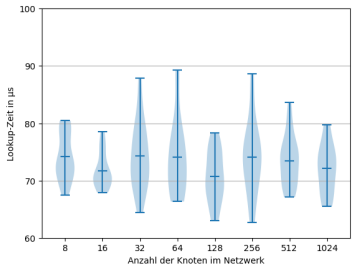
Algorithmus B.4 Algorithmus zur Benachrichtigung bereits im Koorde-Ring enthaltener Knoten

```
1: procedure NODE.UPDATEOTHERS(new, i)
2:     backPointer = node.node >> 1
3:     pre = findPredecessor(backPointer)
4:     pre.inform(node)
5:     backPointer = (node.node + 2^n) >> 1   mod 2^n
6:     second = findPredecessor(backPointer)
7:     if second != pre then
8:         second.inform(node)
9:     end if
10: end procedure
```

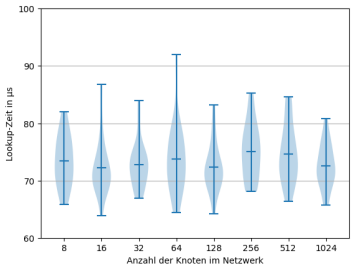
Algorithmus B.5 Algorithmus zur Aktualisierung des de-Bruijn-Nachbarn nach Knotenbeitritt

```
1: procedure NODE.INFORM(new)
2:     pointer = node.node << 1   mod 2^n
3:     if pointer ∈ ( new.node, new.successor] then
4:         node.neighbor = new
5:         node.predecessor.infor(new)
6:     end if
7: end procedure
```

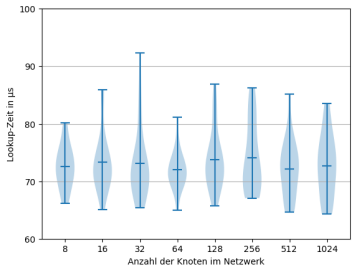
B.3. Zusätzliche Diagramme/Messwerte



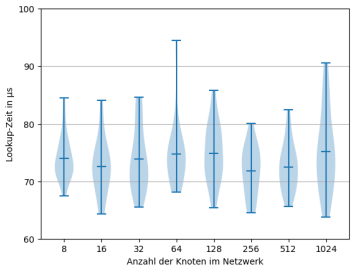
(a) Only-Lookup-Datensatz



(b) Changing-Datensatz

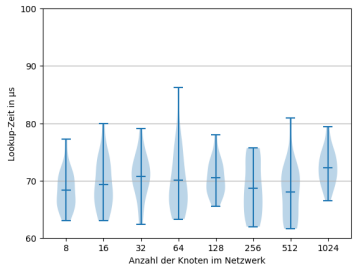


(c) Lookup-Datensatz

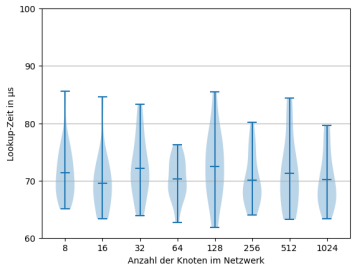


(d) Equal-Datensatz

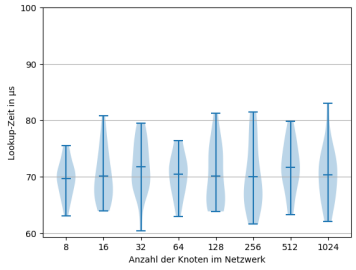
Abbildung B.1.: Auswertung der zentralisierten Variante für eine Adressraumgröße von $n = 16$ Bit



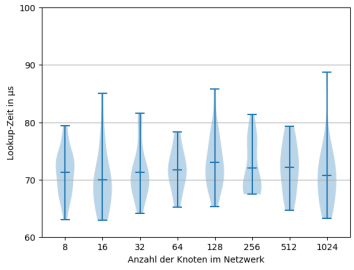
(a) Only-Lookup-Datensatz



(b) Changing-Datensatz

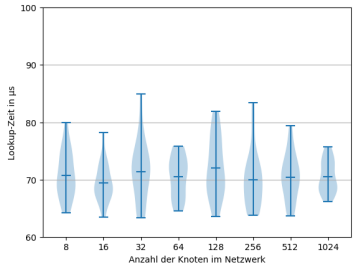


(c) Lookup-Datensatz

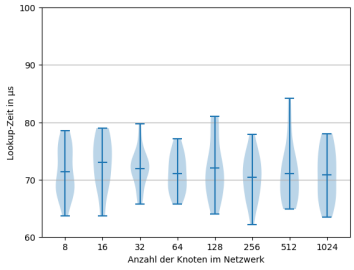


(d) Equal-Datensatz

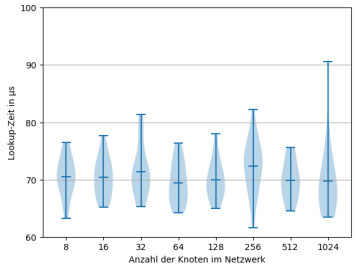
Abbildung B.2.: Auswertung der zentralisierten Variante für eine Adressraumgröße von $n = 24$ Bit



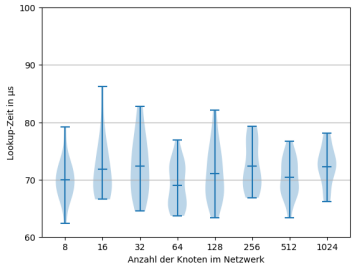
(a) Only-Lookup-Datensatz



(b) Changing-Datensatz



(c) Lookup-Datensatz



(d) Equal-Datensatz

Abbildung B.3.: Auswertung der zentralisierten Variante für eine Adressraumgröße von $n = 32$ Bit

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift