

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Masterarbeit

Adaptives Loadbalancing für strukturierte Peer-to-Peer-Netzwerke am Beispiel von Chord

Philip Wette
6382629

<wette@mail.upb.de>

Minden, den 30. August 2011

vorgelegt bei
Prof. Dr. Christian Scheideler
Dr. Kalman Graffi

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Minden, den 30. August 2011

Philip Wette

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

1 Einleitung **1**
1.1 Motivation 1
1.2 Kapitelübersicht 3

2 Grundlagen **5**
2.1 Peer-to-Peer Netzwerke 5
2.2 Unstrukturierte P2P-Overlays 8
2.3 Strukturierte P2P-Overlays 13
2.4 Chord 16
2.4.1 Bezeichner 16
2.4.2 Routingtabelle 16
2.4.3 Routingalgorithmus 17
2.4.4 Beitreten in ein Chord-Netzwerk 19
2.4.5 Stabilisationsmechanismus 21
2.4.6 Verfügbarkeit der Daten in einem Chord-Netzwerk 23
2.4.7 Effizienz von Chord 23
2.4.8 Varianten 25
2.5 Anwendungen auf strukturierten P2P-Netzwerken 26

3 Einschränkungen von Chord **29**
3.1 Selbst-Stabilisation 29
3.1.1 Selbst-Stabilisation in Chord 30
3.1.2 Re-Chord 31
3.2 Loadbalancing mit heterogenen Benutzern 41
3.2.1 Verteilungseigenschaften in P2P-Netzwerken 42
3.2.2 Dateianfragemuster 45
3.2.3 Loadbalancing in Chord 47
3.2.4 Bestehende Loadbalancing-Techniken für Chord 48

4 Adaptive-Chord: adaptives Loadbalancing für Chord **51**
4.1 Motivation . . . 51
4.2 Idee . . . 52
4.3 Entwurf . . . 55
4.3.1 P2P-Overlay zum Auffinden von freien Ressourcen . . . 55
4.3.2 Der Lastbalancierungs-Algorithmus . . . 58
4.4 Implementierung . . . 63
4.4.1 Aufbau . . . 64
4.4.2 Tie-breaker . . . 66
4.4.3 Speicherplatz . . . 68
4.4.4 Aufstellen und Pflegen von Spiegeln . . . 69

5 Evaluation **71**
5.1 Das Szenario . . . 71
5.1.1 Modell . . . 72
5.1.2 Konfigurationen der einzelnen Szenarien . . . 73
5.1.3 Parametrisierung der Algorithmen . . . 76
5.2 Kenngrößen . . . 77
5.3 Auswertung . . . 78
5.3.1 Szenarien ohne Churn . . . 78
5.3.2 Szenario busy_unstable_nochurn . . . 78
5.3.3 Szenario idle_unstable_nochurn . . . 80
5.3.4 Szenario busy_stable_nochurn . . . 81
5.3.5 Szenario idle_stable_nochurn . . . 83
5.3.6 Szenarien mit Churn . . . 84
5.3.7 Szenario busy_unstable_churn & busy_stable_churn . . . 86
5.3.8 Szenario idle_unstable_churn & idle_stable_churn . . . 88
5.4 Ergebnisse . . . 90
5.4.1 Bewertung von Re-Chord . . . 91
5.4.2 Bewertung von Adaptive-Chord . . . 92

6 Fazit & Ausblick **93**

Literaturverzeichnis **99**

Abbildungsverzeichnis

2.1 Beispielhafte Peer-to-Peer Architektur. 6
2.2 Client-Server Architektur. 6
2.3 Ein Overlay als virtuelles Netzwerk auf Basis des Internets. 7
2.4 Schematische Funktionsweise von Napster. 9
2.5 Typische Netzwerkstruktur in Gnutella 0.4. 10
2.6 Typische Netzwerk-Struktur in Gnutella 0.6. 12
2.7 Chord-Ring. 17
2.8 Finger-Einträge der Routingtabelle von Chord mit $m = 4$. 18
2.9 Einfügen eines Elementes in eine Doppelt-Verkettete-Liste. 20
2.10 Situation nach dem Beitreten eines neuen Peers. 21
2.11 Routingtabellen nach Aufruf von Stabilize() 21
2.12 Situation nach Ausfallen eines Peers. 22

3.1 Fehlerhafter Chord-Ring. 31
3.2 Das Ergebnis des Algorithmus OverlappingNeighborhood. . . . 35
3.3 Das Ergebnis von FindNextRealNeighbors. 36
3.4 Linearisierungsverfahren von Re-Chord. 37
3.5 Ring-Kanten in Re-Chord. 38
3.6 Erstellen von fehlenden Kanten im Re-Chord-Ring. 40
3.7 Verteilungsfunktion der Geschwindigkeiten im Gnutella P2P-Netzwerk 44
3.8 Verteilungsfunktion der Latenz im Gnutella P2P-Netzwerk 44
3.9 Verteilungsfunktion der Uptime im Gnutella P2P-Netzwerk 45
3.10 Zipf-Verteilung mit $p(x) \sim \frac{1}{Rang(x)}$. 46

4.1 Idee von Adaptive-Chord. 53
4.2 Weiterleitung einer Dateianfrage. 54
4.3 Capacity-Chord-Ring zum Sortieren von Peers. 56
4.4 Untere Schranken der Datenraten im Datennetzwerk. 63
4.5 Architektur der Adaptive-Chord-Referenzimplementierung. 65

5.1 Schichtenarchitektur von PeerfactSim.KOM. 72
5.2 Zeitlicher Ablauf eines Szenarios. 74

5.3 In den Simulationen genutzte Zipf-Verteilung. 75
5.4 Durchschnittliche Dauer eines Downloads. 79
5.5 Durchschnittliche Geschwindigkeit eines Downloads. 79
5.6 Durchschnittliche Erfolgsrate eines Downloads. 79
5.7 Anzahl der überlasteten Peers. 79
5.8 Durchschnittliche Dauer eines Downloads. 80
5.9 Durchschnittliche Geschwindigkeit eines Downloads. 80
5.10 Durchschnittliche Erfolgsrate eines Downloads. 81
5.11 Anzahl der überlasteten Peers. 81
5.12 Durchschnittliche Dauer eines Downloads. 81
5.13 Durchschnittliche Geschwindigkeit eines Downloads. 81
5.14 Durchschnittliche Erfolgsrate eines Downloads. 82
5.15 Anzahl der überlasteten Peers. 82
5.16 Durchschnittliche Dauer eines Downloads. 83
5.17 Durchschnittliche Geschwindigkeit eines Downloads. 83
5.18 Durchschnittliche Erfolgsrate eines Downloads. 83
5.19 Anzahl der überlasteten Peers. 83
5.20 Entwicklung der durchschnittlichen Erfolgsrate. 84
5.21 Anzahl an Peers, die aktiv am Netzwerk teilnehmen. 84
5.22 Anzahl der Service-Nachrichten. 85
5.23 Datenverkehr der Service-Nachrichten. 85
5.24 Bereinigte Entwicklung der durchschnittlichen Erfolgsrate. 86
5.25 Durchschnittliche Dauer eines Downloads. 87
5.26 Durchschnittliche Geschwindigkeit eines Downloads. 87
5.27 Durchschnittliche Erfolgsrate eines Downloads. 88
5.28 Anzahl der überlasteten Peers. 88
5.29 Durchschnittliche Dauer eines Downloads. 89
5.30 Durchschnittliche Geschwindigkeit eines Downloads. 89
5.31 Durchschnittliche Erfolgsrate eines Downloads. 89
5.32 Anzahl der überlasteten Peers. 89

1 Einleitung

1.1 Motivation

Die stetige Weiterentwicklung der Computertechnologie führt dazu, dass Heimcomputer über immer mehr Ressourcen verfügen, die oftmals im alltäglichen Gebrauch ungenutzt bleiben. Zudem stieg die Anzahl der Internetnutzer über die Jahre immer weiter an. Dies führt dazu, dass zum einen Server durch die steigenden Teilnehmerzahlen des Internets immer leistungsfähiger werden müssen, und zum anderen, dass immer mehr Kapazitäten ungenutzt bei den Nutzern der im Internet angebotenen Dienste existieren.

Um diese ungenutzten Kapazitäten nutzbar zu machen, wurde daher nach einer Alternative zur Bereitstellung von Diensten durch die Client-Server-Architektur gesucht. Eine dieser Alternativen stellt das *Peer-to-Peer-Paradigma* dar. Mit der Veröffentlichung des Programms *Napster* [MS07] im Jahre 1999 konnte erstmals im großen Stil gezeigt werden, dass es möglich ist, einen großen Teil der durch den Informationsaustausch entstehenden Last von den Nutzern eines Dienstes selbst tragen zu lassen. Dazu führte Napster das Peer-to-Peer-Paradigma ein: Ein Peer-to-Peer-Netzwerk ist ein loser Zusammenschluss von Computersystemen, die gemeinsam Ihre Ressourcen teilen und sich somit gegenseitig einen Dienst zur Verfügung stellen. Napster gehörte zu der ersten Generation von Peer-to-Peer-Netzwerken, die sich noch durch eine zentralisierte Architektur auszeichneten. So existierten in diesen Netzwerken zentrale Kontrollstrukturen, ohne deren Existenz das Netzwerk nicht funktionsfähig ist. Bereits im Jahr 2000 wurde mit *Gnutella* [Cli01] die zweite Ge-

neration von Peer-to-Peer-Netzwerken vorgestellt. Diese zweite Generation war vollkommen unabhängig von zentralen Strukturen. Da das Netzwerk durch den Ausfall von einzelnen Teilen somit weiterhin betriebsbereit bleibt, steigt die Verfügbarkeit eines solchen Netzwerkes im Vergleich zur Client-Server-Architektur an, doch kann Gnutella keine Garantien geben, dass ein vorhandenes Datum im Netzwerk auch von jedem Teilnehmer gefunden werden kann. Zusätzlich ist diese zweite Generation von Peer-to-Peer-Netzwerken ineffizient, da die Suche nach Daten im Netzwerk ein nicht unerhebliches Datenaufkommen generiert.

Aufgrund dieser Schwächen der ersten und zweiten Generation der Peer-to-Peer-Netzwerke konzentrierte sich die Forschung in den nachfolgenden Jahren darauf, Peer-to-Peer-Netzwerke zu entwickeln, die garantieren können, dass im Netzwerk existente Daten von allen Peers des Netzwerkes effektiv gefunden werden können. So wurden die sogenannten *strukturierten Peer-to-Peer-Netzwerke* entwickelt. Die grundlegende Neuerung dieser Netzwerke besteht in der festen Zuteilung von Daten zu Teilnehmern des Netzwerkes. War es in der ersten und zweiten Generation von Peer-to-Peer-Netzwerken unerheblich, wo im Netzwerk sich eine Datei befindet, so ist dies in den strukturierten Peer-to-Peer-Netzwerken eindeutig festgelegt. So werden Dateien durch eine deterministische Regel auf die Teilnehmer des Netzwerkes abgebildet. Auf Basis dieser Zuteilung von Daten zu Teilnehmern ist es möglich, Routingstrategien zu definieren, die garantieren können, dass ein Datum im Netzwerk effektiv gefunden werden kann. Zusätzlich garantieren diese Netzwerke eine sehr hohe Verfügbarkeit und können problemlos mit einer hohen Anzahl an Teilnehmern skalieren. Ein Nachteil dieser strukturierten Peer-to-Peer-Netzwerke ist jedoch, dass durch die deterministische Regel der Zuteilung von Daten auf Teilnehmer des Netzwerkes nicht mehr garantiert werden kann, dass einzelne Daten im Netzwerk gut verfügbar sind. So ist es möglich, dass eine Datei, die eine hohe Popularität besitzt, von einem Teilnehmer im Netzwerk zur Verfügung gestellt wird, der über eine langsame Netzanbindung verfügt. In der Folge ist dieser Teilnehmer mit der Anzahl der Anfragen für diese Datei überfordert und nicht mehr in der Lage, alle Anfragen in einer akzeptablen Zeit zu bedienen.

Im Rahmen dieser Arbeit wird daher ein Lastbalancierungsverfahren, *Adaptive-Chord*, entwickelt, das in der Lage ist, ein Datum in einem strukturierten Peer-to-Peer-Netzwerk von einem Teilnehmer zur Verfügung stellen zu lassen, der über genügend Ressourcen verfügt, um die Anfragen nach diesem Datum in angemessener Zeit beantworten zu können. So wird die Verfügbarkeit der Daten im Netzwerk erhöht, was sich positiv auf die Qualität des vom Netzwerk angebotenen Dienstes auswirkt. Um Adaptive-Chord zu bewerten, wird dieses durch geeignete Simulationen mit den strukturierten Peer-to-Peer-Netzwerken Chord [SMK+01] und Re-Chord [KKS11] verglichen. Re-Chord ist eine neue Chord-Variante, die die Eigenschaft der Selbst-Stabilisation besitzt. So sind die in dieser Arbeit durchgeführten Simulatio-

nen von Re-Chord die ersten unter realen Bedingungen.

1.2 Kapitelübersicht

Im zweiten Kapitel Grundlagen wird zunächst ein Überblick über die Technik der Peer-to-Peer-Netzwerke gegeben. Die *strukturierten* Netzwerke werden von den *unstrukturierten* Netzwerken abgegrenzt und es werden verschiedene Vertreter dieser Netzwerke vorgestellt. Besonderes Augenmerk wird dabei auf das strukturierte Peer-to-Peer-Netzwerk *Chord* gelegt, da dieses eine zentrale Rolle in dem in dieser Arbeit entwickelten Verfahren spielt.

Anschließend werden im dritten Kapitel Einschränkungen von Chord verschiedene Eigenschaften von Chord untersucht. So wird die Eigenschaft der *Selbst-Stabilisation* definiert und anschließend gezeigt, dass Chord diese Eigenschaft nicht besitzt. Mit *Re-Chord* wird eine Variante von Chord vorgestellt, die Chord um diese Eigenschaft erweitert. Des Weiteren wird in diesem Kapitel detailliert auf die Lastbalancierung von Chord eingegangen. So wird herausgestellt, dass die Last, die ein Teilnehmer eines Chord-Netzwerks bedienen muss, in der Regel nicht proportional zu den von dem Teilnehmer zur Verfügung gestellten Ressourcen ist. In der Folge bleiben viele der in einem Chord-Netzwerk zur Verfügung stehenden Ressourcen ungenutzt.

Im vierten Kapitel Adaptive-Chord: adaptives Loadbalancing für Chord wird ein Verfahren entwickelt, das die Qualität der Datenbereitstellung in einem strukturierten Peer-to-Peer-Netzwerk erhöht. Dafür wird zunächst beschrieben, wie ein Chord-Netzwerk modifiziert werden kann, um mit diesem die Teilnehmer des Netzwerkes anhand eines beliebigen Merkmales zu sortieren. Anschließend wird erläutert, wie mit Hilfe dieser Sortierung die Last in einem Chord-Netzwerk fair auf dessen Teilnehmer verteilt werden kann.

Das Kapitel Evaluation beschreibt zunächst das Szenario, das für die Simulationen genutzt wurde. Hierzu zählt zum einen das Modell, das für die Simulation angenommen wird, und zum anderen auch die Lastsituation, unter der das Netzwerk betrachtet wird. Anschließend werden die in diesem fünften Kapitel genutzten Kenngrößen vorgestellt, anhand derer Chord, Re-Chord und das im Rahmen dieser Arbeit entwickelte *Adaptive-Chord* miteinander vergleichen werden.

Abgeschlossen wird diese Arbeit durch das sechste Kapitel Fazit & Ausblick, in dem die Ergebnisse dieser Arbeit kurz zusammengefasst werden. Zusätzlich werden mögliche Themen weiterführender Forschung vorgestellt.

2 Grundlagen

2.1 Peer-to-Peer Netzwerke

Ein *Peer-to-Peer-Netzwerk* (abk. *P2P-Netzwerk*) ist ein *selbst organisierender*, loser Zusammenschluss von *gleichberechtigten* Computersystemen, die gemeinsam Ihre Ressourcen, wie Speicherplatz, Rechenzeit und Netzwerkbandbreite, anderen Teilnehmern des P2P-Netzwerkes zur Verfügung stellen. Dabei kommunizieren die einzelnen Teilnehmer des P2P-Netzwerkes – im weiteren *Peer* (engl. *der Gleichgestellte*) genannt – durch den Austausch von Nachrichten in einem Rechnernetz.

Das charakteristische Merkmal der Peer-to-Peer Netzwerke ist das Teilen der eigenen Ressourcen eines Peers mit den anderen Peers des Netzwerkes. Somit wird der Service, der einem Endnutzer von einem P2P-Netzwerk zur Verfügung gestellt wird, durch die anderen Endnutzer ermöglicht. Da alle Teilnehmer im Netzwerk gleichberechtigt sind, besitzt jeder Nutzer die gleichen Pflichten und Rechte im Netzwerk, daher auch der Name *Peer-to-Peer*. Damit unterscheidet sich das P2P-Paradigma (siehe Abbildung 2.1) grundlegend von dem *Client-Server*-Paradigma, in dem ein Service von einer zentralen Instanz (dem *Server*) zur Verfügung gestellt wird und dieser von Anwendern (den *Clients*) genutzt werden kann (siehe Abbildung 2.2). Die Ressourcen, die von einem solchen Dienst mit Client-Server-Architektur zur Verfügung gestellt werden, liegen somit alle zentral bei dem Server und nicht, wie bei P2P-Netzwerken üblich, dezentral bei den Peers.

Eine der zentralen Eigenschaften eines P2P-Netzwerkes ist die *Robustheit* gegenüber Ausfällen von einzelnen Peers. Diese Eigenschaft eines P2P-Netzwerkes ist sehr

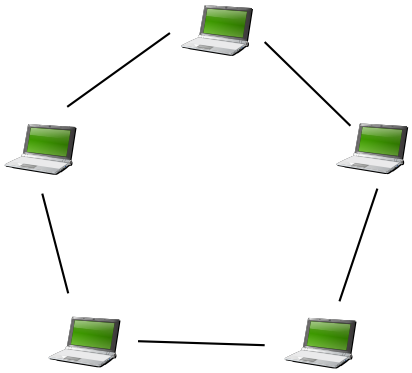


Abbildung 2.1: Beispielhafte Peer-to-Peer Architektur.

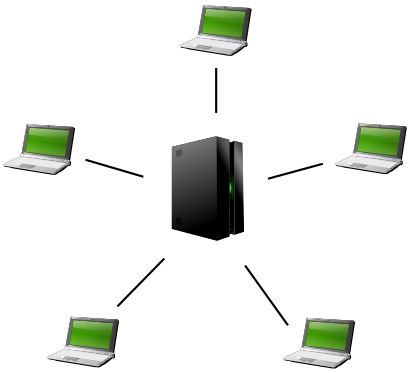


Abbildung 2.2: Client-Server Architektur.

wichtig, da P2P-Netzwerke in der Regel aus Heimcomputern bestehen, die in den meisten Fällen nur eine Zeit lang eingeschaltet sind und somit eine sehr hohe Fluktuation unter den Teilnehmern eines P2P-Netzwerkes herrscht. Da eine Ressource (wie Rechenzeit, Speicherplatz oder eine Datei) somit über die Zeit gesehen von unterschiedlichen Peers angeboten wird, ist es notwendig, dass in einem P2P-Netzwerk eine Ressource – im Kontrast zu der Client-Server Architektur – nicht durch ihre Position im Netzwerk adressiert werden kann, sondern durch einen unabhängigen Bezeichner, der aus der Ressource selbst abgeleitet werden kann. Diese Art der Adressierung wird auch *inhaltsbasierte Adressierung* genannt.

Doch der Verlust der Möglichkeit, Ressourcen über eine eindeutige Position innerhalb des P2P-Netzwerkes anzusprechen, führt zu einem gewissen Mehraufwand, wenn eine Ressource von einem Peer adressiert werden soll. Das Rechnernetz, auf dem ein P2P-Netzwerk typischerweise operiert, ist das Internet. Da die Adressierung von Ressourcen im Internet positionsabhängig ist, das P2P-Netzwerk allerdings – unabhängig von der aktuellen Position einer Ressource im Netzwerk – eine inhaltsbasierte Adressierung benötigt, ist es erforderlich, für P2P-Netzwerke eine weitere Routingschicht, zusätzlich zu der des unterliegenden Rechnernetzes, einzuführen. Dieses zusätzliche Routing arbeitet auf der 7. Schicht des ISO-OSI Referenzmodells, der Anwendungsschicht (vgl. [MS07]).

P2P-Netzwerke werden durch P2P-Overlays aufgebaut. Abbildung 2.3 zeigt ein beispielhaftes P2P-Netzwerk aufbauend auf dem Internet. Ein Overlay besteht in der Regel aus fünf Komponenten (vgl. [AAG+05]), die ein eigenes, virtuelles Netzwerk auf Basis eines Rechnernetzes, wie beispielsweise des Internets, definieren: aus einer Menge von *Bezeichern*, die benötigt werden, um eine Entität im P2P-Netzwerk eindeutig zu identifizieren, einer *Funktion*, die Ressourcen zu Peers zuordnet, einer *Routingtabelle*, die bestimmt, welcher Peer welchen anderen Peer kennt, einem *Routingalgorithmus*, der bestimmt, wie mit Hilfe der Routingtabelle ein Peer im

Netzwerk gefunden werden kann und einer *Suchfunktion*, die zu einer bestimmten Ressource den Peer bestimmt, der diese bereithält.

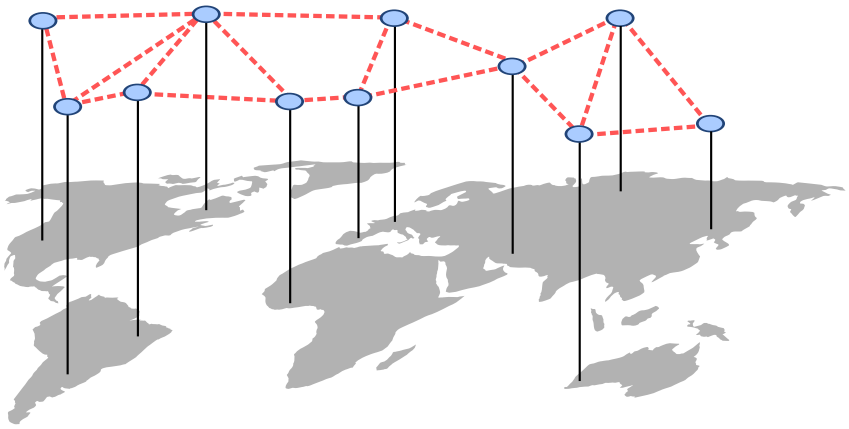


Abbildung 2.3: Ein Overlay als virtuelles Netzwerk auf Basis des Internets.

$\mathcal{I}$ bezeichnet die Menge von gültigen *Bezeichnern* eines Overlays. Diese Bezeichner bestimmen eine Entität (Ressource oder Peer) eindeutig. Um eine *Zuordnung* von Ressourcen $\mathcal{R}$ und Peers $\mathcal{P}$ in die Menge $\mathcal{I}$ zu erhalten, werden zwei Funktionen benötigt: $F_R : \mathcal{R} \to \mathcal{I}$ und $F_P : \mathcal{P} \to \mathcal{I}$. Des Weiteren existiert eine Funktion $G : \mathcal{R} \to \mathcal{P}$, die Ressourcen zu Peers zuordnet. G bestimmt somit, welcher Peer welche Ressource anbietet. Somit können alle Ressourcen und alle Peers in einem P2P-Netzwerk durch einen eindeutigen Bezeichner identifiziert werden und jeder Peer besitzt eine gewisse Menge von Ressourcen, die dieser im Netzwerk anbietet[1].

Da jedoch nicht jeder Peer alle Informationen über jeden anderen Peer besitzt, die es ermöglichen, diese im unterliegenden Rechnernetz zu kontaktieren, muss noch definiert werden, auf welchem Weg durch das Overlay ein Peer Kontakt mit einem anderen Peer aufnehmen kann. Dies wird als *Routing* bezeichnet. Um ein Routing möglich zu machen, muss zunächst definiert werden, welcher Peer im Netzwerk welchen anderen Peer kennt, d.h. weiß, wie dieser Peer im unterliegenden Rechnernetz adressiert werden kann. Somit wird die Struktur der Kanten eines Graphen auf den Peers definiert (dies wird auch *Einbettung* eines Graphen genannt). Dies geschieht über die sogenannte *Routingtabelle*. Die Routingtabelle ist eine Zuordnung von Bezeichnern des P2P-Netzwerkes zu Adressen im unterliegenden Rechnernetz. Jeder Peer besitzt eine eigene Routingtabelle. Würde z.B. jeder Peer in der Routingtabelle jeweils den Peer mit dem nächst kleineren und nächst größeren Bezeichner führen, so wäre der durch die Routingtabelle induzierte Graph eine Linie. Mit diesen lo-

[1] Im Rahmen dieser Arbeit werden als Ressourcen, die ein Peer im Netzwerk anbietet, ausschließlich Dateien betrachtet.

kalen Informationen der Routingtabelle arbeitet wiederum der *Routingalgorithmus*. Dieser bestimmt, wie ein Peer Kontakt mit einem anderen Peer durch das Overlay aufnehmen kann. Das letzte elementare Element eines P2P Overlays ist die *Suchfunktion*. Die Suchfunktion ermöglicht es, zu einer bestimmten Datei den Peer zu finden, der diese zur Verfügung stellt.

2.2 Unstrukturierte P2P-Overlays

Unstrukturierte P2P-Overlays zeichnen sich dadurch aus, dass die Zuteilung von Ressourcen zu Peers keiner klaren Struktur folgt. Jeder Teilnehmer des P2P-Netzwerkes entscheidet selbst, welche Ressourcen dieser zur Verfügung stellt. Dadurch, dass somit nicht ersichtlich ist, wo im Netzwerk eine bestimmte Ressource zu finden ist, muss in den unstrukturierten Overlays eine Suchfunktion für Ressourcen zur Verfügung gestellt werden. Diese Suchfunktion liefert zu einem Suchbegriff, der entweder aus dem genauen Namen der Ressource, Teile des Namens oder in manchen Fällen auch aus Schlagwörtern bestehen kann, eine (möglicherweise leere) Menge von Peers, die eine Ressource anbieten, die der Suchanfrage entspricht [RM06].

Je nach der Art der Suchfunktion innerhalb des Netzwerkes können die unstrukturierten P2P-Overlays in drei Unterkategorien eingeteilt werden: *zentralisierte P2P-Netzwerke*, *hybride P2P-Netzwerke* und *reine P2P-Netzwerke*.

Zentralisierte P2P-Netzwerke

Die Klasse der zentralisierten P2P-Overlays zeichnet sich dadurch aus, dass die Suchfunktion von einem zentralen Index-Server zur Verfügung gestellt wird. Somit kann ein Peer eine Suchanfrage an den zentralen Server stellen, der anschließend eine Liste mit Peers zurücksendet, die die gesuchte Ressource zur Verfügung stellen. Anschließend kann der Initiator der Suche einen dieser Peers auswählen und direkt mit diesem in Kontakt treten [ES05].

Diese Implementierung der Suchfunktion macht es notwendig, dass jeder Peer, der dem Netzwerk beitritt, eine Liste mit allen Ressourcen, die dieser zur Verfügung stellt, an den Server sendet. Dieser erstellt anschließend auf Grundlage dieser Daten einen Index, aus dem die Suchanfragen bedient werden. Somit ist die Suchfunktion in zentralisieren P2P-Overlays nach der Client-Server-Architektur gestaltet, in der die einzelnen Peers einen Service nutzen können, diesen jedoch nicht selbst zur Verfügung stellen.

Nachdem ein Peer p in zentralisierten P2P-Netzwerken über den zentralen Server ein Peer q gefunden hat, der eine gesuchte Ressource anbietet, kann p mit den Informationen, die dieser von dem Server erhalten hat, eine direkte Verbindung mit q aufbauen, um von diesem die gesuchte Ressource anzufordern. Dies ist die

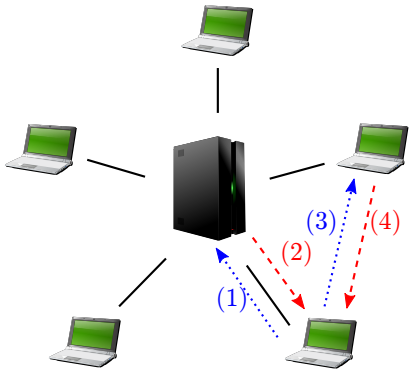


Abbildung 2.4: Schematische Funktionsweise von Napster.

eigentliche P2P-Komponente in den zentralisierten P2P-Netzwerken.

Das erste P2P-Overlay, das diese Architektur eingesetzt hat, war *Napster* [MS07]. Napster war ein zentralisiertes P2P-Overlay, das auf dem Internet operierte und wurde eingesetzt, um Multimediadateien unter den Teilnehmern des Netzwerkes zu kopieren. In Napster bestand der zentrale Server aus einem leistungsstarken Cluster von Servern, der in der Lage war, mehrere Millionen Peers gleichzeitig zu bedienen. Abbildung 2.4 zeigt schematisch, wie in Napster eine Datei lokalisiert und anschließend kopiert werden kann: Sucht ein Peer p eine Datei d, so stellt dieser zunächst eine Suchanfrage an den Server (1). Nachdem der Server die Suchanfrage erhalten hat, durchsucht dieser seinen Index und schickt die Ergebnisse zurück an p (2). Die Ergebnisse bestehen dabei unter anderem aus dem gefundenen Dateinamen und der IP-Adresse des Peers, der diese zur Verfügung stellt. Anschließend kann p eine IP-Adresse aus den Suchergebnissen auswählen und den entsprechenden Peer q zum Dateitransfer auffordern (3). In einem letzten Schritt sendet q die gesuchte Datei an p (4).

Wie an dieser Funktionsweise zu erkennen ist, bieten zentralisierte P2P-Netzwerke somit eine, nach dem Client-Server-Paradigma aufgebaute, Suchfunktion, die es ermöglicht, andere Teilnehmer zu finden, die eine bestimmte Datei zum Download anbieten. Da somit das Routing als auch das Auffinden von Ressourcen in zentralisierten Netzwerken durch eine zentrale Instanz unterstützt werden, ist das gesamte Netzwerk funktionsunfähig, sollte diese zentrale Instanz ausfallen.

Reine P2P-Netzwerke

Reicht in zentralisierten P2P-Overlays die Abschaltung des zentralen Servers, sodass das gesamte Netzwerk funktionsunfähig wird, wurden reine P2P-Overlays so konstruiert, dass diese keinen Single-Point-of-Failure besitzen. Diese Dezentralisation

hat zusätzlich auch positive Auswirkungen auf die mögliche Menge von Teilnehmern des Netzwerkes: Ist ein zentraler Server nur in der Lage eine bestimmte Anzahl von Teilnehmern zu versorgen, so können reine P2P-Overlays besser mit einer steigenden Anzahl an Benutzern skalieren.

Das erste „echte" P2P-Overlay, das so entwickelt wurde, dass es ohne Bestandteile der Client-Server Architektur auskam, war *Gnutella* 0.4 [Cli01]. Eine beispielhafte Struktur eines solchen Gnutella 0.4 Netzwerkes ist in Abbildung 2.5 zu sehen. Durch das Nichtvorhandensein eines zentralen Servers, der das gesamte Netzwerk kontrolliert, werden jedoch Prozesse, wie der Beitritt in das Netzwerk oder die Suche nach einer Datei, komplexer, da diese zusätzlich von den Peers kontrolliert werden müssen, vgl. [MS07]. Ein zentrales Konzept um diese Funktionen in Gnutella 0.4 umzusetzen, ist das sogenannte *Flodding*. Beim Flodding wird eine Nachricht sukzessive von einem Peer an alle seine Nachbarn in dem Netzwerk weitergereicht, sodass diese am Ende allen Teilnehmern, oder einer Teilmenge dieser, vorliegt. Nachrichten sind dabei mit einem speziellen Attribut, der *Time To Live (TTL)*, ausgestattet. Diese wird beim Weiterleiten einer Nachricht dekrementiert. Ist die TTL einer Nachricht 0, so wird diese nicht mehr weitergeleitet. Somit wird erreicht, dass eine Nachricht eine gewisse Verfallszeit besitzt und in der Konsequenz nur einen gewissen Teil des gesamten Netzwerkes erreicht.

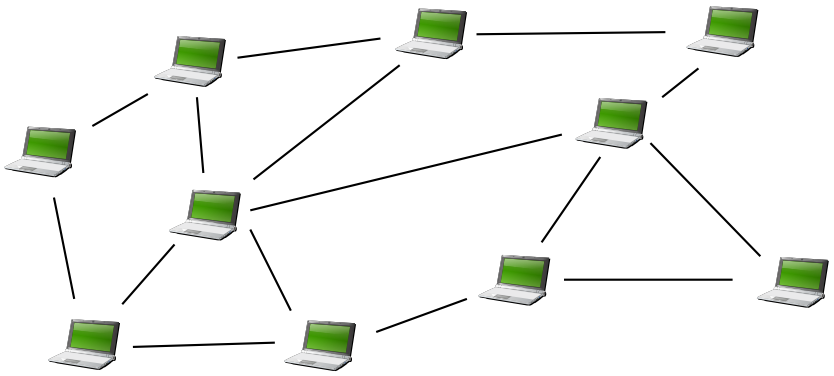


Abbildung 2.5: Typische Netzwerkstruktur in Gnutella 0.4.

Möchte ein Peer p einem Gnutella 0.4 Netzwerk beitreten, so genügt es, einen anderen Peer q zu kennen, der bereits Bestandteil des Netzwerkes ist. p schickt dazu eine sogenannte *Ping*-Nachricht an q. Die TTL der *Ping*-Nachricht zum Beitreten zu einem Netzwerk beträgt dabei 3. Empfängt q von p die *Ping*-Nachricht, dekrementiert q die TTL von 3 auf 2 und beginnt damit an alle seine Nachbarn eine *Ping*-Nachricht mit TTL 2 zu senden. Die Nachbarn von q fahren mit diesem Vorgehen fort, bis die TTL 0 erreicht und die Nachricht fallen gelassen wird. Anschließend wird jeder, der mit dieser Methode die *Ping*-Nachricht von p erhalten

hat, eine *Pong*-Nachricht an p senden. Diese *Pong*-Nachricht wird jedoch nicht auf direktem Weg zu p gesendet, sondern nimmt den gleichen Weg durch das Netzwerk, den bereits die *Ping*-Nachricht genommen hat. p lernt auf diese Weise eine Menge von aktiven Knoten kennen, aus denen er zufällig einige als seine direkten Nachbarn auswählt.

Analog funktioniert die Suche in Gnutella 0.4: Sucht p eine Datei d in dem Netzwerk, so schickt p eine *Query*-Nachricht mit einer bestimmten TTL an alle seine Nachbarn. Diese verringern die TTL um eins und senden die *Query*-Nachricht weiter an alle ihre Nachbarn. Dieser Prozess endet, wenn die TTL bei null angekommen ist und die Nachricht nicht weitergeleitet wird. Besitzt ein Peer q, der die *Query*-Nachricht erhalten hat, die Datei d, so sendet dieser eine *QueryHit*-Nachricht über den gleichen Pfad, den auch die Query-Nachricht durch das Overlay nahm, zurück zu p. Ist diese bei p angekommen, wird dieser eine direkte Verbindung zu q aufbauen und die Dateiübertragung einleiten. Dadurch, dass nach Ablauf der TTL eine Nachricht nicht weitergeleitet wird, kann Gnutella 0.4 jedoch nicht garantieren, dass eine Datei auch gefunden wird, wenn sich diese im Netzwerk befindet: Sei die TTL einer *Query*-Nachricht t. Ist die Länge eines kürzesten Pfades im Overlay-Netzwerk zu einer gesuchten Datei größer als t, so kann diese nicht aufgefunden werden, da die *Query*-Nachricht nach t Schritten auf dem Pfad fallengelassen wird.

Wie an den beschriebenen Operationen zu sehen, ist die Funktionsweise von Gnutella 0.4 vollkommen unabhängig von zentralen Kontrollstrukturen. Dies erhöht die Skalierbarkeit und zusätzlich wird das gesamte System weniger anfällig gegenüber Ausfällen von einzelnen Peers. Jedoch besitzt dieses Vorgehen auch Nachteile: So ist – im Vergleich zu der zentralisierten Verwaltung der Suchfunktion – das Nachrichtenaufkommen im Netzwerk stark gestiegen. Besonders die Effizienz der Suche hat abgenommen, was daran liegt, dass kein Knoten Informationen über die Ressourcen der anderen Knoten in dem Netzwerk besitzt. Dies macht es bei der Suche nach einer Ressource notwendig, jeden Peer einzeln zu kontaktieren. Doch die Anzahl der gesendeten Nachrichten erhöht sich noch weiter, wenn das Rechnernetz betrachtet wird, auf dem ein Gnutella-Netzwerk typischerweise betrieben wird: dem Internet. In die Auswahl der Nachbarn in Gnutella 0.4 fließen keine Informationen über die Position der einzelnen Peers im darunter liegenden Internet ein. So kann es durchaus passieren, dass die direkten Nachbarn im Gnutella-Netzwerk über die gesamte Welt verteilt sind. Somit repräsentiert ein Hop im Overlay-Netzwerk mitunter mehrere Hops im Internet; eine *Query*-Nachricht kann somit mehrfach von Kontinent zu Kontinent gesendet werden, bis diese schließlich an ihr Ziel gelangt.

Hybride P2P-Netzwerke

Die hybriden P2P-Netzwerke greifen die Ideen der reinen P2P-Netzwerke auf. Um das Nachrichtenaufkommen der Teilnehmer untereinander zu verringern, wird dabei teilweise auf das Client-Server-Modell zurückgegriffen. Um das Prinzip der hybriden P2P-Netzwerke zu erläutern, betrachten wir den Gnutella 0.6 Standard [KM02]. Aus dem vorherigen Kapitel geht hervor, dass der Preis für die Unabhängigkeit von zentralen Strukturen ein sehr hohes Nachrichtenaufkommen ist. Um dies zu reduzieren, teilt Gnutella 0.6 das Netzwerk in Teilnetze auf, die jeweils von einem besonderen Peer, dem sogenannten *Ultrapeer* bei der Suche unterstützt werden. Die „normalen" Peers, hier *Leafs* genannt, kommunizieren in Gnutella 0.6 nur mit den Ultrapeers, die wiederum mit den anderen Ultrapeers kommunizieren. Siehe hierzu Abbildung 2.6. Da Ultrapeers Verbindungen mit vielen Leafs gleichzeitig halten müssen, um deren Suchanfragen zu verwalten und Suchanfragen an diese weiterzuleiten, müssen diese über besondere Fähigkeiten verfügen: Da ein großer Teil des Nachrichtenaufkommens des Gnutella-Netzwerkes auf den Ultrapeers liegt, benötigen diese eine gute Netzwerkanbindung. Zusätzlich erfordert diese Arbeit genügend freien Speicherplatz, um Suchanfragen zwischenzuspeichern, und ausreichend Rechenkapazitäten, um Suchanfragen auszuwerten und den lokalen Zwischenspeicher zu durchsuchen.

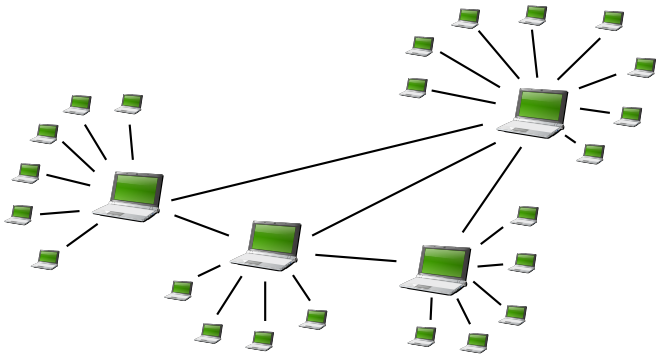


Abbildung 2.6: Typische Netzwerk-Struktur in Gnutella 0.6.

Ultrapeers agieren nicht ausschließlich als Hub für die Leaf-Peers, Ultrapeers werden auch als Cache verwendet. Dies geschieht, indem Ultrapeers die bereits durchgeführten Suchanfragen samt Ergebnissen zwischenspeichern. Bekommt ein Ultrapeer eine Suchanfrage, die vorher schon einmal gestellt wurde, so kann dieser direkt auf die Anfrage antworten. Dies führt dazu, dass das Nachrichtenaufkommen deutlich verringert wird. Durch die Aufteilung in Teilnetze ist der Durchmesser eines Gnutella 0.6-Netzwerkes deutlich kleiner als der eines Gnutella 0.4-Netzwerkes. Daraus

resultierend erreichen die Suchanfragen in Gnutella 0.6 mehr Peers, was dazu führt, dass die Wahrscheinlichkeit steigt, eine Datei zu finden. Jedoch kann auch hier keine Garantie gegeben werden, dass eine Datei gefunden wird, wenn diese im Netzwerk existiert.

2.3 Strukturierte P2P-Overlays

Sobald ein unstrukturiertes P2P-Overlay ohne einen zentralisierten Server auskommen muss, besteht das Problem, dass nicht garantiert werden kann, dass eine bestimmte Datei in vertretbarer Zeit auch von jedem Teilnehmer des P2P-Netzwerkes gefunden werden kann. Dies liegt an der Tatsache, dass in unstrukturierten Netzwerken keine Struktur in der Zuteilung von Daten zu Peers besteht. Somit müssen alle Peers durchsucht werden, um mit absoluter Sicherheit sagen zu können, ob sich eine bestimmte Datei im Netzwerk befindet oder nicht. Diese Suche konnte jedoch bislang in einem unstrukturierten Netzwerk ohne zentrale Steuereinheit nicht effizient – das heißt mit einer Komplexität signifikant besser als $\mathcal{O}(n)$, wobei n die Anzahl der Teilnehmer des Netzwerkes ist – gelöst werden.

Strukturierte P2P-Overlays hingegen kommen ohne zentralen Server aus und können zusätzlich garantieren, dass eine Datei in $\mathcal{O}(\log(n))$ Schritten gefunden wird, wenn diese im Netzwerk existiert. Dafür wird eine gewisse Struktur in der Zuordnung von Daten zu Peers benötigt. Somit besitzen Dateien in gewisser Weise eine feste Position innerhalb eines strukturierten P2P-Overlays. Dies macht es möglich, dass Dateien innerhalb des Netzwerkes adressiert werden können.

Die feste Position einer Datei im Netzwerk hat jedoch Auswirkungen auf die Art der Nutzung von strukturierten P2P-Netzwerken: Um in unstrukturierten P2P-Netzwerken eine Datei zu veröffentlichen, reicht es, diesem beizutreten und die Datei zum Download freizugeben. In einem strukturierten Netzwerk muss eine Datei hingegen aktiv publiziert werden. Das Publizieren funktioniert dabei wie folgt: Möchte Peer p eine Datei d in dem P2P-Netzwerk veröffentlichen, so berechnet p den Bezeichner h von d im Netzwerk. Anschließend kontaktiert p den Peer q, der im Netzwerk für den Bezeichner h zuständig ist und sendet diesem die Datei d. Ab diesem Zeitpunkt stellt q die Datei d unter dem Bezeichner h in dem P2P-Netzwerk zur Verfügung.

Dieses Vorgehen jedoch hat zur Konsequenz, dass ein jeder Peer eines strukturierten P2P-Netzwerkes auch für „fremde" Dateien zuständig ist, also für Dateien die dieser Peer weder erstellt noch für den eigenen Gebrauch von einem anderen Peer kopiert hat. Jeder Peer stellt somit Speicherplatz zur Verfügung, um Daten zu speichern, von denen dieser selbst keinen Nutzen hat. Es ist natürlich auch denkbar, dass p in obigem Szenario nicht die gesamte Datei an q sendet, sondern q nur darum bittet, eine Referenz für die Datei zu erstellen. Dann würde q jede Anfrage nach

d zu p weiterleiten. Im Rahmen dieser Arbeit werden strukturierte P2P-Netzwerke jedoch so betrachtet, dass die gesamte Datei an q übertragen wird und ab diesem Zeitpunkt nur q für d verantwortlich ist.

Strukturierte P2P-Overlays werden auch als *verteilte Hashtabellen* (engl. *Distributed Hash Table*, kurz: *DHT*) bezeichnet. Dies liegt daran, dass strukturierte P2P-Overlays, wie Hashtabellen, zwei Funktionen zur Verfügung stellen:

1. $store(k, d)$

2. $read(k)$

Wobei die Funktion $store(k, d)$ das Datum d unter dem Schlüssel k in der DHT ablegt. Mit $read(k)$ kann das Datum, das mit dem Schlüssel k identifiziert wird, gelesen werden.

In verteilten Hashtabellen wird der Schlüssel $k \in \{0,1\}^m$, $m \in \mathbb{N}$ zu einem Datum $d \in \{0,1\}^*$ in der Regel durch eine lokal zu berechnende, *kollisionsresistente* Hashfunktion (vgl. Definition 2.3.3) erzeugt.

Definition 2.3.1 (Hashfunktion; nach [KL07]) Eine Hashfunktion $\Pi = (Gen, H)$ ist definiert als:

Gen ist ein probabilistischer polynomialzeit Algorithmus, der bei Eingabe 1^n einen Hash-Schlüssel s berechnet, wobei n aus s rekonstruierbar ist. n wird auch als *Sicherheitsparameter* bezeichnet.

H ist ein probabilistischer polynomialzeit Algorithmus, der einen Hash-Schlüssel s und einen String $d \in \{0,1\}^*$ als Eingabe bekommt und $h \in \{0,1\}^{l(n)}$ mit $|d| \geq l(n)$ ausgibt.

1^n beschreibt hierbei die unäre Kodierung von n durch einen Bitstring bestehend aus Einsen mit Länge n.

Definition 2.3.2 (Experiment $Hash\text{-}Col_{A,\Pi}$; nach [KL07]) Das Experiment $Hash\text{-}Col_{A,\Pi}$ für einen probabilistischen polynomialzeit Algorithmus A und eine Hashfunktion Π ist definiert als:

$Hash\text{-}Col_{A,\Pi}$

---

1. Ein Schlüssel s wird generiert durch $\text{Gen}(1^n)$.
2. A bekommt als Eingabe s und produziert die Ausgabe x, x'.
3. Der Ausgang des Experimentes $Hash\text{-}Col_{A,\Pi}$ ist 1 genau dann, wenn $x \neq x'$ und $H^s(x) = H^s(x')$.

Definition 2.3.3 (kollisionsresistente Hashfunktion; nach [KL07]) Eine Hashfunktion Π ist kollisionsresistent wenn für alle A gilt:

$$Pr(Hash\text{-}Col_{A,\Pi}(m) = 1) \leq negl(m)$$

Wobei *negl* eine *vernachlässigbare Funktion* mit folgender Eigenschaft ist:
Für jedes Polynom p existiert ein $n \in \mathbb{N}$, sodass gilt:

$$negl(x) < \frac{1}{p(x)} \qquad \forall x \geq n$$

Eine vernachlässigbare Funktion strebt somit schneller gegen Null als das Inverse eines jeden Polynoms.

Eine Hashfunktion ist laut Definition 2.3.3 eine Funktion, die einen Bitstring als Eingabe erwartet und diesen auf einen kürzeren Bitstring abbildet. Definition 2.3.3 fordert von einer *kollisionsresistenten* Hashfunktion, dass es sehr unwahrscheinlich ist, dass in polynomieller Zeit zwei Bitstrings d_1, d_2 gefunden werden können, die durch Π jeweils auf den gleichen Hashwert abgebildet werden (vgl. [KL07]). Diese Forderung ist insofern wichtig, als dass es somit auch sehr unwahrscheinlich ist, dass in einer DHT zwei Dateien auf den gleichen Bezeichner abgebildet werden, somit eine Datei von einer anderen überschrieben würde.

Um in DHTs eine Zuordnung zwischen Daten und Peers zu finden, werden diese in der Regel in den gleichen Adressraum abgebildet. Anschließend können einem Peer mit Bezeichner p auf Grundlage dieser Adressen mit einer lokal zu berechnenden, deterministischen Regel Dateien zugeordnet werden. Dies könnten z.B. alle Dateien sein, deren Bezeichner nach einer gewissen Metrik in einem gewissen Abstand zu p liegen. Durch die deterministische Zuordnung von Daten zu Peers ist es in verteilten Hashtabellen somit nicht mehr nötig, eine explizite Suchfunktion zur Verfügung zu stellen, die zu dem Bezeichner einer Datei den Peer ausfindig macht, der diese zur Verfügung stellt. Die Suchfunktion wird in verteilten Hashtabellen vom Routingal-

gorithmus übernommen.

2.4 Chord

Chord [SMK+01] ist eine der strukturell einfacheren verteilten Hashtabellen und das erste P2P-Overlay, das vorgestellt wurde, das beweisbar korrekt ist und eine obere Schranke der Routingkosten von $\mathcal{O}(\log(n))$ besitzt. In der folgenden Beschreibung von Chord wird davon ausgegangen, dass Chord ein Netzwerk auf einem IP-basierten Rechnernetz aufbaut, weshalb der Begriff der IP-Adresse an einigen Stellen auftaucht. Soweit nicht anders aus dem Kontext ersichtlich, wird die Anzahl der Peers in dem Netzwerk mit n bezeichnet.

2.4.1 Bezeichner

In Chord werden die Bezeichner der Peers als auch die Bezeichner der Dateien durch die Hashfunktion *SHA-1* erzeugt. SHA-1 ist eine praktische Konstruktion einer Hashfunktion mit festem Hash-Schlüssel s und $l(n) = 160$ (vgl. Definition 2.3.1). Dadurch, dass s fest ist, entfällt das probabilistische Erzeugen, was zur Folge hat, dass jeder Peer die gleiche Hashfunktion verwendet. Nachdem bekannt wurde, dass SHA-1 einige Schwachstellen aufweist (vgl [Man11]) wird SHA-1 nicht mehr als sicher angesehen, da obwohl der Schlüssel eine Länge von 160 Bits aufweist, eine Attacke auf SHA-1 nur $\mathcal{O}(2^{51})$ Schritte benötigt (wäre SHA-1 sicher, würde dies $\mathcal{O}(2^{80})$ Schritte benötigen). Bis zum heutigen Tag wurde jedoch keine Kollision in SHA-1 bekannt.

In Chord wird der Bezeichner eines Peers generiert, indem dessen IP-Adresse gehashed wird. Analog wird der Bezeichner einer Datei als Hashwert dessen Dateinamen produziert. Dadurch, dass die Hashwerte, die von SHA-1 erzeugt werden, 160 Bit lang sind, können insgesamt maximal $2^{160} > 1,4 \cdot 10^{49}$ Peers und Dateien in einem Chord-Netzwerk adressiert werden. Mit den Bezeichnern werden Peers sowie Dateien auf einen Ring mit 2^{160} Positionen abgebildet, wobei die Bezeichner mit rechtsgerichtetem Drehsinn (Ugs. „Uhrzeigersinn") größer werden. Somit liegt insbesondere der Bezeichner 0, in dieser Darstellungsform, rechts von $2^{160} - 1$, vergleiche Abbildung 2.7. Dieser Ring wird im Weiteren als *Chord-Ring* bezeichnet.

2.4.2 Routingtabelle

Die Routingtabelle für einen Peer im Chord-Netzwerk besteht aus $\mathcal{O}(\log(n))$ Einträgen. Diese gliedern sich in den *Predecessor*, den *Successor* sowie $\mathcal{O}(\log(n))$ *Finger*. Der Predecessor (engl. Vorgänger) eines Peers p ist dabei der Peer q mit dem nächstniedrigeren Bezeichner auf dem Chord-Ring. Der Successor (engl. Nachfolger) ist der Peer mit dem nächsthöheren Bezeichner auf dem Ring.

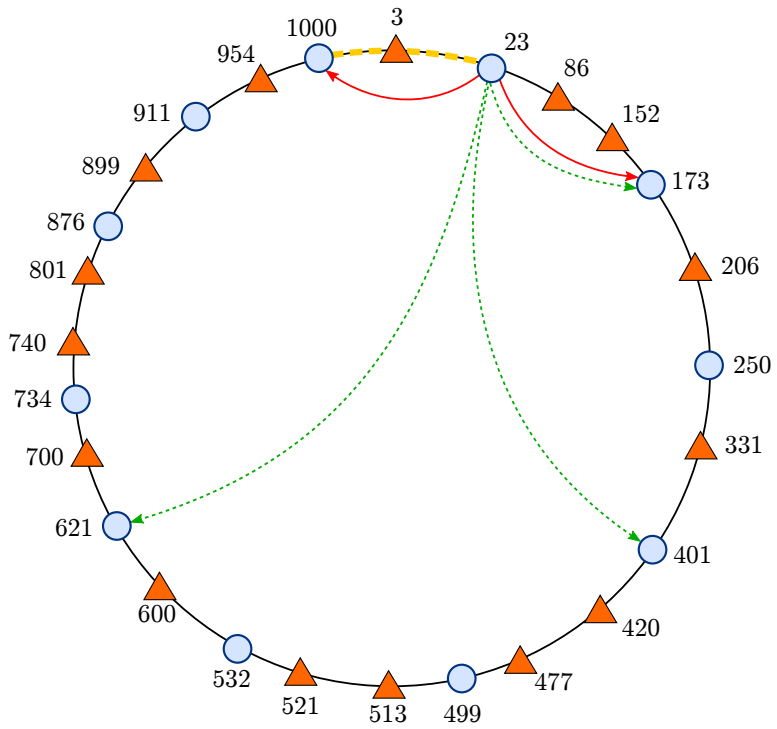


Abbildung 2.7: Chord-Ring der Größe $m = 10$. Peers als Kreise, Daten als Dreiecke. Successor- und Predecessor-Zeiger als durchgezogene Linie, Finger gestrichelt. Der gestreifte Bereich des Rings wird von Peer 23 verwaltet.

Diese beiden Einträge würden bereits ausreichen, damit das Routing in Chord korrekt funktioniert, das bedeutet, dass jeder Peer jeden anderen Peer im Netzwerk kontaktieren könnte. Jedoch würde die Route zwischen zwei Peers eine Länge von bis zu $\frac{n}{2}$ besitzen, wenn sich die Peers auf dem Chord-Ring gegenüberliegen. Um die maximale Länge einer Route im Chord-Netzwerk mit $\mathcal{O}(\log(n))$ zu beschränken, werden dazu weitere Einträge in der Routingtabelle angelegt: die sogenannten Finger. Jeder Peer besitzt bis zu 160 Finger, wobei der i-te Finger von Peer p, $p.Finger[i]$, definiert ist als Peer mit dem nächsthöheren Bezeichner zu $p + 2^{i-1}$ mod 2^{160}, $0 \leq i < 160$. Siehe hierzu Abbildung 2.8. Eine beispielhafte Routingtabelle ist in Tabelle 2.1 zu sehen. Diese zeigt die Routingtabelle von Peer 23 aus Abbildung 2.7.

2.4.3 Routingalgorithmus

Sobald von einem Peer aus auf eine Datei zugegriffen werden soll, muss der für die Datei zuständige Peer bestimmt werden. Das Auffinden dieses Peers ist die Aufgabe des Routingalgorithmus. Bei dem Routingalgorithmus von Chord handelt es sich um

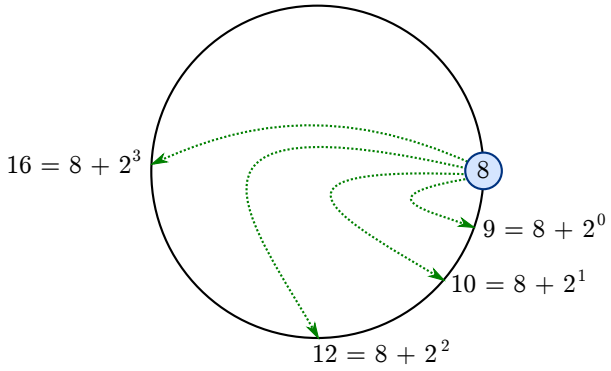


Abbildung 2.8: Finger-Einträge der Routingtabelle von Chord mit $m = 4$.

Beschreibung	**Finger ID**	**Peer ID**	**IP-Adresse**
Predecessor		1000	131.234.106.1
Successor		173	131.234.17.23
$Finger[1]$	$23 + 2^0 = 24$	173	131.234.17.23
...	...	...	...
$Finger[8]$	$23 + 2^7 = 151$	173	131.234.17.23
$Finger[9]$	$23 + 2^8 = 279$	401	131.234.13.37
$Finger[10]$	$23 + 2^9 = 535$	621	131.234.208.21

Tabelle 2.1: Routingtabelle von Peer 23 aus Abbildung 2.7.

einen Greedy-Algorithmus, der in jedem Schritt die Distanz zum Ziel um mindestens eins verringert (vgl. Satz 2.4.1).

Möchte Peer p auf eine Datei d zugreifen, so berechnet dieser zunächst den Bezeichner der Datei $H(d) = h$, wobei H die $SHA-1$ Funktion ist. Liegt h zwischen $p.predecessor$ und p, so ist p selbst für die Datei d zuständig und besitzt die Datei bereits. In diesem Fall muss der Routingalgorithmus nicht ausgeführt werden. Liegt h jedoch zwischen $p.successor$ und p, so ist $p.successor$ für h zuständig. Ist dies auch nicht der Fall, so wählt p den Eintrag in seiner Routingtabelle aus, dessen Bezeichner am nächsten an h liegt, ohne diesen zu übersteigen. Sei dies Peer p_i. p leitet seine Anfrage an p_i weiter. p_i wird wiederum nach dem gleichen Prinzip prüfen, ob die Zuständigkeit für h bei $p_i.successor$ liegt. Ist dies nicht der Fall, wird auch dieser den Eintrag aus der Routingtabelle auswählen, der am nächsten an h liegt, ohne diesen zu übersteigen. Betrachte hierzu Algorithmus 1.

Satz 2.4.1. *Algorithmus 1 terminiert und liefert den Peer zurück, der für k zuständig ist, wenn die Einträge in den Routingtabellen korrekt sind.*

Beweis. Sei v der Peer, der für k zuständig ist, und u der direkte Vorgänger von

Algorithmus 1 n.FindSuccessor(id)

```
1: if id ∈ (n, successor] then
2:     return successor
3: else
4:     n' ← n.ClosestPrecedingNode(id)
5:     return n'.FindSuccessor(id)
6: end if
```

Algorithmus 2 n.ClosestPrecedingNode(id)

```
1: for i = m downto 1 do
2:     if Finger[i] ∈ (n, id) then
3:         return Finger[i]
4:     end if
5: end for
6: return n
```

v auf dem Chord-Ring. Sei weiter n der Peer, der die Suche nach k startet. Wird Algorithmus 1 betrachtet, so existieren zwei Fälle:

Fall 1: $k \in (n, successor]$. Somit gilt $n = u$. Da v der direkte Nachfolger von u ist, gilt $n.successor = v$. Der Algorithmus terminiert und liefert den Peer zurück, der für k zuständig ist.

Fall 2: $k \notin (n, successor]$. Somit gilt $u \neq n$. In diesem Fall wird ein Peer n' durch Algorithmus 2 ausgewählt. Hier ist eine weitere Fallunterscheidung zu untersuchen.

Fall 2.1: $\forall i : n.Finger[i] \notin (n, k)$, $1 \leq i \leq 160$. Somit gilt insbesondere auch $n.Finger[1] \notin (n, k)$. Da $n.Finger[1]$ n's direkter Nachfolger ist, also $n.Finger[1] = n.successor$ und $n.Finger[1] \notin (n, k)$, muss somit entweder $k \in (n.predecessor, n]$ oder $k \in (n, n.successor]$ gelten. Da jedoch $k \notin (n, n.successor]$, muss n somit selbst für k zuständig sein. Der Algorithmus terminiert und liefert den Peer zurück, der für k zuständig ist.

Fall 2.2: $\exists i : n.Finger[i] \in (n, k)$, $1 \leq i \leq 160$. Somit hat sich der Weg zu v um mindestens eins verringert.

□

2.4.4 Beitreten in ein Chord-Netzwerk

Das Beitreten in ein Chord-Netzwerk ist vergleichbar mit dem Einfügen eines Elementes n in eine Doppelt-Verkettete-Liste: Zunächst muss die Einfüge-Position in

der Liste gefunden werden, die aus dem zukünftigen Vorgänger p und zukünftigen Nachfolger s in der Liste besteht. Anschließend kann der Nachfolger-Zeiger von p auf n und der Vorgänger-Zeiger von s auf n sowie der Vorgänger-Zeiger von n auf p und der Nachfolger-Zeiger von n auf s gesetzt werden, siehe hierzu Abbildung 2.9.

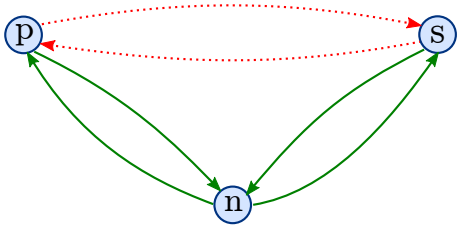


Abbildung 2.9: Einfügen eines Elementes in eine Doppelt-Verkettete-Liste. Zeiger vor dem Einfügen gepunktet, durchgezogen danach.

Soll ein Peer n dem Chord-Netzwerk beitreten, so ermittelt dieser seinen zukünftigen Nachfolger über einen Peer q, der bereits ein Bestandteil des Netzwerkes ist. q ruft FindSuccessor(n) auf, um den Peer s des Chord-Netzwerkes zu finden, der für n zuständig ist. Dies ist der Peer mit dem nächsthöheren Bezeichner zu n. Anschließend setzt n den Eintrag *Successor* seiner Routingtabelle auf s, vgl. Algorithmus 3.

Algorithmus 3 n.Join(p)

```
1: predecessor ← Null
2: successor ← p.FindSuccessor(n)
```

Damit ist der Vorgang des Beitritts in ein Chord-Netzwerk bereits abgeschlossen. Dieser Vorgang an sich wird somit nichts an den Routingtabellen anderer Peers des Netzwerkes verändern. Das bedeutet insbesondere, dass, nachdem ein Peer dem Netzwerk beigetreten ist, kein anderer Peer Informationen über diesen Peer besitzt. Dadurch, dass ein Stabilisationsmechanismus kontinuierlich im Hintergrund ausgeführt wird, wird jedoch erkannt, dass ein Peer im Netzwerk existiert, der nicht korrekt in die *Predecessor*- und *Successor*-Einträge der umliegenden Peers aufgenommen wurde. Durch den Stabilisationsmechanismus werden somit anschließend die Einträge $p.successor$ und $s.predecessor$ auf n gesetzt, sowie $n.predecessor$ auf p verwiesen.

Damit ist der neue Peer korrekt in den Chord-Ring eingegliedert, besitzt jedoch noch keinen *Finger*-Eintrag in der Routingtabelle. Diese werden ebenfalls sukzessive von dem Stabilisationsmechanismus mit der Methode FixFingers() (siehe Algorithmus 6) eingefügt.

2.4.5 Stabilisationsmechanismus

Dadurch, dass P2P-Netzwerke ein loser Zusammenschluss von vielen Teilnehmern sind, sind auch die Veränderungen, denen ein solches Netzwerk unterliegt, hoch. Neue Peers treten dem Netzwerk bei und andere Peers verlassen das Netzwerk. Durch diese Veränderungen, die auch als *Churn* bezeichnet werden, werden die Einträge in den Routingtabellen ungültig. Da zu viele ungültige Einträge in den Routingtabellen die Leistungsfähigkeit von Chord beeinträchtigen können, müssen diese von Zeit zu Zeit durch gültige Einträge ersetzt werden.

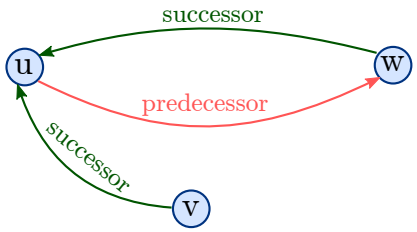


Abbildung 2.10: Situation nach dem Beitreten eines neuen Peers.

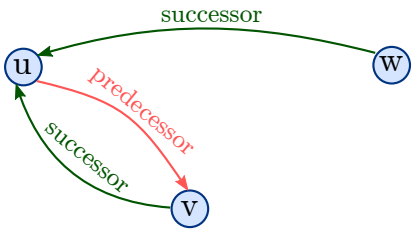


Abbildung 2.11: Einträge der Routingtabellen nachdem Peer v die STABILIZE() Methode aufgerufen hat.

Abbildung 2.10 und Abbildung 2.12 zeigen die Einträge *Predecessor* und *Successor* jeweils direkt, nachdem ein Peer dem Netzwerk beigetreten ist bzw. direkt nachdem ein Peer das Netzwerk verlassen hat. In beiden Fällen müssen die *Successor* und *Predecessor* Einträge in den Routingtabellen von Peers u und w erneuert werden, damit die Chord-typische Ring-Topologie wiederhergestellt werden kann. Zu diesem Zweck existieren in Chord drei Methoden, die periodisch im Hintergrund ausgeführt werden.

Algorithmus 4 n.STABILIZE()

```
1: x ← successor.predecessor
2: if x ∈ (n, successor) then
3:     successor ← x
4: end if
5: successor.NOTIFY(n)
```

Dies ist zum einen die in Algorithmus 4 zu sehende STABILIZE() Methode, die neue Peers in den Ring eingliedert. Diese Situation ist in Abbildung 2.10 zu sehen. Ruft v in diesem Szenario die STABILIZE() Methode auf, so wird dieser währenddessen u.NOTIFY(v) aufrufen, woraufhin u v als neuen *predecessor* in die Routingtabelle aufnimmt. Die Routingtabelleneinträge nach dieser Aktion sind in Abbildung 2.11 zu sehen. Nachdem anschließend w in dieser Situation die STABILIZE() Methode

aufgerufen hat, besitzen die Routingtabellen wieder korrekte Einträge und die Ring-Struktur ist wiederhergestellt, denn $w.successor.predecessor = v$ und $v \in (w, u)$, weshalb w in Zeile 3 des Algorithmus seinen *successor*-Eintrag auf v setzt. Anschließend ruft w auf v die NOTIFY(w) Methode auf und v setzt den *predecessor*-Eintrag auf w.

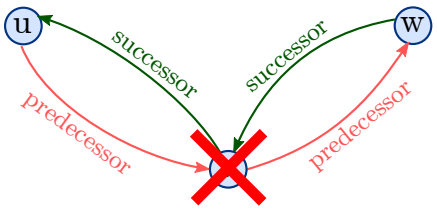


Abbildung 2.12: Situation nach Ausfallen eines Peers.

Fällt jedoch ein Peer aus, wie in Abbildung 2.12 zu sehen, so reicht die STABILIZE() Methode nicht aus, um den Ring wiederherzustellen. Dies ist darin begründet, dass in dieser Situation weder w einen Zeiger auf u noch u einen Zeiger auf w besitzt. Für einen solchen Fall hält jeder Peer mehrere *successor*-Einträge in der Routingtabelle. Diese werden mit $successor[i], 1 \leq i \leq k$ bezeichnet. Es gilt: $n.successor[1] = n.successor.successor$ und $n.successor[i+1] = n.successor[i].successor$. Somit sind $successor[i]$ die nächsten k Peers auf dem Chord-Ring. Ist der direkte Nachfolger eines Peers p nicht mehr zu erreichen, so setzt p $successor = successor[1]$, $successor[i-1] = successor[i], 2 \leq i \leq k$ und $successor[k] = successor[k-1].successor$.

Algorithmus 5 n.NOTIFY(n')

```
1: if predecessor = NULL OR n' ∈ (predecessor, n) then
2:     predecessor ← n'
3: end if
```

Algorithmus 6 n.FIXFINGERS()

```
1: next = next + 1
2: if next > m then
3:     next = 1
4: end if
5: finger[next] = FINDSUCCESSOR(n + 2^(next−1))
```

Die Methoden FIXFINGERS() und CHECKPREDECESSOR() (vgl. Algorithmus 6 und Algorithmus 7) werden ebenfalls periodisch im Hintergrund ausgeführt. Mit p.FIXFINGERS() überprüft p bei jedem Aufruf einen der *Finger*-Einträge in der

Algorithmus 7 n.CheckPredecessor()

```
1: if predecessor has failed then
2:     predecessor ← Null
3: end if
```

Routingtabelle. Dies geschieht, indem eine Suche nach dem Datum mit dem Schlüssel $p + 2^{next}$ durchgeführt wird, wobei $1 \leq next \leq m$. Das Ergebnis dieser Suche ist der Peer, der den zu $p + 2^{next-1}$ nächsthöheren Bezeichner besitzt. Somit ist anschließend der Eintrag *Finger[next]* korrekt.

Mit p.CheckPredecessor() überprüft ein Peer, ob *p.predecessor* noch Teil des Chord-Netzwerkes ist. Ist dies nicht der Fall, wird der Eintrag in der Routingtabelle als ungültig markiert, in dem dieser auf Null gesetzt wird.

2.4.6 Verfügbarkeit der Daten in einem Chord-Netzwerk

In dem in [SMK+01] vorgestellten Chord gibt es keine Garantien, dass eine Datei, die in einem Chord-Netzwerk veröffentlicht wurde, auch im Netzwerk verfügbar bleibt. Wird eine Datei beim Publizieren auf den Peer p übertragen, der für diese verantwortlich ist, so ist dies zunächst die einzige Kopie im Netzwerk, die existiert (mit Ausnahme des Peers, der die Datei publiziert hat). Fällt p aus, so verliert damit das Netzwerk ebenfalls alle Dateien, für die p zuständig war.

Soll das Netzwerk einmal publizierte Dateien nach dem Ausfall von einzelnen Peers weiterhin verfügbar halten, so gibt es keine andere Möglichkeit, als die Daten im Netzwerk zu replizieren. Fällt ein Peer p aus, so wird die Zuständigkeit für alle Daten, die dieser verwaltet hat, an den Peer mit dem zu p nächsthöheren Bezeichner abgegeben. Die Dateien können somit zum Beispiel sicher im Netzwerk verbleiben, wenn der direkte Nachfolger eines jeden Peers p Kopien von allen Daten von p besitzt. Wenn zu jedem Zeitpunkt ein Peer mit der Wahrscheinlichkeit $\frac{1}{\alpha}$, $\alpha \in \mathbb{N}$, ausfällt und bei k Kopien jeder Datei, wäre dann die Wahrscheinlichkeit, dass eine Datei aus dem Netzwerk verschwindet, $\left(\frac{1}{\alpha}\right)^k$.

2.4.7 Effizienz von Chord

Satz 2.4.1 zeigt die Korrektheit des Routingalgorithmus von Chord unter der Voraussetzung, dass jeder Peer korrekte Einträge in seiner Routingtabelle besitzt. Zusammen mit Satz 2.4.2 und Satz 2.4.3 folgt die Effizienz des von Chord eingesetzten Routingverfahrens: Es kann somit unter der Voraussetzung, dass die eingesetzte Hashfunktion die Bezeichner gleichverteilt generiert, bewiesen werden, dass das Auffinden einer Datei in Chord nur $\mathcal{O}(\log(n))$ Schritte im Overlay-Netzwerk benötigt, um eine Datei mit absoluter Sicherheit zu finden. Dies stellt eine signifikante Ver-

besserung zu den unstrukturierten P2P-Overlays dar, in denen das Auffinden einer Datei im schlechtesten Fall eine Komplexität von $\mathcal{O}(n)$ besitzt (vgl. Kapitel 2.2).

Lemma 2.4.1. *Wenn in Chord anstelle von SHA-1 eine kollisionsresistente Hashfunktion benutzt wird, dann gilt mit hoher Wahrscheinlichkeit, dass jeder Peer in Chord für maximal* $\frac{(1+\log(n))\cdot k}{n}$ *Dateien zuständig ist, wobei* n *die Anzahl der Peers und* k *die Anzahl der Dateien im Netzwerk ist.*

Für einen Beweis von Lemma 2.4.1 siehe [KLL+97] oder [Lew98]. Die Worte *„mit hoher Wahrscheinlichkeit"* bedeuten in diesem Zusammenhang, dass die Wahrscheinlichkeit kleiner ist als eine vernachlässigbare Funktion, wie diese in Definition 2.3.3 definiert wurde. Die Wahrscheinlichkeitsaussage dieses Lemmas geht dabei über die Wahl der zu benutzenden Hashfunktion aus einer Familie von Hashfunktionen (für eine Definition siehe [KL07]). Dabei wird der Beweis über die Verteilungseigenschaften der gewählten Hashfunktion geführt. An Stelle der zufälligen Wahl einer Hashfunktion wird in Chord jedoch *SHA-1* benutzt, die in der Praxis bislang gute Verteilungseigenschaften von Schlüsseln gezeigt hat, vgl. [SMK+01].

Satz 2.4.2. *Die Anzahl von Peers, die kontaktiert werden müssen, um zu einem Schlüssel den Peer mit dem nächsthöheren Bezeichner zu finden, liegt mit hoher Wahrscheinlichkeit in* $\mathcal{O}(\log(n))$*, wobei* n *die Anzahl der Peers im Chord-Netzwerk ist.*

Beweis. (nach [SMK+01]) Sei u der Peer, der nach dem Schlüssel k sucht und p der Peer, der den zu k nächstkleineren Bezeichner besitzt. Sei weiter i so gewählt, dass $p \in [u + 2^{i-1}, u + 2^i)$ gilt. Der Routingalgorithmus wird die Anfrage von u somit an den Peer $f = u.Finger[i]$ weiterleiten, wobei f der erste Peer in dem Intervall $[u + 2^{i-1}, u + 2^i)$ ist.

Da die Distanz zwischen f und p höchstens 2^{i-1} beträgt ($\left|[u + 2^{i-1}, u + 2^i)\right| \leq 2^{i-1}$) und die Distanz zwischen u und f mindestens 2^{i-1} beträgt, wird ein Hop des Routingalgorithmus bei korrekter Routingtabelle somit die Distanz mindestens halbieren. Da k höchstens 2^m von u entfernt ist, folgt eine Länge des Pfades von u zu p von maximal m.

Folgt die Verteilung von Peers und Dateien auf dem Chord-Ring der Gleichverteilung, so folgt eine Länge des Pfades von sogar nur $\mathcal{O}(\log(n))$. Denn nach $\log(n)$ Schritten ist die Anfrage bei einem Peer t, dessen maximaler Abstand zu k somit höchstens $\frac{2^m}{n}$ beträgt. Aus Lemma 2.4.1 folgt jedoch, dass die maximale Anzahl von Peers auf einem Intervall der Größe $\frac{2^m}{n}$ auf dem Chord-Ring $\mathcal{O}(\log(n))$ beträgt. Somit würde, wenn die Anfrage von t aus nur noch über die *Successor*-Zeiger weitergeleitet würde, die Route eine Länge von $\log(n) + \mathcal{O}(\log(n)) = \mathcal{O}(\log(n))$ besitzen.

□

Satz 2.4.3. *Wenn einem Chord-Netzwerk mit n Teilnehmern, von denen alle Routingtabelleneinträge korrekt sind und kein Knoten ausfällt, spontan weitere n Peers hinzugefügt werden, so beträgt die Anzahl von Peers, die kontaktiert werden müssen, um zu einem beliebigen Schlüssel den Peer mit dem nächsthöheren Bezeichner zu finden, weiterhin mit hoher Wahrscheinlichkeit* $\mathcal{O}(\log(n))$.

Beweis. (nach [SMK+01]) Sei u der Peer, der die Datei mit Bezeichner k sucht. Sei weiterhin v der erste Peer, der auf der $\mathcal{O}(\log(n))$ langen Route von u zu dem Peer w, der für k zuständig ist, bevor die n neuen Peers eingefügt wurden, liegt. Nach dem Einfügen ist u nach wie vor der nächste Knoten, den u auf der Route kontaktiert. Somit ist die Route zu w nach wie vor $\mathcal{O}(\log(n))$ lang. Da mit hoher Wahrscheinlichkeit $\mathcal{O}(\log(n))$ neue Peers zwischen zwei alten Peers liegen werden, ist somit die Route von w zu dem neuen, für k zuständigen, Peer mit hoher Wahrscheinlichkeit $\mathcal{O}(\log(n))$ lang.

□

2.4.8 Varianten

Durch das einfache Konzept von Chord bietet dieses eine gute Grundlage für Veränderungen und Erweiterungen. So wurden mit der Zeit viele solche Veränderungen an Chord vorgenommen und es entstanden speziell angepasste Chord-Varianten für viele Einsatzbereiche. *AChord* [HWW02] ist eine Chord-Variante, die die Identität der Teilnehmer verbirgt. So ist es in AChord möglich, neue Dateien im Netzwerk zu publizieren, ohne dabei die eigene Identität preiszugeben. Gleiches gilt für das Abrufen von Dateien. In einem AChord-Netzwerk ist es weder möglich Dateien zu löschen noch die Identität eines Peers festzustellen, der ein bestimmtes Dokument im Netzwerk bereitstellt. So eignet sich AChord hervorragend für das sichere unzensierte Veröffentlichen von Informationen, da jeder Teilnehmer des Netzwerkes völlig anonym ist.

Diminished Chord [KR04] ist eine weitere Variante von Chord. Diese Variante macht es möglich, durch einen Chord-Ring verschiedene Dienste gleichzeitig anbieten zu können, für die normalerweise mehrere Chord-Ringe nötig wären. Dafür werden die Peers in Diminished Chord in verschiedene Teilmengen unterteilt. Jede Teilmenge von Peers bietet anschließend einen eigenständigen Dienst an. Der Vorteil von Diminished Chord gegenüber dem Erstellen von je einem Chord-Ring für jeden Dienst ist die wesentlich geringeren Routing-Informationen, die jeder Peer besitzt, und die geringeren Kosten für die Stabilisierungsmethoden.

In [FSY05] wird *S-Chord* vorgestellt, eine Variante, die Chord robust gegenüber byzantinischen Fehlern macht. Damit wird erreicht, dass es nicht möglich ist, die Funktionsfähigkeit eines Chord-Netzwerkes durch eine Gruppe von bösartigen Peers zu beeinträchtigen.

Zusätzlich existieren ebenfalls Chord-Varianten, die nicht nur Chord erweitern oder leicht verändern, sondern versuchen, Chord mit anderen Peer-to-Peer-Overlays zu kombinieren. So verbindet *Koorde* [KK03] das Chord-Overlay mit den *De Bruijn Graphen*, um ein Peer-to-Peer-Netzwerk zu erschaffen, das die obere Grenze der Hops pro Dateianfrage mit $\mathcal{O}(\log(n))$ begrenzt, auch wenn jeder Peer nur zwei korrekte Einträge in den Routingtabellen besitzt. Besitzen die Peers in Koorde $\mathcal{O}(\log(n))$ korrekte Routingtabelleneinträge, so wird diese obere Schranke auf $\mathcal{O}(\frac{\log(n)}{\log(\log(n))})$ verbessert.

Eine andere Chord-Variante, die die Suche im Chord-Netzwerk beschleunigt, ist *EpiChord* [LLD04]. EpiChord ersetzt die Finger-Einträge der Routingtabelle durch einen Cache. Dieser Cache speichert Informationen über alle Peers, mit denen ein Peer in Kontakt getreten ist. Damit wird erreicht, dass bei wiederkehrenden Anfragen des gleichen Datums der zuständige Peer direkt bekannt ist und somit Routen zu häufig nachgefragten Dateien verkürzt werden. Zusätzlich werden mit EpiChord parallele Suchanfragen eingeführt, die dafür sorgen können, dass die Pfadlängen in Netzwerken mit hoher Auslastung eine Länge von $\mathcal{O}(1)$ betragen [LLD04].

B-Chord [WY06] erweitert die Routingtabelle um die Predecessor der jeweiligen Finger. Konkret wird somit in der Routingtabelle zu jedem Peer ebenfalls der Vorhänger dieses Peers gespeichert. Anschließend wird der Routingalgorithmus durch eine Kostenfunktion erweitert. Der so modifizierte Routingalgorithmus ist anschließend in der Lage, die Routingkosten des Netzwerkes zu verringern.

Die in [MCvR03] vorgestellte Chord-Variante ist ebenfalls bemüht, die Routingkosten in dem Netzwerk zu senken. Dafür werden in dieser Variante zusätzlich zu den Finger-Einträgen in der Routingtabelle eine weitere Menge von Fingerzeigern verwendet. Diese werden jedoch nicht mit rechtsgerichtetem, sondern mit linksgerichtetem Drehsinn angelegt. Dadurch ist es möglich, den Chord-Ring in beide Richtungen zu durchsuchen.

2.5 Anwendungen auf strukturierten P2P-Netzwerken

Strukturierte P2P-Netzwerke stellen zwei Funktionen zur Verfügung: as Speichern einer Datei unter einem Bezeichner und das Lesen einer Datei unter Angabe eines Bezeichners, vgl. Kapitel 2.3. Somit stellen strukturierte P2P-Netzwerke lediglich einen Dienst zur Verfügung, der es möglich macht, Daten zu speichern und abzurufen. Auf Grundlage dieses Dienstes ist es möglich, verschiedenste Anwendungen zu betreiben, die das P2P-Overlay beispielsweise dazu nutzen, um Anwendungsdaten zu speichern. Die einfachste und nächstliegende Anwendung von P2P-Netzwerken ist eine *Tauschbörse*. In einer Tauschbörse werden zwischen den Peers Dateien angeboten, die diese bei Bedarf kopieren können. Eine Tauschbörse ist in der Regel öffentlich, d.h. jeder, der Interesse daran hat, den Dienst der Tauschbörse zu nut-

zen, kann dieser beitreten. Da in einem strukturieren P2P-Netzwerk jeder Peer auch „fremde" Dateien zur Verfügung stellt, also Dateien, die dieser nicht selbst in das Netzwerk eingebracht oder aus eigenem Interesse kopiert hat, eignen sich strukturierte P2P-Netzwerke nicht sehr gut für diesen Anwendungsfall: Da ein Peer keine Kontrolle darüber besitzt, welche Daten dieser im Netzwerk anbietet, kann dieser somit ebenfalls Daten im Netzwerk zur Verfügung stellen, die urheberrechtlich geschützt sind. Dies würde den Besitzer dieses Peers rechtlich angreifbar machen.

Daher eignen sich strukturierte P2P-Netzwerke mehr zur Bereitstellung von Daten, die von einer nichtöffentlichen Gruppe von Personen genutzt werden, wie einem verteilten Dateisystem. Bei einem verteilten Dateisystem handelt es sich um ein Dateisystem, das sich über ein Rechnernetz zugreifen lässt. Typischerweise werden die Dateien von Servern gespeichert, die diese für eine Gruppe von Benutzern über ein Rechnernetz zum Lesen und Schreiben bereitstellen. In einem verteilten Dateisystem, das auf einem strukturierten P2P-Overlay aufbaut, werden die Server, die die Daten zur Verfügung stellen, durch das P2P-Netzwerk ersetzt. Eine mögliche Implementierung eines solchen P2P-basieren Dateisystems ist das *Chord File System (kurz: CFS)* [DKK+01]. CFS benutzt Chord als P2P-Overlay, auf dem dieses ganze Blöcke des Dateisystems speichert. Benötigt der Dateisystemtreiber von CFS somit einen bestimmten Block, so wird dieser über das Chord-Netzwerk geladen und lokal interpretiert.

Eine weitere Anwendung von strukturierten P2P-Netzwerken ist das verteilte Indexieren von Internetseiten, wie beispielsweise mit *YaCy*[2]. YaCy ist eine dezentralisierte Internet Suchmaschine, die aus fünf Komponenten besteht:

- einem Crawler, der Internetseiten besucht, um deren Informationen auszulesen
- einer Textanalyse, die Internetseiten klassifiziert und einen Index auf Grundlage der Internetseiten erstellt und wartet.
- einer lokalen Datenbank, die zu einem Schlagwort die Internetseiten auflistet, die dieses beinhalten
- einer Suchschnittstelle, die es erlaubt, nach Schlagwörtern zu suchen und die jeweiligen Ergebnisse anzuzeigen.
- einem strukturierten Peer-to-Peer-Netzwerk, das die einzelnen lokalen Datenbanken miteinander verbindet und es der Suchschnittstelle erlaubt, auch Daten zu finden, die nicht in der lokalen Datenbank gespeichert sind.

Auch soziale Netzwerke lassen sich als Anwendung auf einem P2P-Overlay betreiben. In [GGS+11] wird das soziale Netzwerk *LifeSocial.KOM* vorgestellt. Dieses

[2] http://www.yacy.net

speichert alle zum Betrieb nötigen Daten in einem strukturierten P2P-Netzwerk. Da sich in einem sozialen Netzwerk jedoch auch personenbezogene Daten befinden, werden diese vertraulichen Daten verschlüsselt auf den Peers gespeichert. LifeSocial.KOM besteht aus einer Plugin-Architektur, die sich jederzeit durch die Entwicklung weiterer Plugins erweitern lässt. Dabei bietet LifeSocial.KOM eine sichere Kommunikation zwischen den Nutzern des sozialen Netzwerkes und Zugangsberechtigungen für Dateien, wie Profile, Bilder oder Videos. Dadurch, dass LifeSocial.KOM die Daten auf den Computern der Benutzer speichert, kommt dieses soziale Netzwerk völlig ohne Server aus. Dies führt dazu, dass für die Betreiber des Netzwerkes keine Betriebskosten entstehen und das soziale Netzwerk eigenständig mit steigender Benutzeranzahl skaliert.

Dadurch, dass der Markt für Computerspiele, die über das Internet gemeinsam miteinander gespielt werden, in den letzten Jahren immer weiter gewachsen ist, sind auch die Anforderungen an die Server gestiegen, die diese Onlinespiele koordinieren. Um die Last, die diese Server bedienen müssen, zu verringern, wurden verschiedene Möglichkeiten entwickelt, um mit Hilfe von strukturierten P2P-Overlays die komplette Koordination – oder Teile dieser – auf die Computer der Teilnehmer dieser Spiele zu übertragen [BPS06, KLXH04, PaP]. Dazu werden in [BPS06] mit einem strukturierten P2P-Netzwerk die Zuständigkeiten aller Objekte, mit denen ein Spieler interagieren kann, auf die Computer der Spieler verteilt. Anschließend berechnet jeder Computer die Interaktionen der Objekte, die diesem zugeordnet wurden.

Strukturierte P2P-Netzwerke bieten somit eine Plattform für eine Vielzahl von Anwendungsbereichen, die – im Gegensatz zu der Verwendung von unstrukturierten P2P-Netzwerken – darauf vertrauen können, dass das Netzwerk robust gegenüber Ausfällen einzelner Peers ist, ein Datum sicher gefunden werden kann, wenn dieses im Netzwerk existiert (unter der Voraussetzung, dass die Peers korrekte Einträge in den Routingtabellen halten) und, dass das Netzwerk gut mit einer wachsenden Anzahl von Peers skaliert.

3 Einschränkungen von Chord

3.1 Selbst-Stabilisation

Edsgar W. Dijkstra war 1974 einer der ersten, der das Problem der sogenannten *Selbst-Stabilisation* [Dij74] formulierte. Die *Selbst-Stabilisation* ist eine spezielle Eigenschaft von verteilten Systemen ohne zentrale Steuerung. Dabei geht es um die Frage, ob ein System aus jeder initialen Konfiguration heraus in der Lage ist, einen globalen Zustand zu erreichen und anschließend in diesem zu verweilen. Ein *Zustand* ist in diesem Zusammenhang eine Eigenschaft über die Gesamtheit aller Teile des verteilten Systems.

Bei der Selbst-Stabilisation geht es vornehmlich darum, die Robustheit des Systems gegenüber Fehlern zu steigern. Besitzt ein verteiltes System die Eigenschaft der Selbst-Stabilisation, so wird dieses, unabhängig von Fehlern, wie z.B. Ausfällen von einzelnen Systemen, immer eigenständig in der Lage sein, in einen Zustand überzugehen, in dem dieses System funktionsfähig ist. Die Selbst-Stabilisation ist somit eine sehr mächtige Eigenschaft, da ein selbst-stabilisierendes verteiltes System keiner korrekten Initialisierung bedarf und kein Eingreifen in das System notwendig ist, sollten – durch Fehler oder Angriffe von außen – Teile des verteilten Systems nicht mehr erreichbar oder gar die Topologie des unterliegenden Rechnernetzes verändert worden sein (vgl. [BSW00]).

Definition 3.1.1 (Selbst-Stabilisation) Ein verteiltes System ist selbst-stabilisierend, genau dann, wenn gilt:

1. Beginnend in einem beliebigen Ausgangszustand befindet sich das System nach einer endlichen Anzahl von Zustandsübergängen in einem gültigen Zustand.
2. Ist ein gültiger Zustand erreicht und es treten keine Fehler auf, so wird jeder weitere Zustandsübergang in einem gültigen Zustand enden.

Wie in Definitinon 3.1.1 zu sehen, ist der *gültige Zustand* von der Definition nicht näher spezifiziert, somit kann die Eigenschaft, die dieser gültige Zustand besitzen soll, frei gewählt werden. Wäre für ein Chord-Netzwerk dieser gültige Zustand so gewählt, dass in diesem jeder Peer einen korrekten Successor- sowie Predecessor-Zeiger besitzt, so würde Chord im gültigen Zustand korrekt arbeiten, d.h. ein jedes Datum kann von jedem Peer des Netzwerkes gefunden werden. Insbesondere würde dies bedeuten, dass Chord robust gegenüber Ausfällen von Peers und vor allem gegenüber Änderungen in der Topologie des unterliegenden Rechnernetzes wäre.

Der gültige Zustand könnte jedoch auch so definiert werden, dass in diesem nicht nur die Successor- und Predecessor-Zeiger korrekt sind, sondern zusätzlich kein Peer überlastet ist, d.h. keine der Ressourcen wie Speicherplatz, Rechenkapazität oder Netzwerkanbindung eines Peers voll ausgelastet ist. So würde die Selbst-Stabilisation dafür sorgen, dass keine Engpässe in der Bereitstellung von Daten in dem Netzwerk existieren.

3.1.1 Selbst-Stabilisation in Chord

Sei der *gültige Zustand* von Chord definiert als Zustand, indem jeder Peer p die korrekten Successor- und Predecessor-Einträge in der Routingtablle hält, das bedeutet $p.Successor = \min\left(\{p' \in \mathcal{P} \mid p' >_{mod\ m} p\}\right)$ und $p.Predecessor = \max\left(\{p' \in \mathcal{P} \mid p' <_{mod\ m} p\}\right)$. Dann besitzt Chord die Eigenschaft der Selbst-Stabilisation *nicht*.

Obwohl Chord die in Kapitel 2.4.5 beschriebenen Routinen zur Stabilisation besitzt, ist Chord nicht in der Lage, den Chord-typischen Chord-Ring zu formen, sollte dieser einmal aufgelöst worden sein. Zwar gibt es keine Folge von beitretenden und verlassenden bzw. ausfallenden Knoten, die den Chord-Ring auflösen, wenn zwischen diesen Ereignissen die Stabilisationsroutinen ausgeführt werden (vgl. [SMK+01]), doch fordert Definitinon 3.1.1 einen beliebigen Ausgangszustand, von dem aus der gültige Zustand erreicht wird.

In Abbildung 3.1 ist ein fehlerhafter Chord-Ring abgebildet. Obwohl der unterliegende Graph zusammenhängend ist, kann der Chord-Ring durch die Stabilisationsroutinen nicht wiederhergestellt werden. Dies liegt an der Tatsache, dass die hier abgebildete Topologie zwei unabhängige Ringe zeigt. In beiden Ringen scheinen die Successor- und Predecessor-Zeiger korrekt zu sein, da für jeden Peer p gilt: $p = p.successor.predecessor$. Es ist somit nicht möglich, mit den in Chord eingesetzten Mitteln zu prüfen, ob die vorliegende Topologie tatsächlich einen Ring

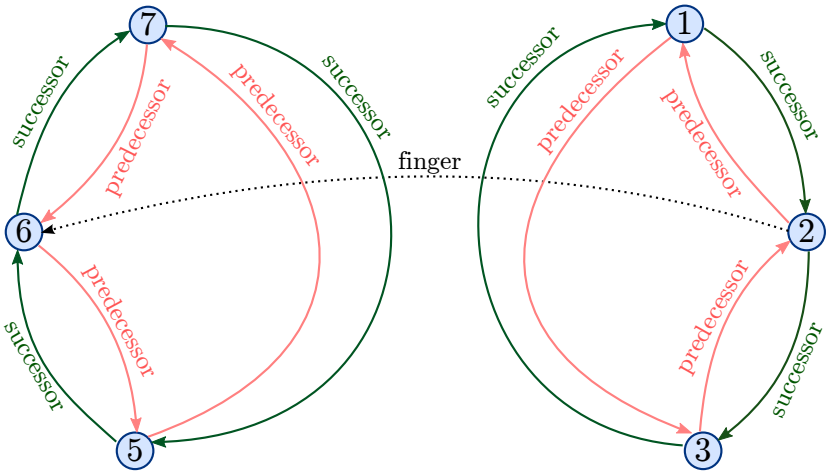


Abbildung 3.1: Fehlerhafter Chord-Ring.

beschreibt. Da somit die in Chord vorgesehenen Stabilisationsroutinen keine korrekten Successor- und Predecessor-Zeiger herstellen können, ist Chord nicht selbststabilisierend.

3.1.2 Re-Chord

Re-Chord [KKS11] erweitert Chord um ein Modul zur Selbst-Stabilisation im Sinn von korrekten Successor- und Predecessor-Zeigern. Dieses Modul ersetzt die Stabilisationsroutinen des klassischen Chord. Mit Hilfe dieser neuen Stabilisationsroutinen ist Re-Chord in der Lage, die Topologie von Chord in $\mathcal{O}(n \log(n))$ Runden aus jedem Ausgangsgraphen herzustellen, wenn dieser zusammenhängend ist.

In Re-Chord werden für diese Aufgabe anstelle der Kanten aus Chord (Successor, Predecessor und Finger) drei neue Mengen von Kanten definiert:

unmarkierte Kanten E_u, die das Pendant zu den Successor und Predecessor Kanten in Re-Chord darstellen

Verbindungskanten E_c, die für ein zusammenhängendes Re-Chord-Netzwerk sorgen

Ringkanten E_r, die dafür benutzt werden, um den Chord-Ring zwischen dem Peer mit dem niedrigsten und dem Peer mit dem höchsten Bezeichner zu schließen

Zusätzlich zu den Peers im Chord-Netzwerk führt Re-Chord außerdem für jeden realen Peer $p \in V_r$ eine Menge von *virtuellen Peers* $p_1, ..., p_m \in V_v$ ein. Virtuelle Peers werden dabei von realen Peers simuliert, d.h. virtuelle Peers verhalten sich im Netzwerk wie reale Peers, jedoch besitzt der Wirt eines virtuellen Peers direkten

Zugriff auf alle Daten des virtuellen Peers. Insbesondere können so virtuelle Peers, die zu dem selben realen Peer gehören, gegenseitig direkt auf ihre internen Daten zugreifen, ohne dabei Kommunikationskosten zu erzeugen. Die Menge der *Geschwister* $S(p_i)$ eines Peers p_i wird definiert als $S(p_i) = \{p_0, ..., p_m\}$. Jeder reale Peer p kann bis zu m virtuelle Peers besitzen, wobei der Bezeichner eines jeden virtuellen Peers p_i als $p + 2^{m-i} \mod 2^m$ definiert ist. Da $p_0 = p$ gilt, wird p im Weiteren auch als p_0 bezeichnet. Somit handelt es sich bei p_0 um den realen Peer p, nicht um einen virtuellen. Ein realer Peer kann zu jeder Zeit neue virtuelle Peers erstellen oder bereits bestehende virtuelle Peers aus dem Netzwerk entfernen, sollte dies notwendig sein.

Sei $V = V_r \cup V_v$ die Menge aller Peers eines Re-Chord-Netzwerkes sowie $N_u(p_i) = \{v \in V \mid (p_i, v) \in E_u\}$ die unmarkierte Nachbarschaft eines Peers p_i und analog $N_r(p_i) = \{v \in V \mid (p_i, v) \in E_r\}$ sowie $N_c(p_i) = \{v \in V \mid (p_i, v) \in E_c\}$ als die Ring-Nachbarschaft sowie die Verbindungs-Nachbarschaft des Peers p_i definiert. Dann ist die Nachbarschaft $N(p_i)$ eines Peers p_i definiert als:

$$N(p_i) = S(p_i) \cup \bigcup_{j=0}^{m} N_u(p_j)$$

Somit liegen alle Peers in der Nachbarschaft $N(p_i)$, die über unmarkierte Kanten von den Geschwistern von p_i aus in einem Schritt erreicht werden können.

Im stabilen Zustand von Re-Chord besitzt jeder Peer $p \in V$ unmarkierte Kanten zu den Peers mit dem jeweils nächsthöheren und nächstniedrigeren Bezeichner im Netzwerk als auch zu den realen Peers mit dem jeweils nächsthöheren und nächstniedrigeren Bezeichner (vgl. [KKS11]). Somit können die Predecessor-, Successor- sowie Finger-Kanten von Chord mit Hilfe der unmarkierten Kanten sowie der virtuellen Peers ausgedrückt werden:

- $p.Predecessor := \min\left(\{v \in V_r | (p, v) \in E_u\}\right)$
- $p.Successor := \max\left(\{v \in V_r | (p, v) \in E_u\}\right)$
- $p.Finger[i] := p_{m-i}.Successor$

Somit ist das Chord-Netzwerk im stabilen Zustand eines Re-Chord-Netzwerkes enthalten.

Der Stabilisationsmechanismus

Ziel des Stabilisationsmechanismus von Re-Chord ist das Bilden eines Ringes, in dem jeder Peer des Netzwerkes eine Kante auf den Peer mit dem nächsthöheren und auf

den Peer mit dem nächstniedrigeren Bezeichner besitzt. Dieser Stabilisationsmechanismus gliedert sich in sechs Methoden, die jeder Peer des Re-Chord-Netzwerkes sequenziell ausführt. Dabei wird davon ausgegangen, dass die Zeit rundenbasiert verläuft, d.h. zu Beginn jeder Runde startet jeder Peer den Stabilisationsmechanismus. Ist eine Runde beendet, so ist auch der Stabilisationsmechanismus eines jeden Peers abgeschlossen. Während dieser sechs Methoden kann ein Peer p eine Kante zwischen zwei Peers u, v erstellen, zu denen dieser ausgehende Kanten besitzt. Dazu sendet p eine *Verbindungsnachricht* an u. Diese Nachricht besteht aus dem Typ der zu erstellenden Kante (unmarkierte Kante, Verbindungskante oder Ringkante) und dem Peer, zu dem die neue Kante erstellt werden soll (v). Im Weiteren wird eine solche Nachricht mit $VN(u, v, t)$ mit $t \in \{u, c, r\}$ bezeichnet, wobei u für eine unmarkierte Kante, c für eine Verbindungskante und r für eine Ringkante steht. Verbindungsnachrichten erstellen nur gerichtete Kanten, das bedeutet, dass eine Verbindungsnachricht $VN(u, v, t)$ bewirkt, dass Peer u von der Existenz des Peers v unterrichtet wird und zu diesem eine Kante mit dem Typ t erstellt. Der Peer v jedoch bleibt von dieser Nachricht vollkommen unberührt. Wird eine Verbindungsnachricht $VN(u, v, \cdot)$ in Runde t versandt, so existiert die Kante (u, v) mit Beginn der Runde $t + 1$. Diese neue Kante hat somit keinen Einfluss auf den Ablauf von Runde t.

Der Stabilisationsmechanismus besteht aus den folgenden sechs Methoden:

1. Virtuelle Knoten verwalten
2. Nachbarschaftsüberschneidungen auflösen
3. Nächste reale Nachbarn finden
4. Linearisierung
5. Ringkante einfügen
6. Verbindungskanten einfügen

Diese werden periodisch von jedem Peer ausgeführt und garantieren, dass sich das Re-Chord-Netzwerk nach $\mathcal{O}(n \cdot \log(n))$ Runden in einem stabilen Zustand befindet, treten keine Fehler wie z.B. Ausfälle von Systemen auf (vgl. [KKS11]).

Virtuelle Knoten verwalten

Je mehr Teilnehmer sich in einem Re-Chord-Netzwerk befinden, desto länger benötigt dieses, um in den stabilen Zustand überzugehen. Wie bereits erwähnt, besitzt

jeder reale Peer eine Menge von virtuellen Peers, die ebenfalls in dem Re-Chord-Netzwerk teilnehmen. Daher wird zu Beginn einer jeden Runde überprüft, ob virtuelle Peers existieren, die entfernt werden können, da diese sonst die Laufzeit des Verfahrens erhöhen würden.

Sind zwei Peers p_i und p_{i-1} des gleichen Wirts direkte Nachbarn auf dem Re-Chord-Ring, so können diese dem Wirt nicht mehr Informationen beschaffen, als ein einziger Peer p_i in dem selben Abschnitt des Rings. Somit muss, um Redundanzen zu vermeiden, ein anderer Peer v_j mit $v \neq p$ existieren, dessen Bezeichner sich zwischen p und dem ersten virtuellen Peer von p befindet. Andernfalls wäre die Nachbarschaft von p und dessen erstem virtuellem Peer gleich.

Algorithmus 8 p.ManageVirtualPeers()

```
1:  oldS ← s
2:  s ← m − 1                                   ▷ calculate id of closest virtual sibling
3:  while p + 2^(m−s) mod 2^m < p.Successor and s > 1 do
4:      s − −
5:  end while
6:  if oldS > s then                             ▷ delete needless virtual siblings
7:      for i = s + 1 to oldS do
8:          N_u(p_s) = N_u(p_s) ∪ N_u(p_i) ∪ N_r(p_i) ∪ N_c(p_i)      ▷ save neighborhood
9:          delete p_i
10:     end for
11: end if
12: if oldS < s then                             ▷ create new virtual siblings
13:     for i = oldS + 1 to s do
14:         create p_i
15:     end for
16: end if
```

Algorithmus 8 zeigt, wie virtuelle Peers verwaltet werden: Besitzt ein realer Peer p virtuelle Knoten, deren Bezeichner zwischen p und $p.Successor$ liegen, so werden diese gelöscht. Damit keine Informationen über die Nachbarschaft dieser gelöschten Peers verloren gehen, werden alle von diesen Peers ausgehenden Kanten als unmarkierte Kanten an den virtuellen Peer von p mit dem kleinsten Bezeichner übertragen.

Nachbarschaftsüberschneidungen auflösen

Jeder Peer soll im stabilen Zustand von Re-Chord ausgehende, unmarkierte Kanten zu den Peers mit dem jeweils nächsthöheren und nächstniedrigeren Bezeichner im Netzwerk als auch zu den realen Peers mit dem jeweils nächsthöheren und nächstniedrigeren Bezeichner besitzen. Existiert ein realer Peer $p \in V_r$, der zwei virtuelle

Peers p_i und p_j besitzt, sowie ein Peer $u \in V$ mit $u > p_i > p_j$ oder $u < p_i < p_j$ existiert, sodass p_j eine Kante zu u besitzt, dann muss diese nach p_i verschoben werden, da p_i näher an u liegt als p_j. Diese Situation ist in Abbildung 3.2 (a) zu sehen. Abbildung 3.2 (b) zeigt das Ergebnis des Algorithmus 9, der diese Nachbarschaftsüberschneidungen auflöst. Diese Operation sorgt somit dafür, dass Informationen über Nachbarn zwischen den virtuellen Peers des gleichen Wirts geteilt werden: p_j erkennt, dass u in dieser Situation nicht dessen direkter Nachbar ist, und gibt diese Information so an p_i weiter.

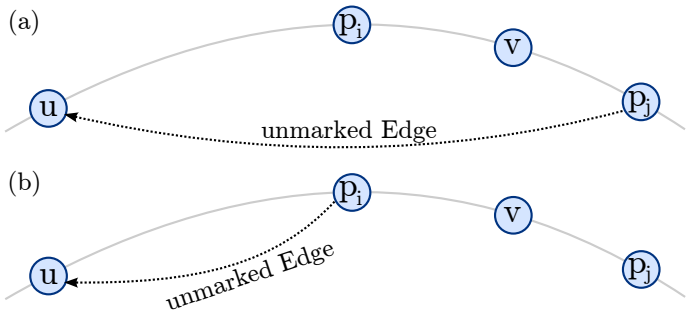


Abbildung 3.2: (a) Die überlappende Nachbarschaft zwischen zwei virtuellen Peers vom gleichen Wirt. (b) Das Ergebnis des Algorithmus OverlappingNeighborhood.

Algorithmus 9 p.OverlappingNeighborhood()

```
1: for each (p_i, p_j) ∈ S(p)^2 do                ▷ for each pair of siblings
2:     if ∃u ∈ N_u(p_i): p_j ∈ [u, p_i] then
3:         N_u(p_i) = N_u(p_i) \ {u}
4:         N_u(p_j) = N_u(p_j) ∪ {u}
5:     end if
6: end for
```

Nächste reale Nachbarn finden

Die nächsten realen Nachbarn in Re-Chord sind definiert als $r_l(p_i)$ und $r_r(p_i)$, wobei

$$r_l(p_i) := \max\Big(\big\{w \in N(p_i) \mid w \in V_r \wedge w < p_i\big\}\Big)$$

$$r_r(p_i) := \min\Big(\big\{w \in N(p_i) \mid w \in V_r \wedge w > p_i\big\}\Big)$$

somit ist $r_l(p_i)$ der nächste reale Peer links von u_i und $r_r(p_i)$ der nächste reale Peer rechts zu p_i auf dem Re-Chord-Ring.

Dadurch, dass das Re-Chord Netzwerk ständigen Veränderungen unterliegt, können sich ebenfalls die nächsten rechten sowie linken realen Nachbarn eines jeden Peers ändern. Somit ist es notwendig, in jeder Runde diese direkten Nachbarn neu zu bestimmen. Nachdem die direkten realen Nachbarn $r_l(p_i)$ und $r_r(p_i)$ eines Peers p_i bestimmt wurden, sendet dieser die Informationen über diese beiden Peers an alle seine Nachbarn, deren Bezeichner in dem Intervall $[r_l(p_i), r_l(p_i)]$ liegen. Damit wird erreicht, dass, wenn für Peer p_i die Kanten auf $r_l(p_i)$ und $r_r(p_i)$ korrekt sind, d.h. auf p_i's echte direkte reale Nachbarn verweisen, auch alle Peers, deren Bezeichner in dem Intervall $[r_l(p_i), r_l(p_i)]$ liegen und zu denen p_i eine Kante besitzt, in der nächsten Runde unmarkierte Kanten auf deren korrekte linke und rechte reale Nachbarn besitzen. In der Folge werden diese in der nächsten Runde ebenfalls ihre korrekten linken und rechten realen Nachbarn benennen können.

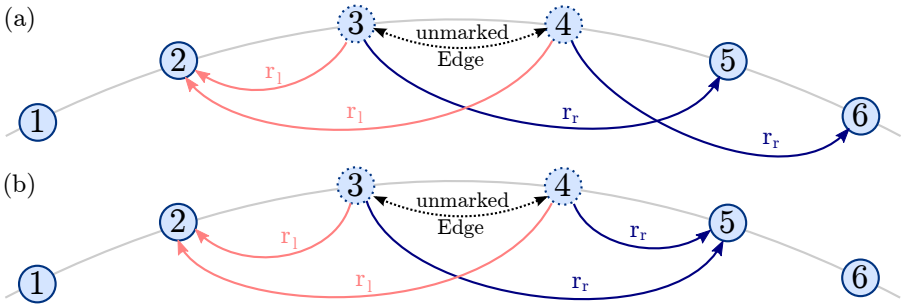


Abbildung 3.3: Reale Peers durchgezogen; virtuelle gestrichelt. (a) Die falschen r_l- und r_r-Kanten von zwei virtuellen Peers. (b) Das Ergebnis von FindNextRealNeighbors.

Algorithmus 10 p.FindNextRealNeighbors()

1: $p.r_l \leftarrow \max\left(\{w \in N(p) \mid w \in V_r \wedge w < p\}\right)$ ▷ left neighbor
2: $p.r_r \leftarrow \min\left(\{w \in N(p) \mid w \in V_r \wedge w > p\}\right)$ ▷ right neighbor
3: $N_u(p) = N_u(p) \cup \{p.r_l\} \cup \{p.r_r\}$ ▷ create unmarked edges to $p.r_l$ and $p.r_r$
4: **for** $v \in N(p) : v \in [p.r_l, p.r_r]$ **do** ▷ inform other peers about $p.r_l$ and $p.r_r$
5: **send** $VN(v, p.r_l, u)$
6: **send** $VN(v, p.r_r, u)$
7: **end for**

Abbildung 3.3 zeigt, wie mit Algorithmus 10 die nächsten realen Nachbarn gefunden werden und zusätzlich alle Peers zwischen diesen realen Nachbarn über deren Existenz informiert werden. Bevor Algorithmus 10 ausgeführt wurde, war die r_r-Kante von Peer 4 falsch, da diese auf Peer 6 und nicht auf Peer 5 zeigt. Dadurch, dass Peer 3 alle Peers zwischen seinem rechten und linken realen Nachbarn mit Hilfe

von unmarkierten Kanten über deren Existenz berichtet, konnte auch Peer 4 eine Kante auf seinen korrekten rechten realen Nachbarn erzeugen.

Linearisierung

Ziel der Linearisierung ist das Transformieren des Graphen in eine Linie, bei der jeder Knoten im Graph eine Kante zu seinem nächsten linken und nächsten rechten Nachbarn besitzt. Hierzu sortiert jeder Peer p_i seine Nachbarn $W_g = \{w_0, w_1, ..., w_k\} = \{w \in N_u(p_i) \mid w > p_i\}$ absteigend nach deren Bezeichner und erzeugt Kanten zwischen w_l und w_{l+1}, $0 \leq l < k$. Analog sortiert p_i seine Nachbarn $W_k = \{w \in N_u \mid w < p_i\}$ aufsteigend und erzeugt ebenfalls Kanten zwischen w_l und w_{l+1}. Da w_{i+1} kein direkter Nachbar von p_i ist, wird, nachdem eine unmarkierte Kante zwischen w_l und w_{l+1} erzeugt wurde, die unmarkierte Kante (p_i, w_{k+1}) von p_i gelöscht. Eine exemplarische Anwendung dieses Verfahrens ist in Abbildung 3.4 (a) und (b) dargestellt. Wie hier gut zu sehen ist, bildet der ungerichtete Teilgraph bestehend aus den Knoten $N_u(p_i)$ und den Kanten $\{(p_i, v) \mid v \in N_u(p_i)\}$ nach der Linearisierung mit den neu entstandenen Kanten eine Linie.

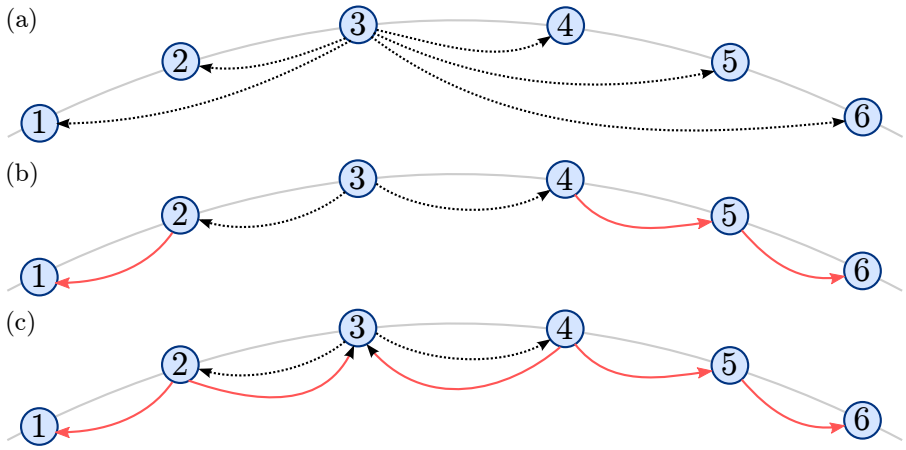


Abbildung 3.4: Linearisierungsverfahren von Re-Chord: unmarkierte Kanten gestrichelt; Kanten, die durch Verbindungsnachrichten entstehen, durchgezogen. (a) Ausgangsgraph. (b) Graph nach dem Linearisierungsverfahren. (c) Graph, nach dem die verbleibenden Kanten gespiegelt wurden.

Nachdem die Linearisierung durchgeführt wurde, besitzt jeder Peer nur noch zwei ausgehende unmarkierte Kanten. Diese Kanten werden anschließend gespiegelt. Das bedeutet, zu jeder unmarkierten Kante (u, v) wird auch eine Kante (v, u) eingefügt. Dies geschieht, indem der Peer p_i Verbindungsnachrichten zu seinen verbleibenden zwei, über unmarkierte Kanten zu erreichenden, Nachbarn sendet. Das Ergebnis dieser Operation ist in Abbildung 3.4 (c) veranschaulicht.

Es ist zu beachten, dass bei dieser Operation nicht zwischen realen und virtuellen Peers unterschieden wird. Das bedeutet, dass die verbleibenden direkten Nachbarn virtuelle und reale Peers sein können. Des Weiteren bleiben die in der vorangegangen Operation gespeicherten direkten realen Nachbarn, r_l und r_r, von dieser Operation unberührt. r_l und r_r können somit als spezielle Kanten betrachtet werden, die unabhängig von unmarkierten Kanten existieren und daher von der Linearisierung nicht verändert werden.

Ringkante einfügen

Da die Linearisierung zu einer Linie führt, die typische Topologie von Chord jedoch der Chord-Ring ist, wird durch diese Phase der Stabilisierung eine Kante zwischen dem Peer p_{min} mit dem kleinsten und dem Peer p_{max} mit dem größten Bezeichner eingefügt. Diese Kante schließt den Ring und wird daher *Ring-Kante* genannt. Im stabilen Zustand von Re-Chord führt eine Ring-Kante von p_{min} zu p_{max} sowie von p_{max} zu p_{min}.

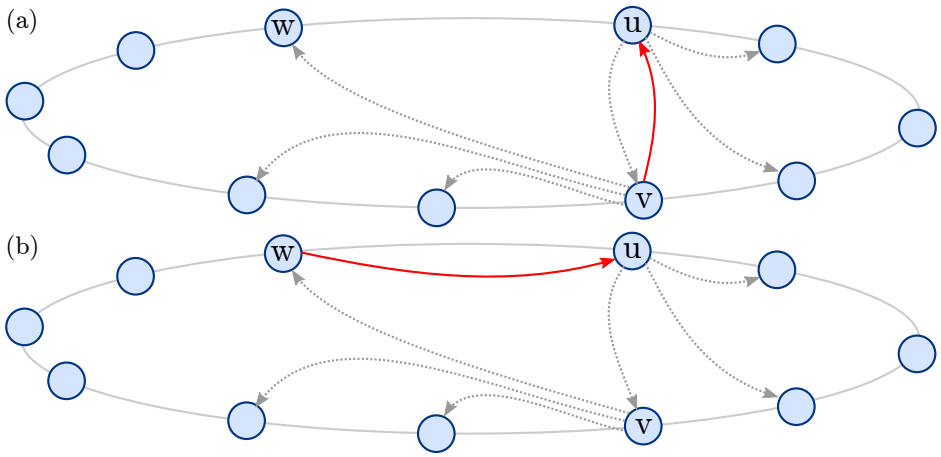


Abbildung 3.5: Ring-Kanten in Re-Chord. Ausgehende Kanten von u und v gestrichelt; Ring-Kanten durchgezogen. (a) u besitzt keinen linken Nachbarn und erzeugt daraufhin eine Ring-Kante zu $v = \max(N(u))$. (b) v besitzt eine Ring-Kante zu u. Da $u < v$, leitet v diese Kante an $w = \max(N(v))$ weiter. Da $\max(N(w)) \leq w$, somit w keinen Peer mit höherem Bezeichner kennt, bleibt die Ringkante zwischen w und u bestehen.

Besitzt nach der Linearisierung ein Peer p_i keine unmarkierte Kante zu einem Peer mit einem kleineren Bezeichner, so ist – aus der lokalen Sicht von p_i – p_i der Peer mit dem kleinsten Bezeichner im Netzwerk. Daher ermittelt p_i den Peer v aus seiner Nachbarschaft $N(p_i)$, der den größten Bezeichner besitzt. Anschließend erzeugt p_i mit einer Verbindungsnachricht $VN(v, p_i, r)$ eine Ring-Kante, die von v

nach p_i verläuft. Diese Ring-Kante wird sukzessive in den nachfolgenden Runden so lange von dem aktuellen Peer v an einen Peer mit größerem Bezeichner aus v's Nachbarschaft weitergeleitet, bis diese Kante bei einem Peer angelangt, der keine ausgehende Kante zu einem Peer mit größerem Bezeichner besitzt. Dieses Verhalten ist in Algorithmus 11 – auch für den analogen Fall mit einer Ring-Kante zu einem Peer mit höherem Bezeichner – beschrieben.

Abbildung 3.5 zeigt wie Algorithmus 11 arbeitet: u bemerkt in Runde t, dass dieser keinen linken Nachbarn besitzt. Daraufhin erstellt u die Ring-Kante (v, u), wobei $v = \max(N(u))$, mit Hilfe einer Verbindungsnachricht. In Runde $t+1$ existiert die Ring-Kante (v, u), woraufhin v ermittelt, dass $u < v$ gilt. Da v einen rechten Nachbarn besitzt, kann jedoch v nicht das eigentliche Ziel der Ring-Kante sein, sodass v diese Kante an w weiterleitet. Dazu berechnet v $w = \max(N(v))$ und sendet diesem die Verbindungsnachricht $VN(w, u, r)$. Anschließend löscht v die Ring-Kante (v, u). In Runde $t+2$ besitzt w eine Ring-Kante zu u. Da w keinen rechten Nachbarn hat, bleibt diese so lange zwischen w und v bestehen, bis w einen Nachbarn mit höherem Bezeichner findet.

Algorithmus 11 p.ManageRingEdges()

```
1:  if ∄v ∈ N(p) : v < p then                ▷ missing left neighbor: create ring-edge
2:      p_max ← max(N(p))
3:      send VN(p_max, p, r)
4:  end if
5:  if ∄v ∈ N(p) : v > p then                ▷ missing right neighbor: create ring-edge
6:      p_min ← min(N(p))
7:      send VN(p_min, p, r)
8:  end if
9:  for v ∈ N_u(p) do
10:     if v < p and ∃u ∈ N(p) : u > p then   ▷ forward ring-edge to higher id
11:         p_max ← max(N(p))
12:         send VN(p_max, v, r)              ▷ create new ring-edge
13:         N_u(p) = N_u(p) \ {v}             ▷ delete my ring-edge
14:     end if
15:     if v > p and ∃u ∈ N(p) : u < p then   ▷ forward ring-edge to lower id
16:         p_min ← min(N(p))
17:         send VN(p_min, v, r)              ▷ create new ring-edge
18:         N_u(p) = N_u(p) \ {v}             ▷ delete my ring-edge
19:     end if
20: end for
```

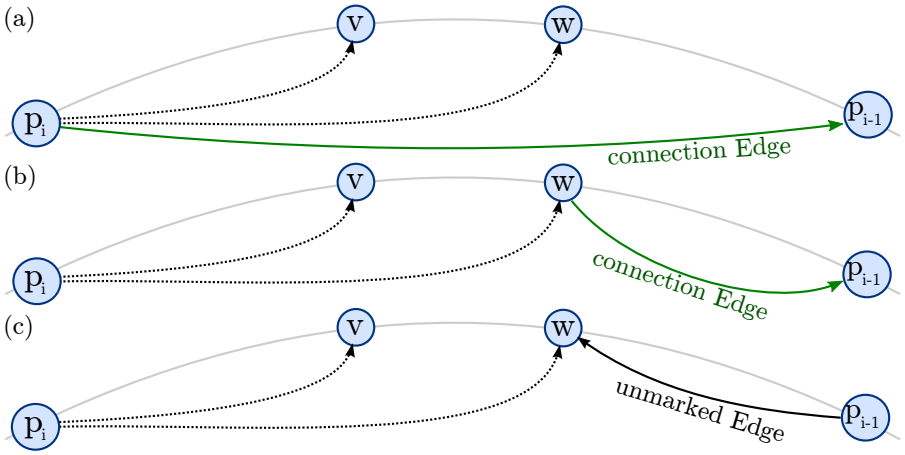


Abbildung 3.6: Erstellen von fehlenden Kanten im Re-Chord-Ring.

Verbindungskanten einfügen

Wie unmarkierte- und Ring-Kanten, existieren in Re-Chord ebenfalls die sogenannten Verbindungskanten. Die Verbindungskanten erfüllen dabei einen ähnlichen Zweck wie die Ring-Kanten, nur dass Verbindungskanten nicht zwischen dem Peer mit dem höchsten und dem niedrigsten Bezeichner des Netzwerkes platziert werden, sondern für jeden Peer p_i zwischen $w = \max\left(\{v \in N(p_i) \mid v < p_{i-1}\}\right)$ und p_{i-1}, also zwischen dem virtuellen Zwilling mit dem nächsthöheren Bezeichner (p_{i-1}) und dem Peer, der aus Sicht von p_i am nächsten an p_{i-1} liegt, ohne dessen Bezeichner zu übersteigen.

Ähnlich zu den Ring-Kanten werden die Verbindungskanten zwischen den Peers im Intervall $[p_i, p_{i-1}]$ anschließend so lange weitergeleitet, bis diese bei einem Peer w angelangen, der keinen Nachbarn besitzt, dessen Bezeichner größer als w aber kleiner als p_{i-1} ist. Existiert ein solcher Peer w, so lässt dieser mit Hilfe einer Verbindungsnachricht $VN(p_{i-1}, w, u)$ eine unmarkierte Kante von p_{i-1} auf w erstellen.

Aufgabe der Verbindungskanten ist somit das Aufspüren eines maximalen Elements w im Intervall $[p_i, p_{i-1}]$, mit dem Ziel, zwischen w und p_{i-1} eine unmarkierte Kante zu erzeugen. So kann ein potenziell neuer linker Nachbar für p_{i-1} – und entsprechend ein neuer Nachbar für w – gefunden werden. Zusätzlich verhindert diese Methode, dass der Re-Chord-Ring in verschiedene Segmente aufgeteilt wird. Dies ist in Abbildung 3.6 zu sehen: (a) zeigt die beiden virtuelle Zwillinge p_i und p_{i-1}. Dadurch, dass diese zu dem gleichen Wirt p_0 gehören, wissen diese gegenseitig von ihrer Existenz. p_{i-1} jedoch besitzt keine Nachbarn in dem Intervall $[p_i, p_{i-1}]$. Da p_i jedoch die Nachbarn v und w in diesem Intervall besitzt, reicht p_i die Verbindungskante an $w = \max\left(\{v \in N(p_i) \mid v < p_{i-1}\}\right)$ weiter. In (b) ist zu sehen, dass

– da w keinen weiteren Nachbarn in dem Intervall $[w, p_{i-1}]$ besitzt – dieser eine unmarkierte Kante zwischen p_{i-1} und w erstellt. Dieses Ergebnis ist in (c) zu sehen: p_{i-1} verbindet die beiden vorher unverbundenen Ringsegmente $[p_i, w]$ und $[p_{p-1}, ..]$ durch eine unmarkierte Kante zwischen p_{i-1} und w. Sollte p_{i-1} bis zur nächsten Ausführung der Linearisierung keinen neuen Nachbarn in dem Intervall $[w, p_{i-1}]$ besitzen, so wird dieser in dem Durchlauf die unmarkierte Kante (p_{i-1}, w) spiegeln und auch w besitzt einen neuen direkten Nachbarn.

Werden die vorgestellten Stabilisierungsoperationen von jedem Peer des Re-Chord-Netzwerkes ausgeführt, so kann gezeigt werden ([KKS11]), dass Satz 3.1.1 gilt:

Satz 3.1.1. *Re-Chord erreicht mit hoher Wahrscheinlichkeit nach $\mathcal{O}(n \cdot \log(n))$ Runden aus jeden Ausgangszustand den stabilen Zustand. In diesem stabilen Zustand enthält Re-Chord Chord als Teilgraphen.*

Hierbei handelt es sich um eine sehr mächtige Eigenschaft, da, wie bereits erwähnt, das Netzwerk somit in der Lage ist, eigenständig auf Veränderungen der Peers als auch auf Veränderungen der Topologie des unterliegenden Rechnernetzes zu reagieren und immer in einen Zustand übergehen, in dem das Netzwerk funktionsfähig ist.

3.2 Loadbalancing mit heterogenen Benutzern

Ziel des Loadbalancing ist eine faire Aufteilung der in einem Netzwerk entstehenden Last auf alle Teilnehmer des Netzwerkes. Die Definition des Begriffs der Fairness kann dabei je nach Anwendungsgebiet unterschiedlich ausfallen. Im Rahmen dieser Arbeit wird die Fairness als eine Aufteilung der Last auf die einzelnen Peers definiert, sodass die Last, die jeder einzelne Peer tragen muss, die Ressourcen, die dieser zur Verfügung stellen kann, nicht überschreitet.

Die Last eines Peers in einem P2P-Netzwerk wird in manchen Arbeiten definiert als Anzahl der Daten, die dieser Peer in dem Netzwerk bereitstellen muss [SW05, RLS+03, SMK+01, BCM03, RS04], andere wiederum definieren die Last als Anzahl der Dateianfragen, die diese pro Zeiteinheit bedienen müssen [BSFK06]. Doch diese Definitionen der Last lassen die Dateigrößen außer Acht, die aufgrund einer Dateianfrage bewegt werden müssen. Eine der kostbarsten Ressourcen eines Peers ist die Kapazität des Internetzugangs, die für den Dateiupload genutzt werden kann. Denn überträgt ein Peer bereits Daten mit voller Kapazität, so kann dieser keine weiteren Dateianfragen bedienen, ohne dabei die Geschwindigkeit der bereits laufenden Dateiübertragungen zu mindern. In der Folge ist die Qualität der Bereitstellung der Dateien dieses Peers gemindert.

Die zwei Ressourcen, die ein Peer in einem P2P-Netzwerk primär zur Verfügung stellt, sind zum einen Speicherplatz, der dafür verwendet wird, um Daten zu spei-

chern, die einem Peer zugeordnet sind, und zum anderen die Upload-Kapazität, die dazu genutzt wird, Dateien zu übertragen. Daher wird im Rahmen dieser Arbeit die Last als Tupel von Speicherplatz und Datenverkehr definiert.

Definition 3.2.1 Die **Last** $l(d)$, die eine Datei d erzeugt, ist definiert als $l(d) = (s, b)$, wobei s der von d benötigte Speicherplatz und b der gemittelte, von d erzeugte Datenverkehr pro Zeiteinheit ist.

Definition 3.2.2 Die **verfügbaren Ressourcen** $w(p)$ eines Peers p sind definiert als $w(p) = (s, b)$, wobei s der von p zur Verfügung gestellte Speicherplatz und b die zur Verfügung gestellte Upload-Kapazität ist.

Wir bezeichnen eine Aufteilung der Last auf die einzelnen Teilnehmer des P2P-Netzwerkes als *fair*, genau dann, wenn kein Peer p existiert, der mehr Last zugeteilt bekommt, als dieser mit seinen verfügbaren Ressourcen befriedigen kann:

Definition 3.2.3 Eine Aufteilung der Last im Netzwerk ist *fair*, genau dann, wenn für alle $p \in \mathcal{P}$ gilt:

$$\sum_{d \in D(p)} l(d) < w(p)$$

wobei $D(p)$ die Menge der p zugeteilten Dateien darstellt, die Summe zweier Lasten l_1 und l_2 definiert ist als $l_1 + l_2 = (s_1, b_1) + (s_2, b_2) = (s_1 + s_2, b_1 + b_2)$ und eine Last $l = (s_l, b_l)$ kleiner als die verfügbaren Ressourcen $w = (s_w, b_w)$ ist, wenn $s_l < s_w$ und $b_l < b_w$ gilt.

3.2.1 Verteilungseigenschaften in P2P-Netzwerken

Die Peer-to-Peer-Netzwerke, die im Rahmen dieser Arbeit betrachtet werden, bestehen aus einem heterogenen Teilnehmerfeld sowie nicht gleich-verteilter Popularität der angebotenen Daten. Des Weiteren operieren diese Netzwerke auf Grundlage des Internets. Dies bedeutet, dass charakteristische Eigenschaften, wie freier Speicherplatz, freie Prozessorzeit und vor allem die Internetanbindung der Teilnehmer und die Zeit, die diese im P2P-Netzwerk verbringen, differieren. Zusätzlich existieren in den P2P-Netzwerken populäre Dateien, die häufiger nachgefragt werden als andere. Diese Popularität weist jedoch – je nach Anwendungsgebiet des P2P-Netzwerkes – eine gewisse Dynamik auf, sodass eine Datei über die Zeit gesehen verschiedenen Anfragehäufigkeiten unterliegt.

In diesem Abschnitt werden diese Unterschiede in der Verteilung der Ressourcen sowie der Dateianfragemuster dargelegt, die als Grundlage der Entwicklung einer adaptiven Lastbalancierung dienen.

Charakteristiken der Peers

Die wichtigsten Leistungsmerkmale eines Peers in einem strukturieren P2P-Netzwerk sind zum einen die verfügbare Geschwindigkeit der *Internetverbindung* und zum anderen die Zeit, die ein Peer im P2P-Netzwerk verbringt, ohne dieses zu verlassen. Letztere wird im Weiteren als *Uptime* bezeichnet. Doch auch weitere Merkmale der Internetverbindung, wie die *Latenz*, sind entscheidend um einen Peer zu charakterisieren (vgl. [SGG02]). Da das Teilnehmen in einem strukturierten P2P-Netzwerk mit dem Zweck der Datenbereitstellung von den Teilnehmern wenig Rechenkapazitäten fordert, ist die verfügbare Rechenleistung der einzelnen Peers kein vorrangiges Kriterium bei der Beurteilung der Leistung eines Peers.

Bandbreite der Internetverbindung eines P2P-Teilnehmers

Damit ein Peer eine Datei möglichst vielen Teilnehmern gleichzeitig zur Verfügung stellen kann, ist es wichtig, dass die sogenannte *Uploadgeschwindigkeit* des Peers möglichst hoch ist. Die Uploadgeschwindigkeit ist die Geschwindigkeit (gemessen in 1000 Bit pro Sekunde, *Kbps*), mit der ein Peer Daten versenden kann. Dem entgegengesetzt existiert die Downloadgeschwindigkeit, also die Geschwindigkeit, mit der ein Peer Daten empfangen kann, ebenfalls gemessen in Kbps. Typische Internetanbindungen, wie *ADSL*, sind asymmetrisch, das bedeutet, dass die Upload- und Downloadgeschwindkeiten nicht gleich sind, sondern in der Regel die Downloadgeschwindigkeit ein Vielfaches der Uploadgeschwindigkeit beträgt.

Die Studie [SGG02] untersuchte im Jahr 2001 die Internetanbindungen von mehr als 17.000 Gnutella-Teilnehmern und stellte dabei fest, dass die Internetanbindungen in etwa einer Normalverteilung mit dem Erwartungswert von circa 3000 *Kbps* im Download und 1000 *Kbps* im Upload folgen, wie in Abbildung 3.7 zu sehen.

Latenz der Internetverbindung eines P2P-Teilnehmers

Die Latenz beschreibt die Zeit, die zwischen dem Absenden eines Datenpaketes und der Abarbeitung dieses Datenpaketes bei dem Empfänger vergeht. Somit besteht die Zeit, die ein vollständiger Dateitransfer benötigt, nicht nur aus der Zeit, die nötig ist, um die Daten zu versenden, sondern zusätzlich aus der Zeit, die diese Daten im Rechnernetz benötigen, um den Empfänger zu erreichen, und der Zeit, die vergeht, bis dieser die Daten verarbeiten kann. Letztere beiden Bestandteile werden zusammen als Latenz bezeichnet.

Je geringer die Zeit ist, die benötigt wird, um eine Datei zu versenden, desto höher ist der Anteil der Latenz an der Gesamtzeit, die ein Datentransfer benötigt. Dies bedeutet insbesondere, dass bei dem Austausch von Dateien mit kleiner Dateigröße die Latenz eine große Rolle in der Dauer der Übertragung spielt, was bedeutet,

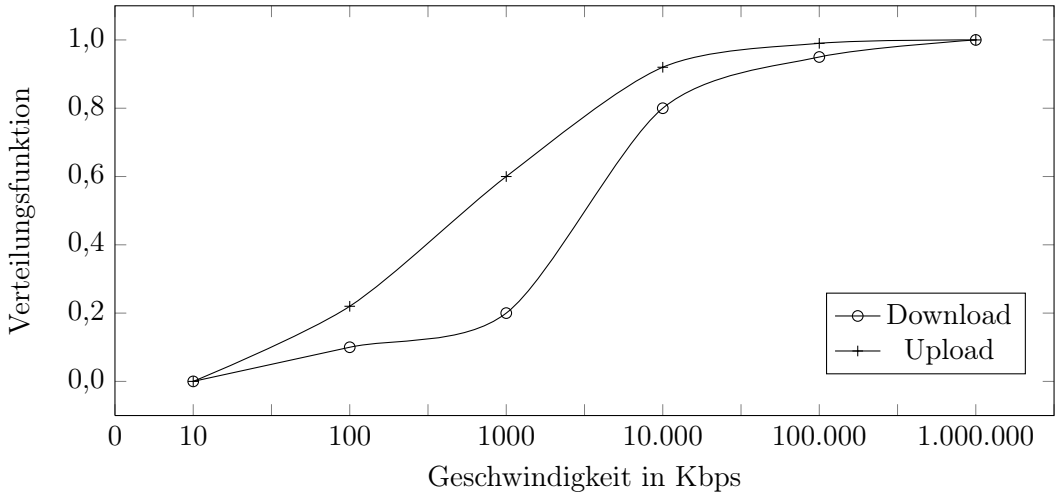


Abbildung 3.7: Verteilungsfunktion der Upload- und Downloadgeschwindigkeiten im Gnutella P2P-Netzwerk, Anfang Mai 2001, gemessen von [SGG02].

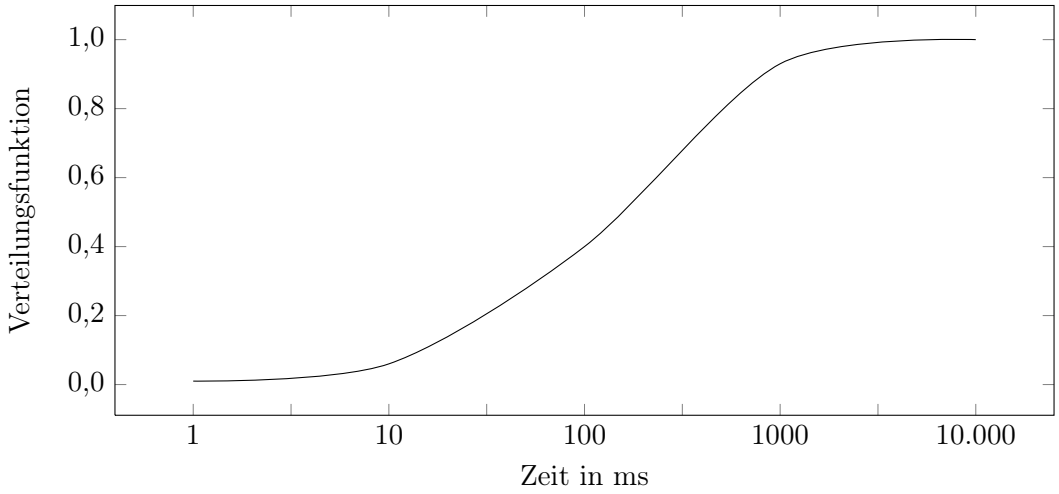


Abbildung 3.8: Verteilungsfunktion der Latenz im Gnutella P2P-Netzwerk, Anfang Mai 2001, gemessen von [SGG02].

dass kleine Dateien signifikant schneller von Peers übertragen werden können, die eine geringe Latenz besitzen. In Abbildung 3.8 ist die Verteilung der Latenzen von Gnutella-Teilnehmern nach [SGG02] zu sehen. Diese Latenzen wurden von der *University of Washington* aus gemessen.

Uptime eines P2P-Teilnehmers

Die *Uptime* bezeichnet den Anteil der Zeit, in der ein Peer mit dem P2P-Netzwerk verbunden ist, und ist ein wichtiger Faktor für die Verfügbarkeit der Daten in einem P2P-Netzwerk. Wie in Kapitel 2.4.6 beschrieben, können Daten aus dem P2P-Netzwerk verschwinden, wenn alle Peers, die diese Daten zur Verfügung stellen, das Netzwerk verlassen, ohne diese an andere Peers weiterzugeben. Somit erhöht ein Peer mit einer geringen Verweilzeit im Netzwerk das Datenaufkommen, da dieser nach dem Beitritt in das Netzwerk zunächst alle Daten übertragen bekommt, für die dieser auf Grund seines Bezeichners zuständig ist. Zusätzlich wird dieser noch Kopien von Daten für andere Peers aufbewahren. Verlässt dieser anschließend nach kurzer Zeit das Netzwerk wieder, so geht die Zuständigkeit der Daten wieder auf einen anderen Peer über. So kommt es, gerade, wenn große Datenmengen kopiert werden müssen, dazu, dass unnötiger Datenverkehr generiert wird. Abbildung 3.9 zeigt die Verteilung der Uptime im Gnutella Netzwerk. Wie hier zu sehen, verweilen 20% der Teilnehmer des Netzwerkes mehr als 50% der Zeit online. Diese Teilnehmer bilden den „festen Kern" des Netzwerkes, wohingegen 50% der Benutzer höchstens 10% der Zeit im Netzwerk verbleiben.

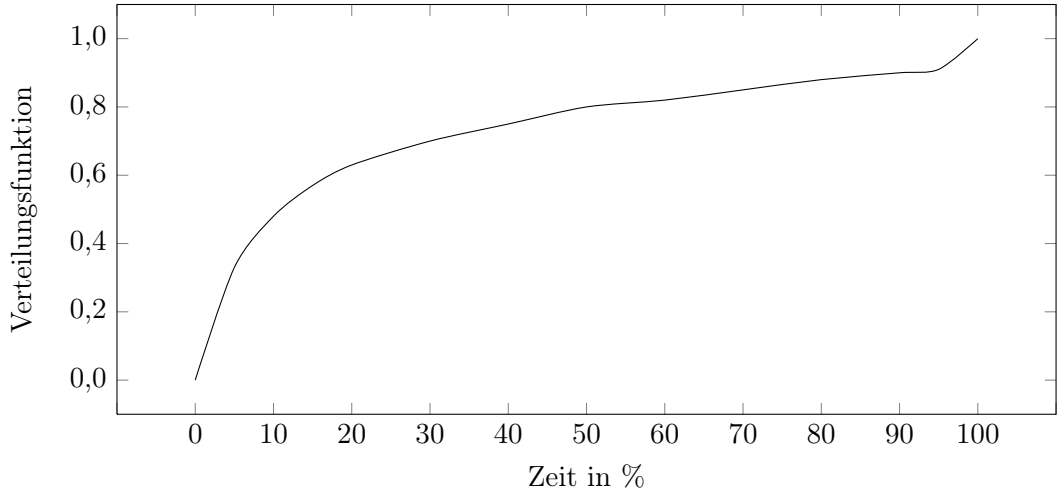


Abbildung 3.9: Verteilungsfunktion der Uptime im Gnutella P2P-Netzwerk, Anfang Mai 2001, gemessen von [SGG02].

3.2.2 Dateianfragemuster

Ein Dateianfragemuster beschreibt die Häufigkeit der Nachfrage von Dateien, die sich in einem P2P-Netzwerk befinden. Es existieren, je nach Anwendung, die auf

dem P2P-Netzwerk arbeitet, ganz verschiedene Annahmen über die Verteilung der Anfragen auf die Dateien des P2P-Netzwerkes.

Viele Arbeiten [RCFB07, BSFK06, DHMK10, Zhu04] unterstellen den Dateianfragemustern in P2P-Netzwerken pauschal eine Zipf-Verteilung, wohingegen auch eine Studie [GDS+03] existiert, die diese Unterstellung, zumindest für das P2P-Netzwerk Kazaa, widerlegt. Die Zipf-Verteilung sagt aus, dass die Wahrscheinlichkeit, dass eine bestimmte Datei x nachgefragt wird, umgekehrt proportional zu ihrem Rang ist. Die Zipf-Verteilung $p(x) \sim \frac{1}{Rang(x)}$ ist in Abbildung 3.10 zu sehen.

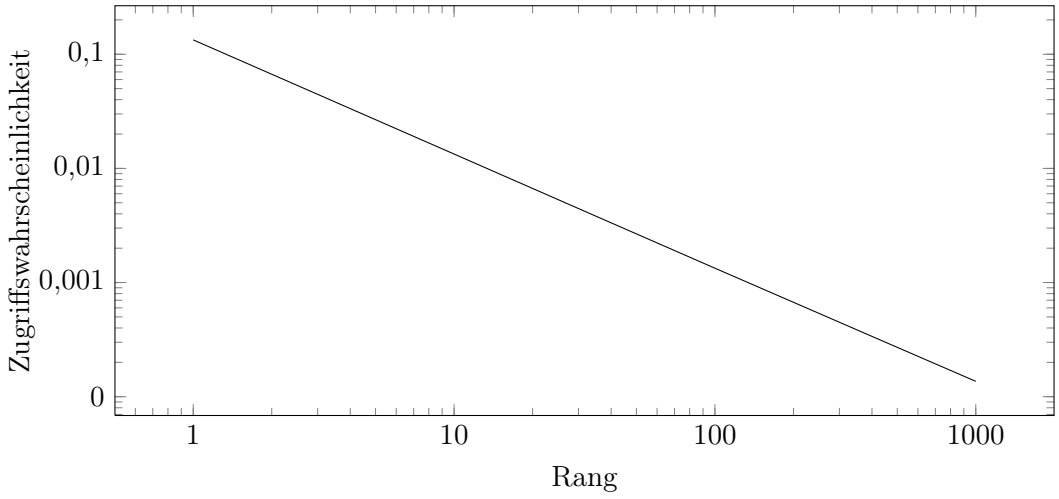


Abbildung 3.10: Zipf-Verteilung mit $p(x) \sim \frac{1}{Rang(x)}$.

Die konkrete Ausprägung des Dateianfragemusters ist abhängig von einer Vielzahl von Faktoren. Einer dieser Faktoren ist die Rate, mit der Dateien aktualisiert werden [GTC+08]. Unterliegt der Inhalt ständiger Veränderung, wie bei Internetseiten wie Blogs, Newsportalen oder sozialen Netzwerken üblich, dann können die Dateianfragemuster mit dem Zipf-Modell gut approximiert werden (vgl. [GDS+03]). Bei vielen P2P-Netzwerken, die in der Praxis eingesetzt werden, geht es jedoch primär um die Bereitstellung von Multimedia-Inhalten, die sich – einmal erstellt – in der Regel nicht ändern. Wenn somit ein Inhalt, nachdem dieser abgerufen wurde, lokal bei dem Anwender gespeichert wird, besteht kein Grund dafür, die gleiche Datei erneut anzufragen. Dieses Verhalten lässt sich allerdings nicht mit der Zipf-Verteilung modellieren. In [GTC+08] wurde jedoch beobachtet, dass sich dieses Verhalten mit Hilfe der gestreckten Exponentialfunktion abbilden lässt. Weiterhin wurde beobachtet, dass sich die Popularität einer Datei über die Zeit gesehen verändert: So befinden sich in Netzwerken, in denen hauptsächlich Multimedia-Inhalte zur Verfügung gestellt werden, größtenteils Neuerscheinungen unter den meist nachgefragten

Daten.

3.2.3 Loadbalancing in Chord

„In a network with N nodes and K keys we would like the distribution of keys to nodes to be tight around $\frac{N}{K}$" formulieren die Autoren von [SMK+01] das Ziel des Loadbalancing von Chord.

Doch erste Simulationsergebnisse [SMK+01] von je $N = 10.000$ Peers mit $K = 100.000$ bis $K = 1.000.000$ Dokumenten im Netzwerk zeigen, dass Chord in der hier vorgestellten Form dieses gesetzte Ziel von je $\frac{N}{K}$ Dokumenten je Peer nicht erfüllt: In jedem Experiment existierten Peers, die für keine Datei zuständig waren, wohingegen der Peer mit den meisten Dokumenten 9.1-mal mehr zugeordnete Dateien besaß als der Durchschnitt über alle Peers. Dieses Ergebnis ist nicht völlig unerwartet, erzeugt doch die in Chord eingesetzte Hashfunktion – SHA-1 – keine gleichverteilten Hashwerte. Daher sind weder Dokumente noch Peers gleich auf dem Chord-Ring verteilt.

Abhilfe schafft hier die Verwendung von virtuellen Peers, wie in Kapitel 3.1.2 im Zusammenhang mit Re-Chord erläutert. Werden in Chord jedem realen Peer r virtuelle Peers zugeordnet, deren Bezeichner unabhängig gleichverteilt durch einen Zufallszahlengenerator gewählt werden, verbessert sich die Lastbalancierung von Chord deutlich: Mit $r = 20$ ließ sich in einem Experiment, in dem 1.000.000 Dokumente auf 10.000 reale Peers (= 200.000 virtuelle Peers) verteilt wurden, beobachten, dass nur 1% der Peers mehr als 1.6-mal mehr Dokumente verwalten mussten, als der Durchschnitt, wohingegen nur 1% der Peers weniger halb so viele Dokumente zugeteilt bekamen als der Durchschnitt.

Einen großen Nachteil dieser Methode stellt jedoch die Menge an zusätzlichen Informationen dar, die jeder reale Peer somit verwalten muss: Ist die Prozessorauslastung, die die Chord-Software auf einem Peer generiert, ein zu vernachlässigender Faktor, so wird dieser mit 20 Instanzen für schwache Peers zu einem wichtigen Leistungskriterium. Zusätzlich wird der Aufwand, der durch das Instandhalten der Routingtabellen entsteht, verzwanzigfacht: Besitzt im Fall ohne virtuelle Peers jeder Peer $\log(10.000) \approx 20$ Einträge in der Routingtabelle, so muss sich jeder reale Peer mit 20 virtuellen Peers um 400 Einträge kümmern.

Weiterhin ist zu bemerken, dass so die tatsächliche Last, die ein jeder realer Peer zu erwarten hat, nicht balanciert wird. Wie in Kapitel 3.2.2 bemerkt, ist die Last, die ein Dokument erzeugt, in den seltensten Fällen gleichverteilt. Es kann eher davon ausgegangen werden, dass diese in den meisten Fällen einer Art Zipf-Verteilung folgt. Da Chord nur eine ungefähre Balancierung der Anzahl der Dokumente pro Peer erreicht, ist es – wird eine Zipf-Verteilung als Dateianfragemuster vorausgesetzt – sehr wahrscheinlich, dass die tatsächliche Last sehr ungleich über die Peers verteilt ist.

Mit der in Abbildung 3.10 abgedruckten Zipf-Verteilung gilt $p(x) = \frac{1}{x \cdot ln(1.78 \cdot N)}$, womit bereits 70% aller Anfragen auf die ersten 100 Dokumente (bei $N = 1000$) fallen. Peers, die diese Dokumente verwalten, werden somit um ein Vielfaches mehr an Last aufnehmen als andere. Wird zusätzlich noch betrachtet, dass jede Datei bedingt durch ihre Größe eine andere Last erzeugt, so fällt auf, dass auch dies nicht von Chord berücksichtigt wird.

3.2.4 Bestehende Loadbalancing-Techniken für Chord

Grundsätzlich können die existierenden Loadbalancing Techniken für Chord vier Kategorien zugeordnet werden (vgl. [HY10]):

1. Nutzung von *virtuellen Knoten*
2. Verändern der Position von Peers auf dem Chord-Ring
3. Verändern der Position von Daten auf dem Chord-Ring
4. Adaptive Anpassung der Zuordnung von Dateibereichen zu Peers

Bei der Methode der *virtuellen Peers* ([HLL+04, LS05, SMK+01]) besitzt jeder physische Peer eine bestimmte Anzahl an virtuellen Peers, die mit dem Chord Netzwerk verbunden sind. Allein die Tatsache, dass sich somit jeder physische Peer an mehreren Positionen im Chord-Ring gleichzeitig befindet, bewirkt, dass die Dateien gleichmäßiger auf die physischen Peers aufgeteilt werden. Besitzt jeder physische Peer $\Omega(\log(n))$ virtuelle Peers, so kann gezeigt werden, dass mit sehr hoher Wahrscheinlichkeit jeder physische Peer, abgesehen von einer kleinen multiplikativen Konstante, für gleich viele Dateien zuständig ist [SMK+01]. Einer der Nachteile dieser Methode ist jedoch, dass jeder physische Peer damit auch $\Omega(\log(n))$-mal mehr Verbindungen mit anderen Peers aus dem Netzwerk halten muss. [HLL+04] verwendet die Methode der virtuellen Peers, um für jeden realen Peer proportional zu seinen verfügbaren Ressourcen eine unterschiedliche Anzahl von virtuellen Peers zu erstellen. Dadurch, dass ein realer Peer somit von mehreren Segmenten des Chord-Ringes aus Daten zugeordnet bekommt und die Anzahl der virtuellen Peers pro realen Peer mit steigender Anzahl der verfügbaren Ressourcen ebenso steigt, werden Peers mit höheren verfügbaren Ressourcen mehr Dateien zugeordnet als Peers mit weniger Ressourcen. Zusätzlich wird in [HLL+04] erwogen, virtuelle Peers von einem realen zu einem anderen realen Peer zu verschieben, sollte ersterer zu stark ausgelastet sein.

Bei der *Veränderung der Position von Peers* auf dem Chord-Ring ([LS05, AB07, BKM05]) werden die Bezeichner der Peers unabhängig von der benutzen Hashfunktion geändert, was dazu führt, dass sich die Zuständigkeitsbereiche dieser Peers

ändern. Wird beispielsweise der Bezeichner eines Peers p verringert, so verringert sich auch der Abstand zu p's Predecessor. Dies wiederum führt dazu, dass p für weniger Dateien zuständig ist, wohingegen p's Successor die Zuständigkeit für die abgegebenen Dateien übernimmt.

Die *Veränderung der Position von Dateien* auf dem Chord-Ring ist eine andere Möglichkeit, die Zuordnung von Dateien zu Peers zu beeinflussen. In [BCM03] wird vorgeschlagen, an Stelle von einer Hashfunktion d Hashfunktionen zu benutzen. Ein Element wird dabei initial nur auf dem am wenigsten ausgelasteten der d Peers gespeichert. Wenn ein Peer bei sich verändernder Dateinachfrage überlastet wird, kann dieser über die d Hashfunktionen die $d-1$ anderen potenziellen Positionen für eine Datei ausfindig machen und die Datei dorthin verschieben. Auch mehrere Duplikate an den d potentiellen Positionen wären denkbar. Diese Methode hat jedoch den Nachteil, dass jeder der d Peers kontaktiert werden muss, wenn eine Datei gesucht oder verändert wird. Insbesondere ist es bei dieser Methode jedoch nicht möglich, die d Peers frei zu wählen. Diese sind durch die d festen Hashfunktionen festgelegt, sodass es, gerade wenn eine Datei stark nachgefragt wird, nicht möglich ist, diese von einem bestimmten Peer bereitstellen zu lassen, der genügend Ressourcen besitzt, um die Last zu bedienen, wenn dieser nicht zufällig einer der d ausgewählten Peers ist.

Die vierte Kategorie ist die *adaptive Anpassung der Zuordnung von Dateibereichen zu Peers*. Dabei wird das Hauptaugenmerk auf die Größe des von einem Peer verwalteten Bereiches des Chord-Ringes gelegt. Die Techniken, die zur Anpassung des Bereichs gemacht werden, sind jedoch der ersten und zweiten Kategorie entliehen; es wird somit auch hier mit virtuellen Peers und dem Verändern des Bezeichners von Peers gearbeitet. In [BKM05] wird ein Algorithmus vorgestellt, der die Abstände der Peers zueinander auf dem Chord-Ring „glättet": Dabei werden die Größen der Zuständigkeitsbereiche der Peers in drei Kategorien aufgeteilt: klein, mittel und groß. Anschließend werden sukzessive Peers, die kleine Bereiche betreuen, durch Verändern des Bezeichners in die Mitte von großen Bereichen gesetzt. Nachdem dieses Verfahren konvergiert ist, sind alle Peers für ungefähr die gleiche Anzahl von Dateien zuständig. Würden somit alle Dateien gleich häufig angefragt und hätten alle Peers die selben Ressourcen, wäre das System in einem gut balancierten Lastzustand. In [RS04] wird unter anderem ein Verfahren vorgestellt, dass je nach Auslastung eines Peers dessen Bezeichner inkrementiert bzw. dekrementiert. Somit wird die Last an die direkten Nachbarn abgegeben.

Alle diese Methoden versuchen, die Last im Chord-Netzwerk so zu balancieren, dass Peers, die geringe Kapazitäten zur Verfügung stellen können, nur für eine kleine Anzahl an Dateien zuständig sind. Doch keine dieser Methoden nutzt aktiv die Tatsache aus, dass der Großteil der Last in einem P2P-Netzwerk von einer geringen Anzahl von Dateien generiert wird und dem gegenübergestellt Peers im Netzwerk

existieren, die wesentlich mehr Last aufnehmen können als andere Peers.

4

Adaptive-Chord: adaptives Loadbalancing für Chord

4.1 Motivation

Es existieren viele Techniken, um die Last, die in einem Chord-Netzwerk entsteht, auf die Teilnehmer des Netzwerkes zu verteilen. Wie in Kapitel 3.2.3 beschrieben, beschränken sich die meisten dieser Verfahren jedoch darauf, die Menge der Peers als homogen anzunehmen, und verfolgen somit das Ziel, jedem Peer möglichst gleich viele Dateien zuzuordnen. Andere Verfahren gehen von einer heterogenen Menge von Teilnehmern aus und teilen einem Peer eine Menge von Dateien zu, sodass die Anzahl der Dateien proportional zu dessen verfügbaren Ressourcen ist. Doch bei diesen Verfahren wird meist davon ausgegangen, dass die Dateianfragemuster gleich verteilt sind, sodass Peers, die reich an Ressourcen sind, mehr Dateien zugeordnet bekommen als ressourcenarme Peers und somit die tatsächliche Last, die bei einem Peer entsteht, nicht ausreichend in die Strategien der Lastbalancierung-Verfahren einfließt.

Aus Kapitel 3.2.2 geht hervor, dass die Dateianfragemuster nicht abhängig von dem P2P-Overlay selbst sind, sondern von der Anwendung, die auf diesem Overlay betrieben wird. Ein P2P-Overlay muss somit in der Lage sein, verschiedenste Dateianfragemuster effizient handhaben zu können. Weiterhin wurde festgestellt, dass die Dateianfragemuster in den meisten Fällen einer zeitlichen Veränderung unterliegen. Neue Inhalte werden erschaffen und verdrängen die älteren Inhalte in ihrer Popula-

rität oder die Gewohnheiten der Benutzer verändern sich und somit verändern sich auch die Dateianfragemuster mit der Zeit.

Es ist somit wichtig, dass ein Loadbalancing-Verfahren in der Lage ist, mit all diesen Rahmenbedingungen umzugehen: Mit homogenen Teilnehmern, starken Unterschieden in der Nachfrage von Dateien und einer sich ständig verändernder Popularität der Daten. Für viele Anwendungsbereiche ist es zusätzlich wichtig, dass die angefragten Dateien auch aktuell sind: Macht die Anwendung, die auf dem Overlay arbeitetet, von der Möglichkeit Gebrauch, dass Daten in einem P2P-Netzwerk verändert werden können, so sollen, nach der Veränderung an einer Datei, alle Kopien, die von dieser Datei durch das P2P-Netzwerk ausgeliefert werden, die jeweils neueste Version einer Datei sein.

In den nachfolgenden Kapiteln wird das in dieser Arbeit entwickelte Lastbalancierungsverfahren *Adaptive-Chord* vorgestellt. Adaptive-Chord arbeitet mit dem Ziel, für jede Datei einen Peer im Netzwerk zu finden, der ausreichend Kapazitäten zur Verfügung stellen kann, um die Nachfrage nach der Datei zu bedienen, ohne von dieser Nachfrage überlastet zu werden. Adaptive-Chord bezieht dazu direkt die tatsächliche Last in das Verfahren ein, die entsteht, wenn eine Datei im Netzwerk übertragen wird. Die nachfolgenden Kapitel sind dabei wie folgt aufgebaut: zunächst wird die von Adaptive-Chord verwirklichte Idee skizziert. Anschließend wird detailliert beschrieben, wie die einzelnen Bestandteile von Adaptive-Chord arbeiten, um diese mit der Beschreibung des eigentlichen Lastbalancierungs-Algorithmus zusammenzufügen. Abschließend werden Details der Referenzimplementierung erläutert.

4.2 Idee

Im Rahmen dieser Arbeit wird eine Loadbalancing-Strategie entwickelt, die die Verantwortung für eine Datei von der Bereitstellung dieser Datei entkoppelt. In strukturierten P2P-Netzwerken besitzt jeder Peer die Verantwortung für die Bereitstellung von einer gewissen Menge an Dateien. In Chord sind dies beispielsweise alle Dateien, deren Bezeichner sich zwischen dem Bezeichner eines Peers p und dessen Vorgänger auf dem Chord-Ring einordnen lassen. Durch den Routing-Algorithmus werden somit alle Anfragen für diese Dateien im Netzwerk an p gestellt. Daher ist p für diese Dateien verantwortlich. Da p eine Kopie der angefragten Datei besitzt, kann dieser Anfragen direkt beantworten, in dem dieser die gesuchte Datei an den anfragenden Peer versendet. Somit ist p nicht nur für die Datei verantwortlich, p stellt diese Datei auch im Netzwerk bereit. Da sich in vielen denkbaren Szenarien die verfügbaren Ressourcen der Peers jedoch voneinander unterscheiden und sich somit unter den Teilnehmern des Netzwerkes „starke" und „schwache" Peers befinden, die viele bzw. wenig Ressourcen mit den anderen Teilnehmern des Netzwerkes teilen können, kann

es passieren, dass eine Datei, die eine hohe Last erzeugt, von einem schwachen Peer bereitgestellt wird. Bei dem hier entwickelten Verfahren wird die Last, die eine Datei generiert, im Hinblick auf die Menge von Daten definiert, die übertragen werden müssen, wenn diese Datei im Netzwerk nachgefragt wird, vgl. Definition 3.2.1.

Das hier entwickelte Verfahren bietet die Möglichkeit, jede Datei von einem, für die Last, die diese Datei erzeugt, angemessenem Peer bereitstellen zu lassen. Dies geschieht, indem der Peer, der die Verantwortung für eine Datei besitzt, die durchschnittliche Last, die diese erzeugt, ermittelt und anschließend einen Peer in P2P-Netzwerk ausfindig macht, der über genügend freie Ressourcen verfügt, um die Datei bereitstellen zu können. Dieser Peer wird anschließend für eine gewisse Zeit eine Kopie der Datei im Netzwerk bereitstellen.

Da sich die Popularität der Daten über die Zeit verändert, und somit auch die Last, die eine Datei generiert, muss die Zuteilung von Dateien auf Peers adaptiv erfolgen. Dies bedeutet, dass sobald eine Datei droht, mehr Last zu erzeugen, als der aktuell bereitstellende Peer bedienen kann, diese Datei auf einen stärkeren Peer ausgelagert werden muss. Dieser Prozess der Auslagerung sollte dabei so effektiv wie möglich sein, sodass schnell auf sich verändernde Lasten reagiert werden kann. Für diesen Auslagerungsprozess wird in dem hier vorgestellten Verfahren eine zusätzliche Overlay-Ebene eingeführt. Diese zusätzliche Overlay-Ebene ist in der Lage, gut mit einer wachsenden Anzahl von Peers zu skalieren und sich schnell an verändernde freie Ressourcen der teilnehmenden Peers anzupassen.

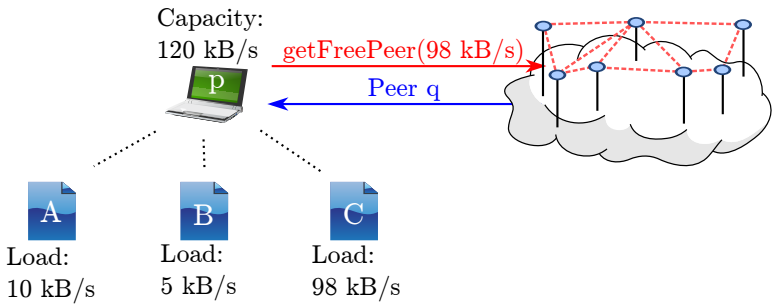


Abbildung 4.1: Idee von Adaptive-Chord: Ein überlasteter Peer stellt eine Anfrage an ein zusätzliches Overlay und erhält die Kontaktdaten eines Peers q, der genug freie Ressourcen zur Verfügung stellt, um eine Datei von p zu übernehmen.

Ist ein Peer p zu einem Zeitpunkt im Sinne der Definition 3.2.3 überlastet, das bedeutet, dass die Summe der Lasten aller Dateien, die dieser zur Verfügung stellt, dessen maximale Übertragungsrate übersteigen, so macht es das hier entwickelte Verfahren möglich, die Bereitstellung von p zugeteilten Dateien auf andere Peers im Netzwerk auszulagern. Hierfür kann dieser Peer, wie in Abbildung 4.1 zu sehen, mit der zusätzlichen Overlay-Ebene einen Peer q finden, der eine gewisse Anzahl

an freien Ressourcen besitzt. Anschließend lässt p eine Datei d von dem Peer q im Netzwerk bereitstellen, für die die Verantwortung bei p liegt. Dies geschieht, indem p die Datei d an q überträgt. q speichert diese Datei und wird ab diesem Zeitpunkt Suchanfragen nach d beantworten.

Nachdem ein Peer p eine Datei d an einen Peer q ausgelagert hat, muss dafür gesorgt werden, dass Dateianfragen nach der Datei d zu q gelangen. Betrachten wir dazu noch einmal die Methode, mit der in Chord nach einer Datei im Netzwerk gesucht wird: Um eine Datei zu finden, wird zunächst mit Hilfe der Hashfunktion SHA-1 der Bezeichner h der Datei d berechnet. Anschließend wird der Peer mit dem zu h nächsthöheren Bezeichner gesucht, da dieser für d zuständig ist. Dies geschieht mit dem in Kapitel 2.4.3 vorgestellten Algorithmus FindPredecessor. Da in den Routingalgorithmus jedoch keine Informationen über bereits ausgelagerte Dateien einfließen, wird eine Anfrage nach der Datei d immer an den Peer p gestellt. Da p die Datei jedoch an q ausgelagert hat, muss p die Suchanfrage anschließend weiter zu q leiten. Damit erhöht sich die Anzahl der Hops im Overlay auf der Route nach einer ausgelagerten Datei um eins. Siehe hierzu Abbildung 4.2. Dadurch, dass p zwar weiterhin Anfragen zu der Datei d beantworten muss, wird dieser auch, nachdem dieser eine Datei ausgelagert hat, weiter einen Teil der entstehenden Last tragen müssen, jedoch ist dieser nicht mehr für die Übermittlung der Datei selbst verantwortlich, die – eine gewisse Dateigröße vorausgesetzt – den Großteil der entstehenden Last ausmacht.

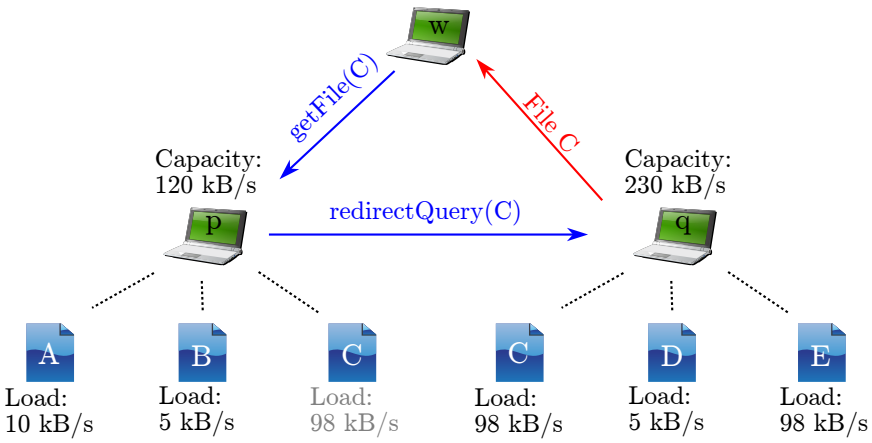


Abbildung 4.2: Peer p leitet eine Anfrage von Peer w nach Datei C an den Peer q weiter, der eine Kopie der Datei C bereit hält.

4.3 Entwurf

In diesem Abschnitt wird das Adaptive-Chord-Verfahren beschrieben, ein Verfahren, das mit der eingangs erläuterten Idee die erzeugte Last in einem P2P-Netzwerk balanciert. Dabei wird zunächst auf die erwähnte zusätzliche Overlay-Ebene eingegangen, die es möglich macht, Peers aufzufinden, die eine gewisse Anzahl an freien Ressourcen zur Verfügung stellen können. Anschließend wird ein Algorithmus vorgestellt, der mit Hilfe dieser zusätzlichen Overlay-Ebene die Last im Netzwerk auf die Peers verteilt, sodass unter bestimmten Rahmenbedingungen kein Peer existiert, der mehr Last tragen muss, als dieser zur Verfügung stellen kann.

4.3.1 P2P-Overlay zum Auffinden von freien Ressourcen

Um effizient einen Partner für die Lastbalancierung zu finden, der eine bestimmte Menge an freien Ressourcen zur Verfügung stellt, wird in Adaptive-Chord eine zusätzliche Overlay-Ebene eingesetzt. Da diese zweite Overlay-Ebene ebenfalls von allen Peers der ersten Ebene benutzt wird, gelten für beide Ebenen die gleichen Anforderungen: eine gute Skalierbarkeit bei wachsender Anzahl an Teilnehmern, Robustheit gegenüber Ausfällen von einzelnen Peers und das schnelle Auffinden von Informationen. In Adaptive-Chord wird als zweite Overlay-Ebene ein leicht verändertes Chord-Overlay genutzt. Dieses abgewandelte Chord-Overlay wird im Weiteren *Capacity-Chord* genannt. In diesem veränderten Chord-Overlay werden sich keine Dateien befinden. Das Netzwerk verfolgt ausschließlich den Zweck, zu einer gegebenen Anzahl an freien Ressourcen effektiv einen Peer zu finden, der diese Menge an freien Ressourcen zur Verfügung stellen kann.

Im originalen Chord-Overlay werden die Bezeichner der Peers, wie in Kapitel 2.4 erläutert, mit Hilfe der Hashfunktion SHA-1 aus deren IP-Adresse abgeleitet. Daraus resultierend erhalten die Peers eine pseudozufällige Position auf dem Chord-Ring. Dies erfolgt mit dem Ziel, die Dateien, die in einem Chord-Netzwerk existieren, möglichst gleich auf die Peers des Netzwerkes zu verteilen. Werden in einem Chord-Netzwerk jedoch keine Dateien angeboten, so entsteht kein Vorteil daraus, dass die Peers gleichverteilt auf dem Chord-Ring angeordnet sind. Die Bezeichner eines Peers können in einem solchen Szenario beliebig gewählt werden. Diese Freiheit in der Wahl der Bezeichner wird anschließend in Capacity-Chord wie folgt ausgenutzt, um den Chord-Ring in eine sortierte Liste von Peers zu transformieren:

Tritt ein Peer p dem Capacity-Chord-Netzwerk bei, berechnet dieser zunächst seinen Bezeichner. Dieser besteht aus einer Zahl zwischen 0 und $2^{160} - 1$ und wird im Capacity-Chord-Netzwerk wie folgt berechnet:

1. Bestimmung der durchschnittlichen maximalen Uploadgeschwindigkeit v_{max} mit einen Onlinedienst wie `http://www.speedtest.net/` durch mehrere Mes-

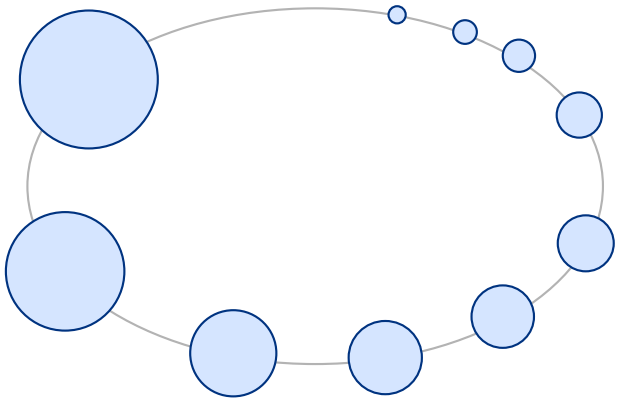


Abbildung 4.3: Capacity-Chord-Ring zum Sortieren von Peers. Peers als Kreise. Je größer ein Peer, desto höher die Anzahl an freien Ressourcen.

sungen zu verschiedenen, zufällig ausgewählten Servern. (v_{max} ist somit für jeden Peer individuell.)

2. Berechnung des Koeffizienten $r = \min\left(1, \frac{v_{max}}{M}\right)$, wobei M die maximale zu erwartende Uploadgeschwindigkeit eines Peers darstellt (z.B. $M = 1000MBit/s$) und eine globale Konstante ist.

3. Berechnung des Bezeichners $id = \left(2^{160} - 1\right) \cdot r$

Diese spezielle Wahl der Bezeichner hat zur Folge, dass die Peers auf dem Capacity-Chord-Ring nach ihrer durchschnittlichen maximalen Uploadgeschwindigkeit sortiert werden. Ein solcher Capacity-Chord-Ring ist in Abbildung 4.3 schematisch dargestellt.

Finden eines geeigneten Peers

Soll in Capacity-Chord ein Peer gefunden werden, der in der Lage ist, eine Datei aufzunehmen, die einen durchschnittlichen Datenverkehr von l $KByte/s$ generiert, so kann der Bezeichner eines passenden Peers berechnet werden als:

$$k = \frac{l}{M} \cdot \left(2^{160} - 1\right) \quad .$$

Wird anschließend im Netzwerk nach dem Peer gesucht, der für k zuständig ist, liefert der Routingalgorithmus von Capacity-Chord den Peer mit dem zu k nächsthöheren Bezeichner. Dieser Peer ist somit in der Lage, eine zusätzliche Last von l

$KByte/s$ aufzunehmen, ohne davon überlastet zu werden. Es existieren jedoch zwei Ausnahmen von dieser Beobachtung:

1. $\frac{l}{M} > 1$: In der Konsequenz gilt $k > 2^{160} - 1$. Da die Last l in diesem Fall größer ist als die maximale zu erwartende Uploadgeschwindigkeit M, wird das Netzwerk nicht in der Lage sein, einen Peer für diese Aufgabe bereitzustellen, da das Netzwerk nicht dafür ausgelegt ist, eine solche Last zu bedienen. Abhilfe schafft in dieser Situation die Erhöhung der globalen Konstante M.

2. Der für k zuständige Peer besitzt einen kleineren Bezeichner als k. In diesem Fall befindet sich k auf dem Chord-Ring zwischen dem Peer mit dem höchsten und dem Peer mit dem niedrigsten Bezeichner. In der Folge existiert kein Peer, der eine Last von l $KByte/s$ aufnehmen kann.

Stabilisationsroutinen

Da mit dem Capacity-Chord-Netzwerk ein Peer gefunden werden soll, dessen *freie* Ressourcen ausreichen, um eine bestimmte Last zu tragen, ist es notwendig, den Bezeichner im Capacity-Chord-Netzwerk zu ändern, wenn sich die *freien* Ressourcen eines Peers verändert haben. Daher muss, zusätzlich zu den Stabilisationsmethoden, die Chord mitbringt, noch eine weitere Stabilisationsmethode in Capacity-Chord periodisch ausgeführt werden, die zunächst die durchschnittliche Geschwindigkeit v ermittelt, mit der in der letzten Zeit Daten übertragen wurden, und anschließend einen neuen Bezeichner $id_{new} = \frac{v_{max}-v}{M} \cdot \left(2^{160} - 1\right)$ berechnet.

Algorithmus 12 n.RenewIdent()

```
1: v ← n.getMeanUploadSpeed()
2: if abs(v − oldV) > oldV/τ then
3:     oldV ← v
4:     n.id ← ((v_max − v)/M) · (2^160 − 1)
5: end if
```

Algorithmus 12 zeigt die Methode, mit der in diesem Chord-Netzwerk der neue Bezeichner bestimmt wird. Da durch die Änderung des Bezeichners der Peer im Netzwerk nicht mehr unter dem alten Bezeichner erreichbar ist, werden durch die Änderung des Bezeichners alle Einträge in den Routingtabellen der anderen Peers ungültig. Somit erhöht die Veränderung des Bezeichners künstlich den Churn im Netzwerk, weshalb der Bezeichner so selten verändert werden sollte wie möglich. Aus diesem Grund wird in Zeile 2 des Algorithmus 12 überprüft, ob sich die durchschnittliche Rate, mit der Daten übertragen werden, seit dem letzten Ändern des Bezeichners um mehr als $\frac{1}{\tau}$ verändert hat. Nur in diesem Fall wird der Bezeichner geändert. Dies mindert zwar die Genauigkeit, mit der die freien Ressourcen im

Netzwerk repräsentiert werden, jedoch ergeben sich daraus weniger Veränderungen im Netzwerk, was dazu führt, dass Peers schneller gefunden werden können.

4.3.2 Der Lastbalancierungs-Algorithmus

Das Adaptive-Chord-Netzwerk besteht aus zwei Chord-Netzwerken: aus einem Netzwerk, mit dessen Hilfe Dateien angeboten und übertragen werden, und einem Netzwerk, das, wie in Kapitel 4.3.1 beschrieben, den Zweck hat, Peers, die eine bestimmte Last aufnehmen können, effektiv zu bestimmen. Dies bedeutet insbesondere, dass in diesem zweiten Netzwerk keine Dateien zu finden sind. Es hat ausschließlich das Ziel, die Peers anhand ihrer ungenutzten Uploadkapazitäten zu sortieren. Im Weiteren wird ersteres Netzwerk als Datennetzwerk und letzteres als Kapazitätsnetzwerk bezeichnet. Peers, die an dem Adaptive-Chord-Netzwerk teilnehmen, sind somit Teilnehmer im Datennetzwerk als auch im Kapazitätsnetzwerk.

Das Datennetzwerk

Das Datennetzwerk besteht aus einem Chord-Netzwerk, wie dieses in Kapitel 2.4 vorgestellt wurde. Zusätzlich zu den Möglichkeiten, die ein Chord-Netzwerk bietet, ist dieses Datennetzwerk in der Lage, sogenannte *Spiegel* zu erstellen. Ein Spiegel einer Datei ist eine exakte Kopie, die von einem Peer angeboten wird, der, laut Chord, nicht für die Datei verantwortlich ist. Spiegel sind somit zusätzliche Kopien, die im Netzwerk angelegt werden können. Die Erstellung eines Spiegels für eine Datei d wird dabei ausschließlich von dem Peer p initiiert, der laut Chord für d zuständig ist. Dieser kann für diesen Prozess einen beliebigen Peer q auswählen und diesem eine Kopie der Datei d senden. Sobald ein Peer einen Spiegel für eine Datei besitzt, beantwortet dieser auch Anfragen nach dieser Datei. Damit p Anfragen, die dieser in Zukunft für Datei d bekommt, an q weiterleiten kann, notiert $p\ q$ als Spiegel für d. Zusätzlich zu dieser Modifikation führen die Peers Zugriffsstatistiken für jede Datei, die diese bereitstellen. Die Zugriffsstatistiken bestehen dabei aus den letzten ϱ Anfragen für jede angebotene Datei und protokollieren die Zeiten, zu denen diese abgerufen wurden. So ist es möglich zu bestimmen, welche Datei wie viel Last in der Vergangenheit aufgrund der bestehenden Anfragen im Netzwerk erzeugt hat.

Analog zu Definition 3.2.3 werden die freien Ressourcen v eines Peers p in Adaptive-Chord definiert als

$$v = v_{max} - \sum_{d \in D(p)} load(d)$$

wobei $load(d)$ die aus den Zugriffsstatistiken ermittelte durchschnittliche Last, gemessen in KByte pro Sekunde, darstellt und $D(p)$ die Menge aller Dateien ist, für die p zuständig ist und die nicht ausgelagert wurden. Ein Peer p gilt als *überladen*,

wenn gilt: $v < \kappa$. Wobei κ die absolute Untergrenze (in $KByte/s$) an freien Ressourcen darstellt. Mit κ kann somit eine Mindestmenge an Ressourcen ausgedrückt werden, die einem Peer zur Verfügung stehen sollen, um unter anderem am Routing teilnehmen zu können. Zusätzlich wird κ benötigt, um Schwankungen in der erzeugten Last zu kompensieren, die auf Grund von sich verändernder Dateianfragemuster auftreten.

Das Kapazitätsnetzwerk

Das Kapazitätsnetzwerk hat die Aufgabe, Peers zu finden, die eine bestimmte Anzahl an freien Kapazitäten besitzen. Dazu wird in Adaptive-Chord das Kapazitätsnetzwerk durch ein Capacity-Chord-Netzwerk aufgebaut, wie dieses bereits in Kapitel 4.3.1 beschrieben wurde.

Der Lastaustausch

Der Algorithmus zur Lastbalancierung von Adaptive-Chord ist in Algorithmus 13 dargestellt und besteht grundlegend aus drei Operationen: Bevor ein Peer Dateien auf andere Peers spiegeln lässt, so muss dieser selbst frei von Spiegeln für andere Peers sein. Daher wird in Zeile 4 des Algorithmus zunächst überprüft, ob p Spiegel für andere Peers bereithält. Ist dies der Fall, so wird p jedes mal, wenn der Algorithmus aufgerufen wird, den Spiegel entfernen, der die höchste Last erzeugt. Anschließend wird der für den Spiegel zuständige Peer über diese Aktion informiert, damit dieser nicht weiter Anfragen nach der Datei an p weiterleitet. So wird verhindert, dass ein Peer durch Daten überlastet wird, für die dieser keine Zuständigkeit besitzt.

Besitzt p jedoch keine Spiegel für andere Peers, so ist die einzige Möglichkeit, Last zu verlieren, Dateien auf andere Peers zu spiegeln. Daher wird immer, wenn der Algorithmus in einer solchen Situation aufgerufen wird, die Datei f ermittelt, die die höchste Last für p erzeugt (vgl. Zeile 7). Anschließend wird mit Zeile 8 ein Peer q bestimmt, der genügend freie Ressourcen bereitstellen kann, um die ermittelte Datei aufnehmen zu können. Dies geschieht über das Kapazitätsnetzwerk. Existiert ein solcher Peer q, so wird diesem in Zeile 10 eine Kopie der Datei f gesendet. Daraufhin empfängt q diese und erstellt so einen Spiegel für f. Anschließend (Zeile 11) markiert p die Datei als ausgelagert. Jedes Mal, wenn im Anschluss eine Anfrage nach der Datei f an p gestellt wird, wird dieser die Anfrage an q weiterleiten.

Um dafür zu sorgen, dass nicht zu viele Spiegel von Dateien im Netzwerk entstehen, um damit zum einen die Länge der Routen zu verkürzen, und zum anderen dafür zu sorgen, dass Peers mit hohen Kapazitäten nicht über die Zeit immer mehr Spiegel zugeordnet bekommen, kann ein Peer vorher aufgestellte Spiegel auch wieder entfernen. Dies ist in den Zeilen 15 bis 19 des Algorithmus 13 dargestellt: Zunächst

Algorithmus 13 n.LoadBalancing()

```
1:  load ← n.getMeanUploadSpeed()
2:  remain ← v_max − load
3:  if remain < κ then                                   ▷ peer is overloaded
4:      if n.MirrorCount > 0 then   ▷ get rid of objects mirrored for other peers
5:          n.removeMirrorWithHighestLoadAndInfromOrigin()
6:      else
7:          File f ← n.getFileWithHighestLoad()
8:          Peer q ← n.findPeerWithFreeLoad(f.getLoad())
9:          if q ≠ Null then
10:             q.addMirror(f)                  ▷ create a mirror for f on peer q
11:             n.fileOutsourcedTo(f, q)        ▷ mark file f to be outsourced to q
12:         end if
13:     end if
14: else                                                 ▷ peer is not overloaded
15:     threshold ← (remain − κ) · (1 − γ)                ▷ calculate free load
16:     if ∃ File m ∈ outsourcedFiles : m.load < threshold then
17:         Peer q ← m.getMirrorHoster()                  ▷ lookup mirror for m
18:         q.removeMirror(m)                    ▷ tell q to delete mirror for m
19:         n.gotFileBack(m)                ▷ mark file as beeing not outsourced
20:     end if
21: end if
```

wird ermittelt, wie viel Ressourcen dem Peer zur Verfügung stehen, um die Bereitstellung von Dateien zurückzunehmen, ohne davon überlastet zu werden. Dieser Wert wird anschließend mit $(1 - \gamma)$ multipliziert. $\gamma \in [0, 1]$ bestimmt somit eine anteilige Sicherheit, die dafür sorgt, dass ein Peer dadurch, dass dieser eine Datei zurücknimmt, nicht soweit belastet ist, dass dieser in der nächsten Lastbalancierungsrunde durch die leicht schwankenden Dateianfragemuster wieder eine Datei auslagern muss. Anschließend wird überprüft, ob eine Datei m existiert, die p auf einen anderen Peer q ausgelagert hat, die so wenig Last erzeugt, dass p diese mit den freien Ressourcen bedienen kann, ohne damit überlastet zu werden. Ist dies der Fall, so kontaktiert p q und teilt diesem mit, dass die Datei m nicht weiter bereitgestellt werden muss. q wird daraufhin die lokale Kopie von m entfernen. Abschließend entfernt p mit Zeile 19 des Algorithmus die Markierung der Datei m, die diese als an q ausgelagert kennzeichnet. Ab diesem Zeitpunkt wird p wieder alle Anfragen nach m selbst beantworten.

Zeile 9 von Algorithmus 13 verdient in diesem Zusammenhang eine gesonderte Betrachtung. q ist Null, wenn kein Peer ermittelt werden kann, der in der Lage ist, die Last aufzunehmen, die von Datei f erzeugt wird. In diesem Fall verbleibt die Datei auf dem Peer p, der für f zuständig ist, auch wenn dieser nicht in der Lage

ist, die Last zu tragen, die diese Datei erzeugt. Der Lastbalancierungsalgorithmus arbeitet somit nach dem Prinzip, dass kein Peer durch Dateien überlastet wird, für die dieser nicht selbst zuständig (im Sinne von Chord) ist. Dieses Prinzip wird angewendet, da es nicht sinnvoll ist, eine Datei auf einen Peer auszulagern, der nicht genügend Ressourcen für diese Datei zur Verfügung stellen kann. Denn auch dieser Peer wäre nicht in der Lage, mehr Anfragen pro Zeiteinheit zu bedienen, als neue Anfragen für die Datei gestellt werden. In der Folge würden auch bei diesem Peer immer mehr Dateiübertragungen parallel ausgeführt werden. Dies führt dazu, dass die Raten, mit denen die Dateien ausgeliefert werden, gegen Null streben, beziehungsweise, wenn nur eine feste Anzahl von Dateiübertragungen parallel ausgeführt werden und die restlichen in eine Warteschlange eingereiht werden, die Länge der Warteschlange mit der Zeit gegen unendlich strebt.

Erhöhung des durchschnittlichen Datendurchsatzes im Netzwerk

Durch die in Algorithmus 13 dargestellte Methode des Loadbalancing wird immer dann Last von einem Peer auf einen anderen übertragen, wenn ersterer Peer nicht über genügend Ressourcen verfügt, um pro Zeiteinheit mehr Anfragen zu beantworten als neue Anfragen hinzukommen. Dies verhindert, dass Dateien auf Grund der Überlastung des anbietenden Peers nicht im Netzwerk verfügbar sind. Diese Methode wird jedoch in der Regel nicht dazu beitragen, dass die Datenraten, mit denen die Dateien übertragen werden, steigen: So wird eine Datei, die durch Chord einem Peer zugeordnet ist, der über eine geringe Uploadgeschwindigkeit verfügt, so lange von diesem Peer angeboten werden, bis die Datei mehr Last erzeugt, als dieser Peer bedienen kann. Wird diese Datei jedoch selten angefragt, sodass die erzeugte Last nicht ausreicht, damit diese Datei ausgelagert wird, so verbleibt diese auf dem schwachen Peer und wird in der Folge für andere Peers nur mit einer langsamen Datenrate abrufbar sein, obwohl andere Peers im Netzwerk existieren könnten, die genug freie Ressourcen zur Verfügung stellen könnten, um diese Datei mit höheren Datenraten im Netzwerk anzubieten.

Betrachten wir dazu folgendes Beispiel: Sei der Peer p für die Datei d im Sinne von Chord verantwortlich, wobei p eine durchschnittliche maximale Uploaddatenrate von $v_{max} = 2,5\ kB/s$ zur Verfügung stellt. Betrage weiter die Dateigröße von d $10\ kB$. Damit beträgt die durchschnittliche Zeit, um die Datei d zu übertragen $t = \frac{10\ kB}{2,5\ kB/s} = 4s$. Wird die Datei d alle 5 Sekunden angefragt und p besitzt keine weiteren Dateien, so ist p nicht überlastet und wird, mit einer gewissen Wahl von κ, nicht als überlastet eingestuft. Die Datei d würde somit nicht zu einem anderen Peer q gespiegelt werden, obwohl es möglich wäre, dass ein solcher Peer q im Netzwerk existiert, der in der Lage wäre, d schneller auszuliefern als p.

Um dennoch die durchschnittliche Datenrate im Netzwerk zu erhöhen, wird aus-

geschlossen, dass Peers, die über eine sehr geringe Datenrate verfügen, Daten im Netzwerk zur Verfügung stellen. Dies geschieht, indem die in Algorithmus 13 benutzte Definition des überlasteten Peers verändert wird. Zu diesem Zweck wird Zeile 3 des Algorithmus ausgetauscht durch

$$\textbf{if } remain < \kappa \textbf{ or } v_{max} < \lambda \textbf{ then}$$

Somit kann durch λ eine untere Schranke für die Uploadgeschwindigkeit festgelegt werden, mit der ein Peer Daten im Netzwerk zur Verfügung stellen darf.

Zusätzlich zu dieser unteren Schranke für die Uploadgeschwindigkeit kann noch eine weitere untere Schranke festgelegt werden. Diese Schranke bezieht sich auf die freie Uploadgeschwindigkeit möglicher Partner zum Lastaustausch: Je mehr freie Ressourcen ein Peer zur Verfügung stellen kann, desto schneller ist auch der Download von diesem Peer. Somit hat die Wahl des Peers, auf den eine Datei gespiegelt wird, direkte Auswirkungen auf die Qualität, mit der diese Datei im Netzwerk zur Verfügung gestellt wird. Aus diesem Grund ist es, soll die durchschnittliche Datenrate, mit der Dateien ausgetauscht werden, erhöht werden, wichtig, dass der Peer, auf den eine Datei gespiegelt wird, möglichst viele freie Ressourcen besitzt. Um einen solchen Peer mit möglichst vielen freien Ressourcen im Netzwerk zu finden, wird daher eine untere Schranke ω von jedem Peer gepflegt, die angibt, wie viele freie Ressourcen ein Peer mindestens besitzen muss, damit dieser eine Datei spiegeln kann. Abbildung 4.4 zeigt das Datennetzwerk mit den eingezeichneten Grenzen λ und ω: Besitzt ein Peer weniger freie Ressourcen als λ, so sucht dieser einen Peer, der mehr Ressourcen als ω zur Verfügung stellen kann, und erstellt auf diesem einen Spiegel für eine Datei.

Dadurch, dass für die Lastbalancierung somit nur Peers gesucht werden, deren freie Ressourcen größer als ω sind, kann keine Last mehr verschoben werden, wenn sich kein Peer im Netzwerk befindet, der diesen Anforderungen genügt. In diesem Szenario können jedoch weiterhin Peers existieren, die mehr als λ freie Ressourcen besitzen, sodass es in diesem Fall durchaus möglich wäre, die Lastsituation im Netzwerk zu verbessern. Aus diesem Grund handelt es sich bei ω nicht um eine feste, sondern um eine variable Grenze, die jeder Peer durch lokale Regeln mit der Zeit verändert.

Algorithmus 14 zeigt, wie jeder Peer, nachdem die Stabilisierungsroutinen ausgeführt wurden, die untere Schranke ω aktualisiert. Wurde eine Datei erfolgreich auf einen Peer gespiegelt, so wird ω für die nächste Operation um 20% erhöht, sodass der nächste Peer, der im Kapazitätsnetzwerk gesucht wird, eine höhere Anzahl an Ressourcen zur Verfügung stellen kann. Ist jedoch kein Peer gefunden worden, der mehr als ω Ressourcen zur Verfügung stellen kann, so wird ω um 20% reduziert. So passt Algorithmus 14 die Anforderungen an einen möglichen Spiegel für eine Datei

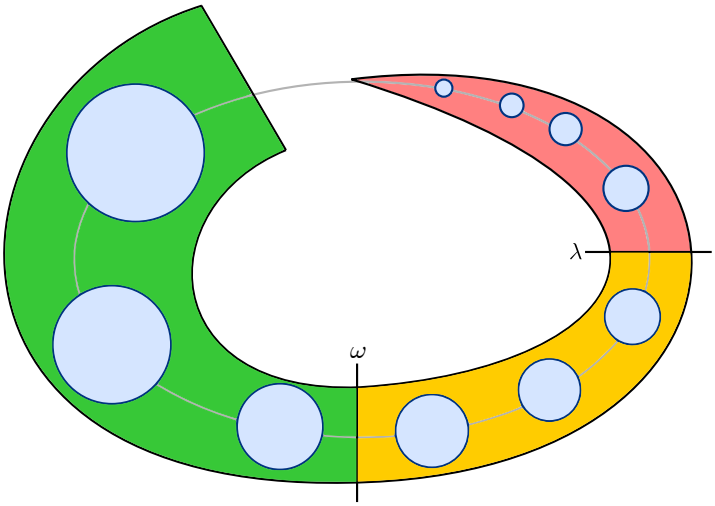


Abbildung 4.4: Untere Schranken der Datenraten im Datennetzwerk.

an die jeweils aktuelle Situation im Kapazitätsnetzwerk an, was dazu führt, dass sich ω mit steigender Zahl an Operationen näher an die höchste Anzahl an freien Ressourcen im Kapazitätsnetzwerk annähert.

4.4 Implementierung

Die hier vorgestellte Erweiterung von Chord – Adaptive-Chord – wurde im Rahmen dieser Arbeit als Erweiterung für den P2P-Simulator *PeerfactSim.KOM*[1] implementiert. Dabei stand bereits eine vollständig funktionstüchtige Version von Chord zur Verfügung, die jedoch noch so erweitert werden musste, dass diese das Konzept der Spiegel, wie in Kapitel 4.2 beschrieben, realisiert. Anschließend wurde, auf dieser Chord-Version aufbauend, das Adaptive-Chord-Verfahren implementiert. Erläuterungen zu dieser Implementierung finden sich in den nachfolgenden Kapiteln. Zunächst wird ein Überblick über den Aufbau des implementierten Adaptive-Chord-Overlays gegeben. Dabei wird insbesondere auf die Nähe zwischen den Implementierungen von Chord und Adaptive-Chord eingegangen und vorgestellt, wie sich das Konzept des Adaptive-Chord auch auf andere Overlays übertragen lässt. Anschließend wird erläutert, wie durch Addieren einer niedrigen Zufallszahl zu den Bezeichnern im Kapazitätsnetzwerk das Adaptive-Chord Verfahren robuster gegenüber Ausfällen von einzelnen Peers gemacht wird.

[1] `http://www.peerfact.org`

Algorithmus 14 n.UpdateBounds()

```
1: ω_old ← ω
2: if last Mirror Operation successful then
3:     ω ← 1.2 ω
4: else
5:     ω ← 0.8 ω
6: end if
7: if ω > M then
8:     ω ← ω_old
9: end if
10: if ω < λ then
11:     ω ← ω_initial
12: end if
```

4.4.1 Aufbau

Das Adaptive-Chord Netzwerk besteht, wie in Kapitel 4.3.2 beschrieben, aus zwei voneinander unabhängigen Chord-Netzwerken, die sich jedoch gegenseitig durch den Lastbalancierungsalgorithmus beeinflussen. Es wäre dafür durchaus denkbar, dass, im Falle von Adaptive-Chord, diese beiden Netzwerke zu einem einzigen Netzwerk zusammengefasst werden. Jeder Peer könnte somit durch die Technik der virtuellen Peers zwei Positionen innerhalb des Netzwerkes besitzen. Spezielle Einträge in den Routingtabellen, ähnlich zu r_r und r_l in Re-Chord, könnten anschließend dafür sorgen, dass sicher zwischen Teilnehmern des Datennetzwerkes und des Kapazitätsnetzwerkes unterschieden werden kann. Vorteile einer solchen Implementierung wäre die mögliche gegenseitige Unterstützung dieser zwei Peers im Netzwerk. Diese könnte, ähnlich wie in Re-Chord beschrieben, ausgenutzt werden, um Informationen über das Netzwerk zwischen diesen virtuellen Peers zu teilen. So würde das Netzwerk an Stabilität gewinnen, da sich die Anzahl der Informationen, die jedem Peer zur Verfügung stehen, erhöht. Ein Nachteil dieser Implementierung wäre jedoch, dass somit die Möglichkeit genommen wird, das Datennetzwerk, ohne großen Aufwand zu verursachen, durch ein anderes P2P-Overlay auszutauschen.

Die in Kapitel 4.2 dargelegte Idee des Loadbalancing beschreibt einen Prozess, in dem ein Peer eingehende Dateianfragen an einen Spiegel für diese Datei weiterleitet, wenn ein solcher existiert. Diese Idee ist vollkommen unabhängig von dem Overlay, das diese Art von Loadbalancing nutzt. Somit existieren eine Vielzahl von Overlay-Netzwerken, die dieses Loadbalancing-Verfahren nutzen können. Diese Tatsache wurde in der Implementierung von Adaptive-Chord berücksichtigt, und daher wurde eine abstrakte Klasse entworfen, die alle nötigen Methoden aufweist, die ein Overlay zur Verfügung stellen muss, wenn dieses den Adaptive-Chord-Ansatz verwenden soll. Eine Overlay-Implementierug muss sich somit nur von dieser abstrakten

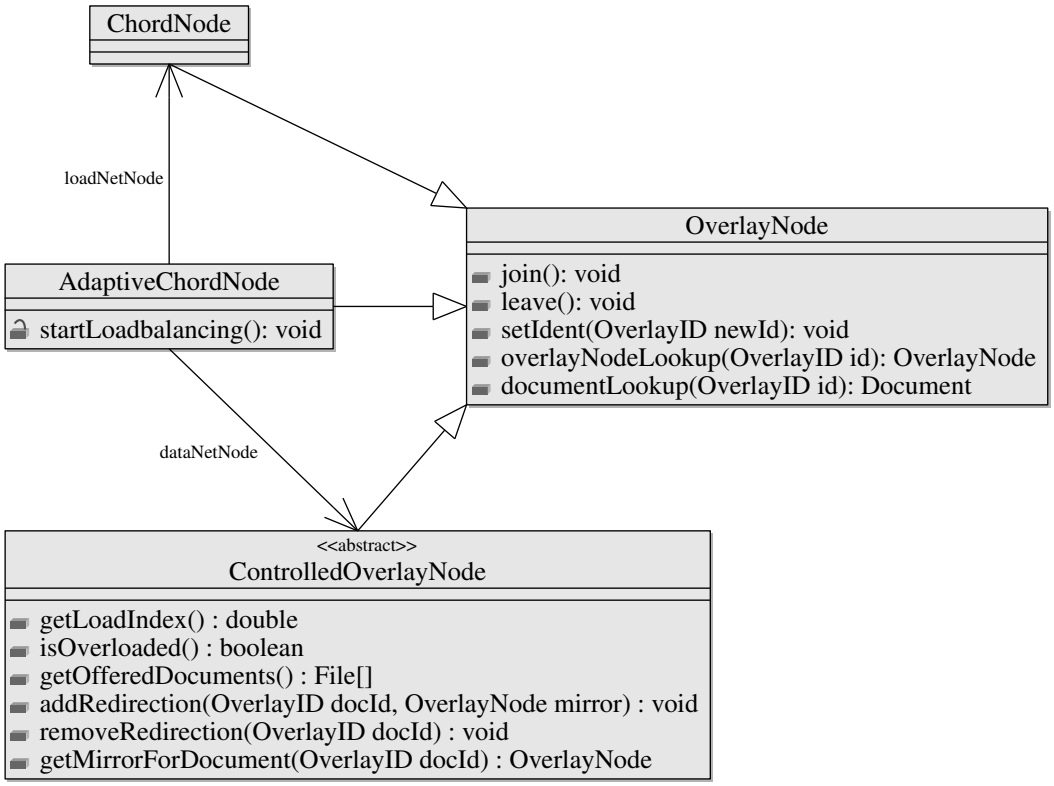


Abbildung 4.5: Architektur der Adaptive-Chord-Referenzimplementierung.

Klasse ableiten, einige zusätzliche Methoden implementieren und kann anschließend als Datennetzwerk in der Adaptive-Chord-Implementierung genutzt werden.

Abbildung 4.5 zeigt die Architektur, die in der Referenzimplementierung gewählt wurde: Ein Peer, der im Adaptive-Chord-Netzwerk teilnimmt, besitzt zwei virtuelle Peers: Dies sind der `loadNetNode`, der an einem Chord-Netzwerk teilnimmt, und der `dataNetNode`, der an einem Capacity-Chord-Netzwerk teilnimmt. Wie in dem abgebildeten Klassendiagramm zu sehen, ist das Datennetzwerk nicht auf ein Chord-Netzwerk beschränkt. In der Implementierung wird für das Datennetzwerk eine Klasse genutzt, die von der abstrakten Klasse `ControlledOverlayNode` abgeleitet ist. Diese abstrakte Klasse erbt vom Typ `OverlayNode` und erweitert diese um Methoden, die die sonst internen Methoden zum Anlegen und Entfernen von Spiegeln ebenfalls auch nach außen sichtbar machen. Zusätzlich werden durch die abstrakte Klasse die Methoden `getLoadIndex()` und `isOverloaded()` eingeführt. `getLoadIndex()` ermittelt einen Wert zwischen 0 und 1, der aussagt, wie viel freie Ressourcen der Peer noch zur Verfügung stellen kann. Dieser Index berechnet sich als $\frac{v_{max}-v}{M}$, wie dies bereits in Algorithmus 12 geschieht. Somit erlaubt dieser Index direkte Rückschlüsse auf die aktuelle Position des zu dem `dataNetNode` zugehörigen

`loadNetNode` auf dem Capacity-Chord-Ring. Die Methode `isOverloaded()` gibt Auskunft darüber, ob der `dataNetNode` überladen ist. Dies geschieht vollkommen analog zu den ersten drei Zeilen des Algorithmus 13, indem die verbliebenen freien Ressourcen mit einer unteren Schranke verglichen werden.

Algorithmus 15 UpdateLoadNetIdent()

1: $id_{new} \leftarrow$ `this.dataNetNode.getLoadIndex()` $\cdot (2^{160} - 1)$
2: **if** $abs(id_{new} -$ `this.dataNetNode.id`$) > \frac{\texttt{this.dataNetNode.id}}{\tau}$ **then**
3: `this.loadNetNode.setIdent(`id_{new}`)`
4: **end if**

Algorithmus 15 zeigt beispielhaft, wie ein Objekt der Klasse `AdaptiveChordNode` den Bezeichner im Kapazitätsnetzwerk je nach Anzahl der freien Ressourcen des Peers im Datennetzwerk anpassen kann. Auch andere Methoden, wie das Finden eines Partners zum Lastaustausch und das anschließende Anlegen eines Spiegels, sind so über die Klasse `AdaptiveChordNode` steuerbar, ohne dass zusätzliche Methoden der Klasse `ControlledOverlayNode` ausgeführt werden müssen. Der Loadbalancing-Algorithmus wird somit vollständig auf dem `AdaptiveChordNode` ausgeführt, der dazu auf die öffentlichen Methoden der Klassen `ChordNode` und `ControlledOverlayNode` zurückgreift.

4.4.2 Tie-breaker

Im Chord-Overlay, wie in Kapitel 2.4 beschrieben, wird davon ausgegangen, dass sowohl die Bezeichner für Dateien als auch für die Peers eindeutig sind. Wird ein entsprechendes Capacity-Chord-Netzwerk betrachtet, tritt das Problem auf, dass alle Peers, die eine gleiche Menge von freien Ressourcen aufweisen, auch den gleichen Bezeichner im Netzwerk besitzen. Dies kann dazu führen, dass die Topologie des Capacity-Chord-Netzwerkes nicht aufrechterhalten werden kann und in der Folge der Capacity-Chord-Ring kein Ring mehr ist. Aus diesem Grund ist es wichtig, dass die Bezeichner der Peers im Kapazitätsnetzwerk eindeutig sind – somit keine zwei Peers existieren, die den gleichen Bezeichner besitzen.

Um dies zu erreichen, besteht in der Referenzimplementierung von Adaptive-Chord der Bezeichner eines Peers im Kapazitätsnetzwerk nicht allein aus der Anzahl an freien Ressourcen, sondern zusätzlich noch aus einer Zufallszahl, die zum Bezeichner addiert wird. Um die Auswirkungen dieser Modifikation auf den Betrieb des Kapazitätsnetzwerks zu betrachten, sollte zunächst ermittelt werden, wie weit die Bezeichner von zwei Peers mindestens auseinanderliegen, deren freie Uploadge-

schwindigkeit um 1 $KByte/s$ differieren[2]:

$$\Delta = \frac{2^{160}-1}{M} \cdot 1\ KByte/s = \frac{2^{160}-1}{1000MBit/s} \cdot 8192Bit/s \quad > \quad 10^{43}$$

Somit liegen zwischen den Bezeichnern von zwei Peers, deren freie Uploadgeschwindigkeit sich um mindestens 1 $KByte/s$ unterscheiden, mehr als 10^{43} weitere Bezeichner. Dies kann wie folgt ausgenutzt werden, um zu erreichen, dass die Bezeichner im Kapazitätsnetzwerk paarweise verschieden sind: Wird ein Bezeichner berechnet, geschieht dies zunächst, wie in Kapitel 4.3.2 beschrieben, indem $id_{new} = \frac{v_{max}-v}{M} \cdot \left(2^{160}-1\right)$ berechnet wird. Anschließend wird gleichverteilt ein Element aus $[0, 10^{43}]$ ausgewählt, das auf id_{new} addiert wird. Dies verhindert somit, dass Peers mit der gleichen Anzahl an Ressourcen auch den gleichen Bezeichner im Kapazitätsnetzwerk zugeordnet bekommen. Trotzdem kann im Kapazitätsnetzwerk weiterhin KByte-genau adressiert werden. Sollte trotzdem ein Peer im Kapazitätsnetzwerk einen Bezeichner gewählt haben, der bereits im Netzwerk vorhanden ist, kann dies noch vor der Änderung des Bezeichners erkannt werden: Hierfür sucht ein Peer p, bevor dieser den Bezeichner im Kapazitätsnetzwerk auf id_{new} ändert, nach dem Peer mit dem Bezeichner id_{new}. Wird dieser Peer gefunden, so wählt p einen neuen Bezeichner und überprüft anschließend wieder, ob dieser bereits im Netzwerk vorhanden ist.

Das gleiche Prinzip lässt sich auch auf die Suche im Kapazitätsnetzwerk anwenden: Wird nach einem Peer mit v $KByte/s$ freier Uploadgeschwindigkeit gesucht, so kann auch für die Suche nach diesem Bezeichner ein zufällig gleichverteiltes Element aus $[0, 10^{43}]$ zu dem gesuchten Bezeichner addiert werden. Damit wird erreicht, dass die Anfragen nicht nur zu dem Peer geroutet werden, der den jeweils kleinsten zufälligen Teil des Bezeichners gewählt hat. Wird dieser zufällige Teil für die Suche jedoch aufaddiert, so kann es passieren, dass ein Peer gefunden wird, der $v+1$ $KByte/s$ freie Ressourcen besitzt. Damit wird das Ergebnis der Suche im Netzwerk verfälscht. Der Vorteil dieser Methode ist jedoch, dass so die Anfragen für einen bestimmten $KByte$-Bereich des Ringes nicht mehr nur vor einem Peer beantwortet werden, sondern diese auf alle Teilnehmer in diesem Abschnitt verteilt werden.

Diese Modifikation der Suche hat zusätzlich positive Auswirkungen auf das Verhalten des Loadbalancing bei Churn: Besitzt ein Peer eine maximale Uploadgeschwindigkeit, die kleiner ist als die untere Schranke λ, so wird dieser generell als überlastet eingestuft (vgl. Kapitel 4.3.2). Dieser überlastete Peer wird somit alle Dokumente, die diesem durch Chord zugeordnet werden, auf einen anderen Peer

[2]Für Adaptive-Chord reicht es aus, KByte-genau adressieren zu können, da die beobachteten Lasten einer gewissen Schwankung unterliegen, die es überflüssig macht, diese Byte-genau zu betrachten.

auslagern. Für die erste Datei, die dieser Peer auslagert, wird nach einem Peer gesucht, der noch mindestens ω freie Ressourcen zur Verfügung stellen kann. Dieser besitzt somit den zu $\frac{2^{160}-1}{M} \cdot \omega$ nächsthöheren Bezeichner im Netzwerk. Da diese ausgelagerten Dateien nicht zwangsläufig bereits von anderen Peers angefragt wurden, diese somit aktuell gegebenenfalls noch keine Last erzeugt haben, wird der Peer p, der diese Dateien aufnimmt, auch seinen Bezeichner im Kapazitätsnetzwerk nicht zwangsläufig ändern. Daraus folgt, dass p potentiell von jedem überlasteten Peer die erste Datei, die dieser auslagert, zugeordnet bekommt. Dies hat jedoch zur Folge, dass, wenn p das Netzwerk verlässt, auch alle Spiegel, die dieser Peer besitzt, aus dem Netzwerk verschwinden, woraufhin diese Spiegel erneut aufgestellt werden müssen. Dieser unerwünschte Effekt wird durch den zufälligen Anteil des Bezeichners bei der Suche abgeschwächt.

4.4.3 Speicherplatz

In Definition 3.2.1 wurde die Last, die eine Datei erzeugt, als Tupel von Datenverkehr pro Zeiteinheit und benötigtem Speicherplatz definiert. Diese Definition wurde jedoch nur zu einem Teil von Adaptive-Chord genutzt. So nimmt Adaptive-Chord in der hier vorgestellten Variante keine Rücksicht auf den Speicherplatz, den eine gespeicherte Datei verbraucht. Es wird somit davon ausgegangen, dass jeder Peer, der am Netzwerk teilnimmt, keiner Beschränkung des Speicherplatzes unterliegt. Um diese Beschränkung in das Adaptive-Chord Netzwerk einzuführen, genügt es, wenn ein Peer, nachdem dieser keinen weiteren Speicherplatz mehr zur Verfügung hat, seinen Bezeichner im Kapazitätsnetzwerk auf 0 ändert. Somit wird verhindert, dass auf diesem noch zusätzliche Dateien gespiegelt werden. Des Weiteren existiert in der Referenzimplementierung eine obere Schranke η, die die Anzahl der gespiegelten Dateien pro Peer beschränkt. Besitzt ein Peer mehr als η Dateien, die auf diesen gespiegelt wurden, so ändert dieser seinen Bezeichner im Kapazitätsnetzwerk ebenfalls auf 0. Sobald die Anzahl der Dateien wieder unter η fällt, wird der Peer den Bezeichner im Netzwerk neu berechnen. Dies hat zur Folge, dass ein Peer nicht durch zu viele gespiegelte Dateien überlastet werden kann: In der Berechnung der Last wird in Adaptive-Chord nur der Menge der übertragenen Nutzdaten Rechnung getragen. Der Overhead, der durch das Beantworten von Anfragen und das Aufbauen von Verbindungen generiert wird, wird nicht mit berücksichtigt. Daher existiert mit η eine obere Schranke an Spiegeln, die ein jeder Peer maximal bereitstellen wird. η hat zusätzlich noch eine weitere Aufgabe: Verlässt ein Peer das Netzwerk, so kann dies somit zur Folge haben, dass nur maximal η Spiegel aus dem Netzwerk verschwinden, die neu aufgestellt werden müssen. Somit verbessert diese Schranke zusätzlich das Verhalten des Verfahrens unter Churn.

4.4.4 Aufstellen und Pflegen von Spiegeln

Wie bereits in der Einleitung zu diesem Kapitel bemerkt, stand für die Implementierung von Adaptive-Chord eine funktionstüchtige Version von Chord zur Verfügung. Diese musste im Rahmen dieser Arbeit so erweitert werden, dass diese in der Lage ist, Spiegel für Dateien aufzustellen und zu verwalten. Konkret mussten folgende Funktionalitäten implementiert werden:

Erstellen von Spiegeln Beim Erstellen eines Spiegels kontaktiert ein Peer p einen anderen Peer q, um auf diesen eine Datei d auszulagern.

Entfernen von Spiegeln Beim Entfernen eines Spiegels kontaktiert ein Peer p einen anderen Peer q, auf den p bereits zuvor eine Datei d ausgelagert hat, und teilt diesem mit, den Spiegel für Datei d zu entfernen. Anschließend löscht q den Spiegel für Datei d.

Überwachen von Spiegeln Um sicher zu stellen, dass ein Spiegel weiterhin existiert, überwacht ein Peer p regelmäßig alle anderen Peers q, zu denen dieser vorher Dateien ausgelagert hat.

Aufgeben von Spiegeln Hat ein Peer p eine Kopie einer Datei d auf einem Peer q erstellt und q löscht diese Kopie nach einer gewissen Zeit, so gibt p diesen Spiegel auf. In dieser Situation muss p über diese Aktion informiert werden, damit dieser keine weiteren Anfragen für die Datei an q weiterleitet.

Im Weiteren werden die Konzepte der Implementierung dieser vier Funktionen beschrieben.

Erstellen von Spiegeln

Besitzt ein Peer p eine Datei d, die auf einen Peer q ausgelagert werden soll, so sendet p zunächst eine `createMirrorMessage` an p. Inhalt dieser `createMirrorMessage` ist der Bezeichner des Dokumentes, das auf q gespiegelt werden soll. Empfängt der Peer q die `createMirrorMessage`, so prüft dieser zunächst, ob es möglich ist, einen Spiegel für dieses Dokument zu erstellen. Ist dies der Fall, so startet q einen Dateidownload der Datei d von p und stellt diese ab diesem Zeitpunkt als Spiegel bereit. Als Antwort auf eine `createMirrorMessage` wird eine `createMirrorReplyMessage` versendet. Die `CreateMirrorReplyMessage` enthält den Bezeichner der Datei, die gespiegelt werden soll, und zusätzlich ein Attribut, das Auskunft darüber gibt, ob q einen Spiegel erstellt oder nicht. Empfängt p eine `CreateMirrorReplyMessage`, die darüber mitteilt, dass q eine Kopie von d erstellt, so markiert p die Datei d als an q ausgelagert. In der Folge werden alle Dateianfragen für d, die p erhält, an q weitergeleitet.

Entfernen von Spiegeln

Im Laufe des Adaptive-Chord Algorithmus kann es vorkommen, dass ein Peer p einen Spiegel, den dieser bereits auf einem Peer q erstellt hat, wieder entfernen möchte. Zu diesem Zweck entfernt p die Markierung der Datei d, die diese als ausgelagert markiert. In dieser Operation wird q nicht explizit über diese Maßnahme unterrichtet. q wird die gespiegelte Datei erst dann entfernen, wenn dieser eine Zeit lang keine Überwachungsnachrichten von p für die Datei d erhalten hat.

Überwachen von Spiegeln

Das Überwachen von Spiegeln erfüllt in dieser Implementierung gleich zwei Zwecke: Ein Peer p ist damit in der Lage, zu überprüfen, ob ein vorher aufgestellter Spiegel noch existiert, und Peers, auf die Dateien gespiegelt wurden, können durch die Überwachung prüfen, ob Spiegel noch gebraucht werden. Zur Überwachung von Spiegeln sendet Peer p in periodischen Abständen an alle Peers q_i, zu denen dieser Dateien ausgelagert hat, eine `CheckMirrorMessage`. Inhalt dieser Nachricht ist der Bezeichner der Datei d, die überprüft werden soll. Besitzt q_i einen Spiegel für die Datei d, so antwortet dieser mit einer `CheckMirrorReplyMessage`. Empfängt p auf mehrere `CheckMirrorMessage`s hintereinander keine entsprechende `CheckMirrorReplyMessage`, so gilt der Spiegel auf q_i als gelöscht. p wird in diesem Fall die Markierung auf der Datei d löschen. Empfängt q_i wiederum in einer gewissen Zeitspanne für einen Spiegel keine `CheckMirrorMessage`, so gilt der Spiegel als verwaist und wird von q_i gelöscht.

Aufgeben von Spiegeln

Sobald ein Peer q, auf den eine Datei d von einem Peer p gespiegelt wurde, diesen Spiegel entfernt, muss sichergestellt werden, dass p von dieser Aktion Kenntnis erhält. Da p alle Anfragen an den Peer q für die Datei d weiterleitet, ist es somit wichtig, dass p von der Löschung des Spiegels so schnell wie möglich unterrichtet wird. Löscht ein Peer q einen Spiegel oder verlässt das Netzwerk, so teilt q dies den entsprechenden Peers mit, indem q diesen eine `RemovedMirrorMessage` sendet. Diese beinhaltet den Bezeichner der Datei, die ab diesem Zeitpunkt nicht mehr als Spiegel bereitsteht. Empfängt ein Peer p eine `RemovedMirrorMessage`, so wird dieser die von der Nachricht betroffene Datei nicht länger als ausgelagert markieren.

Doch auch wenn ein Peer q das Adaptive-Chord-Netzwerk verlässt, ohne andere Peers darüber zu informieren, werden Peers, die Dateien auf q gespiegelt haben, dies nach einer gewissen Zeit durch die ausbleibenden Antworten auf die `CheckMirrorMessage`-Nachrichten bemerken.

5 Evaluation

5.1 Das Szenario

Um fundierte Aussagen über die Leistungsfähigkeit eines P2P-Overlays treffen zu können, ist es notwendig, dieses Overlay unter verschiedensten Arbeitsbelastungen zu betrachten. Ein P2P-Netzwerk bietet einer Anwendung den Dienst der Dateispeicherung an. Somit können in einem P2P-Netzwerk Dateien hinterlegt und anschließend unter einem bestimmten Bezeichner wieder abgerufen werden. Auf Grundlage dieses Dienstes ist es möglich, verschiedenste Anwendungen zu betreiben, die somit je nach Art und Nutzung der Anwendung dazu führen, dass das zu Grund liegende P2P-Netzwerk unterschiedlichste Arbeitsbelastungen erfährt.

Wie bereits in Kapitel 3.2.2 diskutiert, ist eine häufig dokumentierte Beobachtung, dass viele Anwendungen ein Dateianfragemuster induzieren, das sich gut durch eine Zipf-Verteilung modellieren lässt. Daher werden im Rahmen dieser Evaluation Szenarien mit einem solchen Dateianfragemuster untersucht. Die Dateianfragemuster allein sind jedoch nur ein Teil eines Szenarios. So wird die auftretende Last vor allem durch die Häufigkeit der Dateianfragen und durch die Größe der Dateien bestimmt.

In diesem Kapitel wird zunächst das Modell erläutert, mit dessen Hilfe die P2P-Netzwerke untersucht werden. Dazu wird erst die Simulationsumgebung vorgestellt, mit der die in diesem Kapitel vorgestellten Experimente durchgeführt werden. Anschließend werden die einzelnen Parameter eines Szenarios beschrieben, die in der Simulation verändert werden können. Danach werden verschiedene Konfigurationen dieser Parameter zusammengefasst zu einzelnen Szenarien, mit denen die P2P-

Netzwerke letztendlich evaluiert werden.

5.1.1 Modell

Als Simulationsumgebung für die Evaluation dient der P2P-Simulator *Peerfact-Sim.KOM* [KKL+07, Gra11]. Bei PeerfactSim.KOM handelt es sich um einen diskreten, Event-basierten Simulator, mit dem es möglich ist, detaillierte Simulationen von Peer-to-Peer-Netzwerken durchzuführen. Sein modularer Aufbau ermöglicht es, verschiedenste Konfigurationen für eine Simulation zu erstellen, weshalb sich PeerfactSim.KOM hervorragend für die in dieser Arbeit benötigte Simulation eignet.

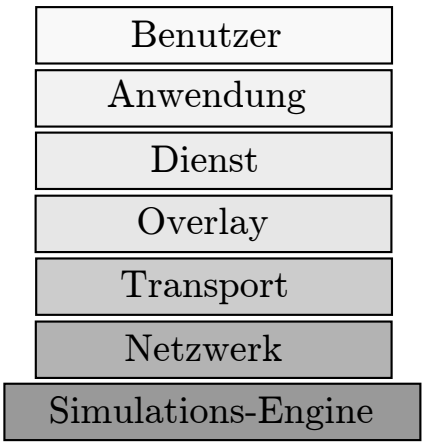


Abbildung 5.1: Schichtenarchitektur von PeerfactSim.KOM.

Abbildung 5.1 zeigt den modularen Aufbau von PeerfactSim.KOM, in dem jede Schicht separat konfiguriert und gegebenenfalls auch ausgetauscht werden kann. Direkt über der *Simulations-Engine* liegt die *Netzwerk-Schicht*. Für diese Schicht stehen in PeerfactSim.KOM zwei von Grund auf verschiedene Implementierungen zur Verfügung: der `SimpleNetLayer` und der `GNPNetworkLayer` (vgl. [PSK]). Der `SimpleNetLayer` stellt ein sehr einfaches Modell zur Berechnung des Verhaltens der Netzwerkschicht zur Verfügung. Dieses kann sehr effizient berechnet werden, ist jedoch weit von dem Verhalten einer realen Netzwerk-Schicht entfernt. Dem gegenüber steht der `GNPNetLayer`. Der `GNPNetLayer` versucht, die typischen Charakteristiken des in der Gegenwart zu findenden Internets nachzubilden [PSK]. Dazu setzt der `GNPNetLayer` auf empirischen Studien der sogenannten *Round-Trip Time*, des *Jitter*, der *geographischen Positionen* und des *Packet loss* auf. Um das *GNP-Modell* mit Daten zu versorgen, wurden daher verschiedenste Messungen der

Projekte *PingER*[1], *CAIDA*[2] und *MaxMind*[3] zusammengetragen, die zusammen in das *GNP-Modell* [KPL+09, KN07] einflossen.

Auf der Netzwerk-Schicht aufsetzend existiert die *Transport-Schicht*. Diese ermöglicht es, in PeerfactSim.KOM Nachrichten mit dem *TCP*- als auch mit dem *UDP-Protokoll* zu versenden. Diese Nachrichten werden von dem verwendeten *Overlay* verschickt. In diesem Fall sind das die drei, in dieser Arbeit vorgestellten P2P-Overlays: Chord, Re-Chord und Adaptive-Chord. Diese drei Overlays stellen, wie in Kapitel 2.3 diskutiert, einen *Dienst* zur Verfügung, der es erlaubt, zu einem Schlüssel einen beliebigen Datensatz abzulegen, um anschließend mit Hilfe des Schlüssels wieder auf diesen Datensatz zugreifen zu können. Dieser Dienst wird von der *Anwendungs-Schicht* genutzt, die wiederum von einem *Benutzer* kontrolliert wird. Bei der einzigen Implementierung einer Anwendung in PeerfactSim.KOM handelt es sich um `filesharing2`. Einer Anwendung, mit der Dateien publiziert, gesucht und kopiert werden können.

Die Last, die im P2P-Netzwerk generiert wird, wird stark von der Art der Anwendung und vor allem von dem Verhalten der Benutzer beeinflusst. Daher ist es in PeerfactSim.KOM möglich, detaillierte Einstellungen sowohl an der Anwendung als auch an dem Verhalten der Benutzer vorzunehmen. Das durch den Benutzer gesteuerte Verhalten der Anwendung vereint dabei die Dauer, die jeder einzelne Teilnehmer im Netzwerk verbleibt, die Häufigkeit, mit der Dateianfragen ausgeführt werden, und die Dateianfragemuster. Zusätzlich bestimmt die Art der Anwendung die zu übertragenden Dateigrößen maßgeblich.

5.1.2 Konfigurationen der einzelnen Szenarien

Damit die Qualität der vorliegenden P2P-Overlays unter möglichst unterschiedlichen Arbeitsbelastungen verglichen werden können, werden in diesem Kapitel verschiedene Parametrisierungen für die Anwendungs-Schicht als auch das Verhalten des Benutzers vorgestellt. Diese Konfigurationen werden anschließend in den Simulationen benutzt, um die P2P-Overlays zu bewerten. Zunächst werden dafür die Parameter beschrieben, die verändert werden können, anschließend werden verschiedene Belegungen für diese Parameter zu einzelnen Szenarien zusammengefasst.

Dateigröße Je größer eine Datei ist, desto mehr Daten müssen bei einem Zugriff auf die Datei übertragen werden. Die Dateigröße hat somit direkten Einfluss auf die entstehende Last.

Anfragezeit Die Anfragezeit beschreibt die Zeit, die zwischen zwei Dateianfragen,

[1] `http://www-iepm.slac.stanford.edu/pinger/`
[2] `http://www.caida.org/home/`
[3] `http://www.maxmind.com/app/ip-location`

ausgehend vom gleichen Peer, vergeht. Je kürzer dieses Intervall, desto mehr Last entsteht im Netzwerk.

Veränderung der Popularität Die Veränderung der Popularität beschreibt die Zeitspanne, in der die Popularität einer Datei gleich bleibt. Um die Veränderung der Popularität über die Zeit zu simulieren, ist es in PeerfactSim.KOM möglich, den Rang (siehe hierzu Kapitel 3.2.2) einer Datei nach einer gewissen Zeitspanne mit einer zufällig ausgewählten anderen Datei zu vertauschen.

Churn Das Churn beschreibt den Prozess des Beitretens und Verlassens des Netzwerkes durch Peers. Die dadurch entstehende Fluktuation hat negative Auswirkungen auf die Stabilität und die Leistungsfähigkeit des Netzwerkes.

Dateianfragemuster Das Dateianfragemuster beschreibt die Popularitätsmuster der einzelnen Dateien, wie dies in Kapitel 3.2.2 erläutert wurde.

Allen hier erstellten Konfigurationen gleich ist der Ablauf, der simuliert wird: Jedes Szenario wird für eine Dauer von 24 Stunden betrachtet. In den ersten 40 Minuten dieser 24 Stunden werden dem Netzwerk 1000 Peers hinzugefügt. Danach werden für 20 Minuten die Stabilisationsmethoden der jeweiligen Overlays ausgeführt, ohne dabei von Aktionen wie Churn oder dem Publizieren von Dateien gestört zu werden. Dadurch gewinnt das Netzwerk an Stabilität, da die Routingtabellen mit korrekten Einträgen gefüllt werden. In den nächsten 30 Minuten werden 10000 Dokumente im Netzwerk publiziert. Anschließend werden die Peers damit beginnen, Dateien anzufragen und diese zu kopieren. Vergleiche hierzu Abbildung 5.2.

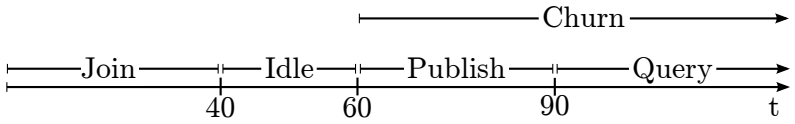


Abbildung 5.2: Zeitlicher Ablauf eines Szenarios.

Die Dateigrößen der publizierten Dateien werden in allen Szenarien gleichverteilt zwischen 0 und 10 Megabyte gewählt werden. Die Verteilung, mit der bestimmt wird, wann welcher Peer das Netzwerk verlässt und wieder betritt, wird durch eine *Weibull-Verteilung* bestimmt. Dieses Modell des Churn ist bereits in PeerfactSim.KOM implementiert und setzt ein Modell um, das auf den Daten der Studie [SENB09] beruht. Die Popularität der einzelnen Dateien für die Dateianfragen wird Zipf-verteilt modelliert. Hierfür wird die Zipf-Verteilung $p(x) \sim \left(\frac{1}{Rang(x)}\right)^{0.6}$ verwendet. Diese Wahrscheinlichkeitsverteilung ist in Abbildung 5.3 dargestellt. Ein solches Szenario könnte aufgrund der Verteilungseigenschaften und des verwendeten

Name	Anfragezeit	Pop. Änderung	Churn
busy_unstable_nochurn	10 Min	10 Min	nein
idle_unstable_nochurn	30 Min	10 Min	nein
busy_stable_nochurn	10 Min	60 Min	nein
idle_stable_nochurn	30 Min	60 Min	nein
busy_unstable_churn	10 Min	10 Min	ja
idle_unstable_churn	30 Min	10 Min	ja
busy_stable_churn	10 Min	60 Min	ja
idle_stable_churn	30 Min	60 Min	ja

Tabelle 5.1: In dieser Auswertung betrachtete Szenarien.

Bereiches der Dateigröße die Arbeitsbelastung eines P2P-Netzwerkes beschreiben, das vorwiegend aus Heimcomputern besteht und in dem Audio-Dateien und kürzere Video-Sequenzen gespeichert und abgerufen werden.

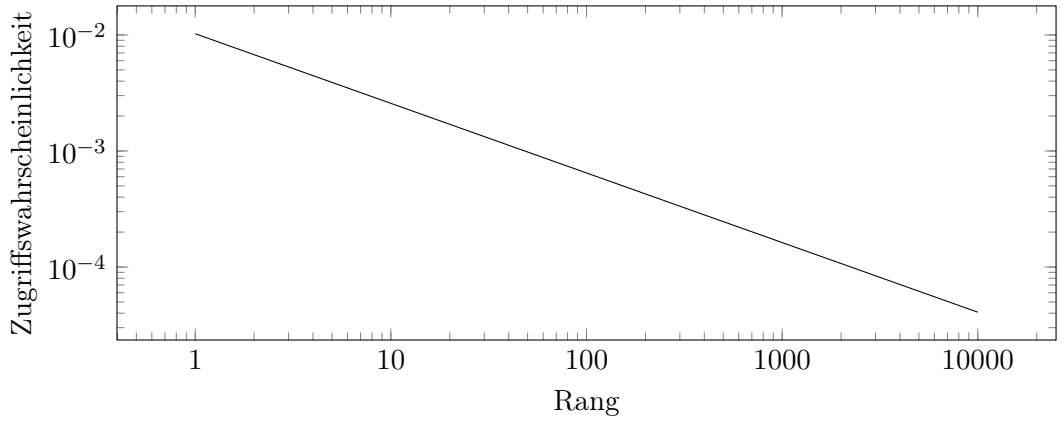


Abbildung 5.3: Zipf-Verteilung mit $p(x) \sim \left(\frac{1}{Rang(x)}\right)^{0.6}$.

Tabelle 5.1 zeigt die in dieser Arbeit betrachteten Szenarien: Durch die Veränderung der Anfragezeit werden mehr bzw. weniger häufige Dateianfragen simuliert. Dadurch, dass die Zeitspanne, in der sich die Popularität verändert, variiert wird, kann beobachtet werden, welche Auswirkungen eine länger stabil bleibende Anfrage auf die durchschnittliche Downloadgeschwindigkeit einer Datei besitzt. Zusätzlich wird jedes Szenario einmal mit Churn und einmal ohne Churn simuliert. Dies geschieht, damit die Effekte, die durch das Adaptive-Chord-Verfahren entstehen, besser von den Effekten, die durch das Churn auftreten, separiert werden können.

5.1.3 Parametrisierung der Algorithmen

Für die in dieser Arbeit durchgeführten Experimente wurde für jedes Overlay eine feste Konfiguration von Parametern benutzt, die für alle Experimente verwendet wird. Diese Parameter werden in diesem Abschnitt dargestellt. Allen Overlays gleich ist die Einstellung, dass eine Nachricht, die mehr als 50 Hops auf dem Overlay-Netzwerk zurückgelegt hat, nicht mehr weitergeleitet und somit fallengelassen wird. Zusätzlich gilt eine Nachricht als verloren, wenn auf diese nicht innerhalb von 10 Sekunden geantwortet wurde.

Chord

Das Intervall zwischen den von Chord verwendeten Stabilisierungsmethoden beträgt in den nachfolgenden Simulationen jeweils 30 Sekunden. Somit wird von jedem Peer alle 30 Sekunden der direkte Nachfolger überprüft, ein Finger-Eintrag in der Routingtabelle aktualisiert und die Liste der zusätzlichen direkten Nachfolger, wie diese in Kapitel 2.4.6 beschrieben ist, erneuert. Dabei hält jeder Peer 3 zusätzliche Nachfolger in der Liste.

Re-Chord

Im Re-Chord-Netzwerk wird auf jedem Peer alle 30 Sekunden die Stabilisationsmethode ausgeführt. Diese besitzt keine Parameter, die konfiguriert werden können.

Adaptive-Chord

Da im Kapazitätsnetzwerk von Adaptive-Chord zusätzlich zum natürlichen Churn noch künstlicher Churn durch die ständige Veränderung der Bezeichner besteht, werden in diesem Netzwerk die Stabilisierungsmethoden häufiger ausgeführt, als in dem Chord-Netzwerk. Im Kapazitätsnetzwerk wird alle 10 Sekunden der direkte Nachfolger überprüft, alle 20 Sekunden ein Finger-Eintrag aktualisiert und alle 10 Sekunden die Liste der direkten Nachfolger erneuert. Durch das hohe Maß an Churn hält in diesem Netzwerk jeder Peer 10 Einträge in der Liste der direkten Nachfolger. Für das Datennetzwerk wird, analog zu der Konfiguration von Chord, jede Stabilisierungsmethode alle 30 Sekunden ausgeführt, und die Tabelle der direkten Nachfolger enthält 3 Peers.

Für den Lastbalancierungsalgorithmus wurden die folgenden Parameter gewählt: Die untere Schranke λ, die angibt, wie viele Ressourcen ein Peer mindestens besitzen muss, damit dieser nicht pauschal als überlastet bewertet wird, beträgt $\lambda = 70\ KByte/s$, der initiale Wert für ω beträgt 150 $KByte/s$. Die anteilige Sicherheit bei der Rücknahme von Spiegeln, γ, beträgt 0.8, wohingegen κ als $0.2 \cdot v_{max}$ gewählt wurde. Um dem künstlichen Churn durch das Verändern der Bezeichner

entgegenzuwirken, wurde $\tau = 0.3$ gesetzt. Der Lastbalancierungsalgorithmus wird alle 4 Minuten von jedem Peer ausgeführt.

5.2 Kenngrößen

Um die vorgestellten Overlays miteinander zu vergleichen, werden verschiedene Kenngrößen benötigt, auf Grundlage derer die Simulationen bewertet werden können. Diese Kenngrößen sollten die Qualität des vom Overlay zur Verfügung gestellten Dienstes möglichst gut widerspiegeln, sodass Veränderungen der Qualität, die von einem Endnutzer spürbar sind, einen unmittelbaren Einfluss auf die betrachteten Kenngrößen haben. Daher werden im Rahmen dieser Arbeit die Kenngrößen *Dauer eines Dateitransfers*, *Geschwindigkeit einer Datenübertragung* und die *Erfolgsrate der Dateiübertragungen* betrachtet. Zusätzlich zu diesen Größen wird zu jedem Zeitpunkt die jeweils aktuelle Anzahl an überlasteten Peers ermittelt. Bei dieser Messung gilt ein Peer als überlastet, wenn dieser im Durchschnitt mehr als 90% seiner verfügbaren Uploadkapazität nutzt. Als *Dauer eines Dateitransfers* wird dabei die Zeit definiert, die zwischen dem Absenden der Dateianfrage und dem Empfangen des letzten Bits der Datei vergeht. Die *Geschwindigkeit einer Datenübertragung* ist die durchschnittliche Geschwindigkeit, mit der die angefragte Datei übertragen wird. Eine Dateiübertragung gilt als fehlgeschlagen, wenn entweder die gesuchte Datei nicht im Netzwerk gefunden wird oder die Übertragung der Nutzdaten der angeforderten Datei abbricht, bevor diese abgeschlossen wurde.

Da diese Kenngrößen jeweils bei einem jeden Peer des Netzwerkes messbar sind, werden von diesen Werten das arithmetische Mittel sowie die Standardabweichung betrachtet. Diese werden dabei wie folgt bestimmt: In regelmäßigen Abständen von 5 Minuten wird – um die zeitliche Entwicklung der Kenngrößen über den Simulationszeitraum zu betrachten – jeweils für jeden Peer ein Messwert ermittelt. Dieser Messwert spiegelt dabei alle Daten, bis zurück zu dem Zeitpunkt, an dem der letzte Messwert genommen wurde, wider. Von diesen Messwerten wird für jeden Peer das arithmetische Mittel gebildet, das wiederum in das Gesamtergebnis des Netzwerkes eingeht. Dieses Gesamtergebnis spiegelt somit eine Kenngröße für die vergangenen 5 Minuten des Netzwerkes wider.

Somit berechnet sich beispielsweise die durchschnittliche Dauer d eines Dateitransfers im Netzwerk als

$$d = \frac{\sum_{p \in \mathcal{P}} F_p}{|\mathcal{P}|} \quad ,$$

wobei $\mathcal{P}$ die Menge aller Peers und F_p die durchschnittliche Dauer eines Dateitransfers (der in den letzten 5 Minuten beendet wurde) zu Peer p darstellt. Analog be-

rechnet sich die Standardabweichung $\sigma(d)$ als

$$\sigma(d) = \sqrt{\frac{\sum_{p \in \mathcal{P}} (d - F_p)^2}{|\mathcal{P}|}} \quad .$$

5.3 Auswertung

In diesem Kapitel werden die Ergebnisse der Evaluation vorgestellt. Dazu wurden alle Szenarien, wie diese in Tabelle 5.1 aufgelistet sind, mit einem Chord-, Re-Chord- und einem Adaptive-Chord-Netzwerk simuliert. Dabei wurden in regelmäßigen Abständen die in Kapitel 5.2 vorgestellten Kenngrößen ermittelt und protokolliert. Auf Grundlage dieser Daten werden Rückschlüsse auf die Qualität der entsprechenden Netzwerke gezogen und diese anschließend miteinander verglichen. Zunächst werden dazu die Szenarien ohne Churn betrachtet, anschließend mit Churn.

5.3.1 Szenarien ohne Churn

Die nachfolgenden Szenarien ohne Churn stellen eine idealisierte Weltsicht dar, in der Teilnehmer, die einmal das Netzwerk betreten haben, dieses nicht wieder verlassen. Jeder Teilnehmer ist somit immer im Netzwerk erreichbar. Dies hat einige Vorteile im Bezug auf die Auswertung der Szenarien: So werden die in den verschiedenen Overlays realisierten Strategien verschiedene Effekte erzielen, die sich auch in den betrachteten Kenngrößen wiederfinden lassen. Diese Effekte können jedoch durch das Churn beeinflusst werden, bzw. die Effekte, die das Churn auf die Kenngrößen hat, können andere Effekte überlagern, sodass diese nur verfälscht betrachtet werden können. Daher wird jedes Szenario in dieser Arbeit einmal ohne und einmal mit Churn betrachtet. Somit können die entstehenden Effekte unverfälscht und unabhängig von Churn in einem sich nicht verändernden Netzwerk betrachtet werden.

5.3.2 Szenario busy_unstable_nochurn

Das Szenario busy_unstable_nochurn beschreibt eine Anwendung, in der die Frequenz der Dateianfragen gleich ist mit der Frequenz der Änderung der Popularität der Daten (vgl. Tabelle 5.1). Damit wird erreicht, dass sich keine Dateien herausbilden, die über eine längere Zeit gesehen dafür sorgen, dass einzelne Peers dauerhaft stark belastet werden. In diesem Szenario existiert kein Churn, weshalb jeder Peer die gesamte Zeit im Netzwerk erreichbar ist.

Abbildung 5.4 zeigt die durchschnittliche Dauer eines abgeschlossenen Dateitransfers im Netzwerk. Wie zu erkennen ist, unterscheidet sich die Dauer der Downloads in diesem Szenario zwischen Chord und Re-Chord nicht. Die durchschnittliche Zeit,

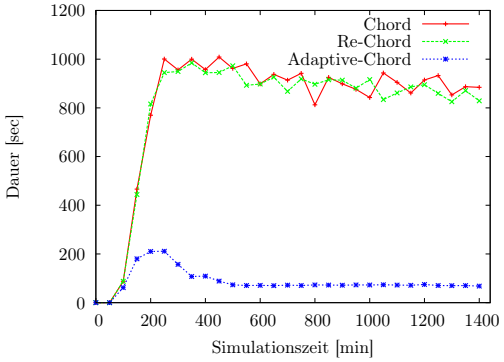


Abbildung 5.4: Durchschnittliche Dauer eines Downloads.

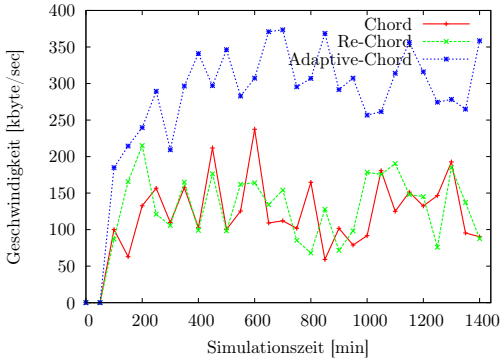


Abbildung 5.5: Durchschnittliche Geschwindigkeit eines Downloads.

die ein Download benötigt, ist im Adaptive-Chord-Netzwerk jedoch um ein Vielfaches geringer als in den anderen beiden Netzwerken. So pendelt sich diese bei Adaptive-Chord um 85 Sekunden für einen Dateitransfer ein, wohingegen dies bei Chord und Re-Chord im Mittel über 790 Sekunden beansprucht.

Werden die durchschnittlichen Downloadgeschwindigkeiten in Abbildung 5.5 betrachtet, so ist ein ähnliches Bild zu erkennen, wie bereits bei der Betrachtung der durchschnittlichen Dauer eines Transfers: Die Geschwindigkeit bei Chord und bei Re-Chord beträgt im Mittel 118 KByte bzw. 125 KByte pro Sekunde, wohingegen die durchschnittliche Geschwindigkeit eines Dateitransfers bei Adaptive-Chord bei 275 KByte pro Sekunde liegt.

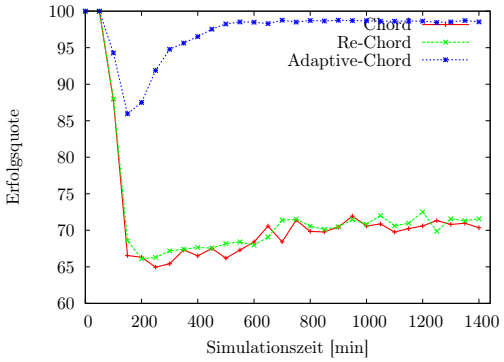


Abbildung 5.6: Durchschnittliche Erfolgsrate eines Downloads.

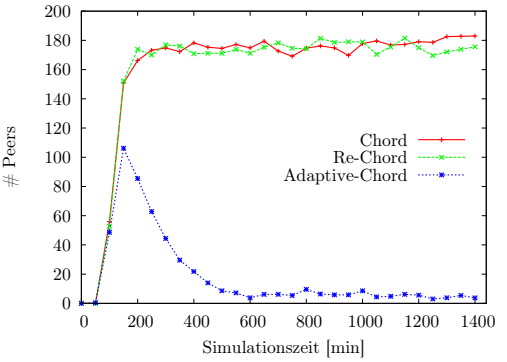


Abbildung 5.7: Anzahl der überlasteten Peers.

Gemeinsam mit Abbildung 5.6 und Abbildung 5.7 kann der Einfluss der überlasteten Peers auf die Erfolgsrate der Dateitransfers erkannt werden: Unterschreitet ein Dateitransfer über einen längeren Zeitraum die Geschwindigkeit von 1 KByte pro

Sekunde, so wird dieser Transfer abgebrochen und taucht in der Statistik, die in Abbildung 5.6 zu sehen ist, als fehlgeschlagener Download auf. Da das Adaptive-Chord-Netzwerk in diesem Szenario so gut wie keine überlasteten Peers aufweist, ist auch die Erfolgsrate der Dateiübertragungen nah an 100%. Lediglich um $t = 150\ min$ bricht diese Quote kurzzeitig ein, da das Adaptive-Chord-Netzwerk zu diesem Zeitpunkt noch nicht genügend Dateien auf starke Peers ausgelagert hat. Zusätzlich kann in Abbildung 5.7 deutlich erkannt werden, dass die Lastbalancierung des Adaptive-Chord-Overlays die Anzahl an überlasteten Peers senkt. So existieren in diesem Szenario nach $t = 600\ min$ im Adaptive-Chord-Netzwerk nur noch sehr sporadisch Peers, die überlastet sind.

5.3.3 Szenario idle_unstable_nochurn

Das Szenario idle_unstable_nochurn ist identisch mit dem vorangegangenen Szenario busy_unstable_nochurn, zeichnet sich jedoch durch eine dreimal geringere Anfragehäufigkeit aus. In der Folge sinkt das Datenaufkommen im Netzwerk und auch die Anzahl an überlasteten Peers geht in allen betrachteten Netzwerken im Vergleich merklich zurück.

Diese Beobachtungen spiegeln sich bereits in Abbildung 5.8 wider: Durch die verringerte Last im Netzwerk sinkt auch bei Chord und Re-Chord die durchschnittliche Dauer eines Dateitransfers im Mittel auf einen Wert von 620 bzw. 576 Sekunden.

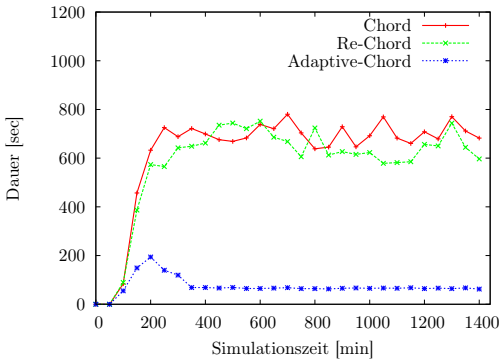


Abbildung 5.8: Durchschnittliche Dauer eines Downloads.

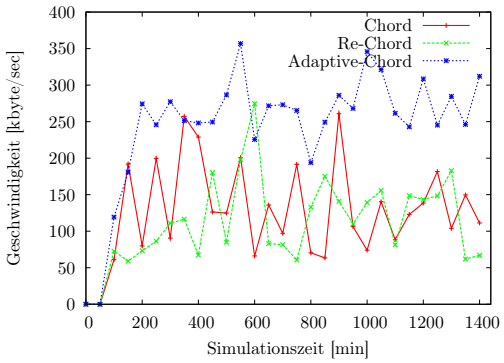


Abbildung 5.9: Durchschnittliche Geschwindigkeit eines Downloads.

Wird Abbildung 5.11 betrachtet, fällt auf, dass, wie eingangs behauptet, die Anzahl der überlasteten Peers in Chord sowie Re-Chord auf Grund der gesunkenen Frequenz der Dateianfragen ebenfalls sinkt. Daraus resultierend können die Peers Dateien schneller ausliefern, was dazu führt, dass, wie in Abbildung 5.10 zu sehen, die Erfolgsquote der Dateiübertragungen deutlich steigt.

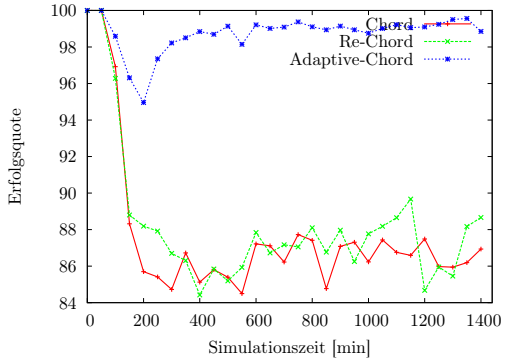


Abbildung 5.10: Durchschnittliche Erfolgsrate eines Downloads.

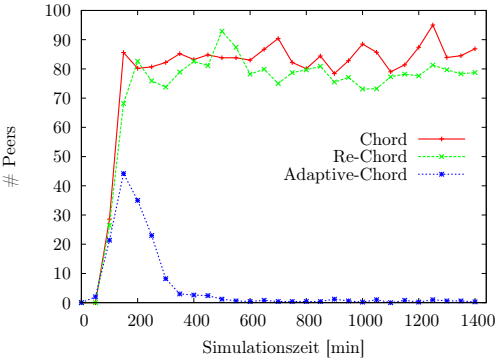


Abbildung 5.11: Anzahl der überlasteten Peers.

Bei allen betrachteten Statistiken fällt auf, dass Adaptive-Chord in diesem Szenario, in dem die Last im Netzwerk gering ist, überlegen gegenüber Chord und Re-Chord ist. Dies ist der Tatsache zuzuschreiben, dass in Adaptive-Chord die Peers mit geringer Uploadkapazität alle Dateien an stärkere Peers auslagern, auch wenn diese nicht akut überlastet sind. Dies führt dazu, dass diese Dateien innerhalb des Netzwerkes schneller ausgeliefert werden können.

5.3.4 Szenario busy_stable_nochurn

Mit dem Szenario busy_stable_nochurn wird eine Situation untersucht, in der eine hohe Frequenz an Dateianfragen pro Peer existiert, die Popularität der einzelnen Dateien jedoch nur einer sehr langsamen Veränderung unterliegen. Daraus resultierend existieren Dateien im Netzwerk, die über eine lange Zeit sehr stark angefragt werden, und somit erzeugen diese Dateien auch eine sehr hohe Last.

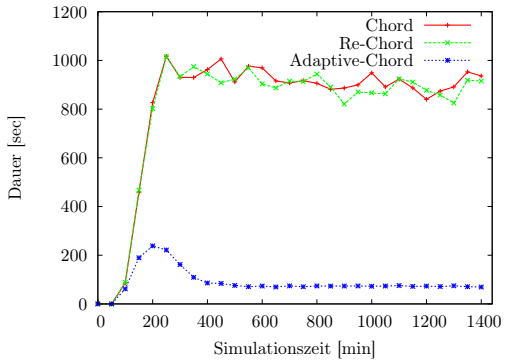


Abbildung 5.12: Durchschnittliche Dauer eines Downloads.

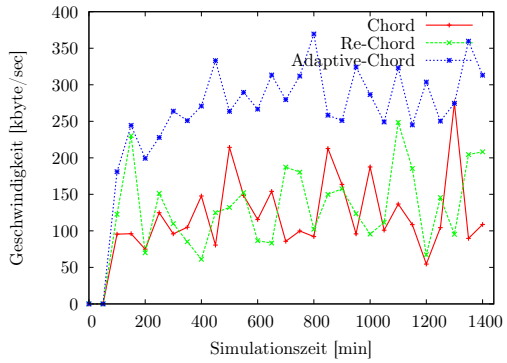


Abbildung 5.13: Durchschnittliche Geschwindigkeit eines Downloads.

Wird Abbildung 5.12 betrachtet, so ist zu erkennen, dass die durchschnittliche Dauer eines Dateitransfers im Chord- und dem Re-Chord-Netzwerk bei über 801 bzw. 796 Sekunden liegt, wohingegen dies bei Adaptive-Chord im Durchschnitt nur 87 Sekunden beansprucht. Im Gegensatz zum Szenario busy_unstable_nochurn ist somit die durchschnittliche Dauer eines Dateitransfers im Chord- sowie dem Re-Chord-Netzwerk leicht gestiegen, wohingegen diese im Adaptive-Chord-Netzwerk beinahe gleich blieb. Das gleiche Verhalten ist in Abbildung 5.13 zu beobachten, wo die durchschnitlichen Geschwindigkeiten der Dateitransfers gegenübergestellt werden: Auch hier ist zu erkennen, dass diese in allen drei Netzwerken im Vergleich zum Szenario busy_unstable_nochurn leicht gefallen sind.

Analog verhält es sich mit der Anzahl der überlasteten Peers, diese steigt in diesem Szenario im Durchschnitt leicht an, vgl. Abbildung 5.15. Dieser leichte Anstieg bewirkt jedoch bei der in Abbildung 5.14 zu sehenden Erfolgsrate der Dateitransfers keine signifikanten Veränderungen.

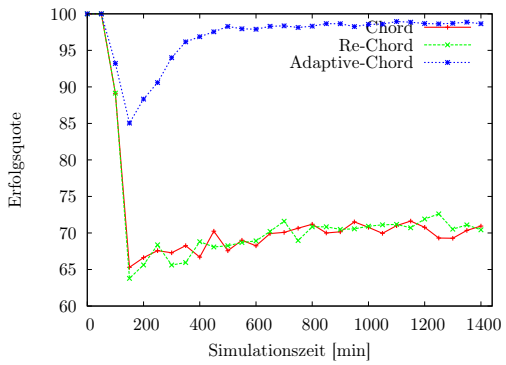


Abbildung 5.14: Durchschnittliche Erfolgsrate eines Downloads.

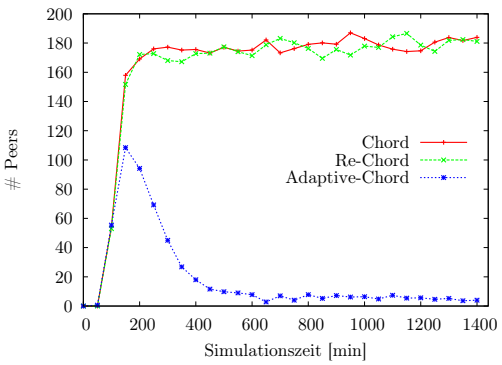


Abbildung 5.15: Anzahl der überlasteten Peers.

Auch in diesem Szenario wird die Überlegenheit von Adaptive-Chord gegenüber Chord und Re-Chord deutlich: Die Dauer eines Dateitransfers ist im Schnitt im Adaptive-Chord-Netzwerk über 9-mal schneller als in dem Chord- sowie Re-Chord-Netzwerk. Und auch in diesem Szenario ist der Zusammenhang zwischen der Anzahl von überlasteten Peers und der Erfolgsrate der Dateitransfers deutlich zu erkennen: Nachdem ab $t = 600$ das Adaptive-Chord-Netzwerk kaum über überlastete Peers verfügt, liegt die Erfolgsquote der Dateitransfers sehr nah an 100%, wohingegen diese Quote, als die Anzahl der überlasteten Peers ihren Höchststand in Adaptive-Chord besaß, auf ihren niedrigsten Wert von 85% fiel.

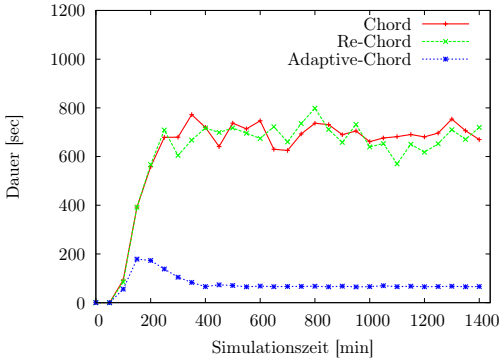


Abbildung 5.16: Durchschnittliche Dauer eines Downloads.

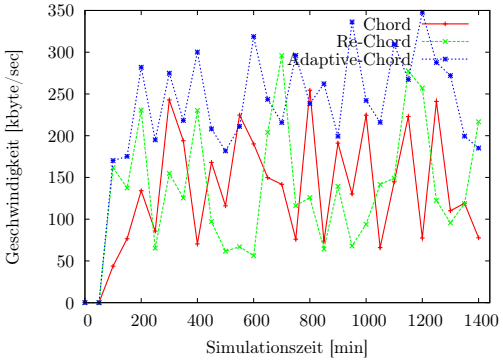


Abbildung 5.17: Durchschnittliche Geschwindigkeit eines Downloads.

5.3.5 Szenario idle_stable_nochurn

Der Unterschied zu dem vorangegangenen Szenario liegt bei idle_stable_nochurn darin, dass die Last im Netzwerk aufgrund der geringen Nachfrage der Peers niedrig ist. Dies ist insbesondere daran zu erkennen, dass die in Abbildung 5.19 zusehende Anzahl an überlasteten Peers in Chord als auch in Re-Chord im Vergleich zum Szenario busy_stable_nochurn deutlich gesunken ist.

Wird die durchschnittliche Dauer eines Dateitransfers betrachtet, so fällt auf, dass diese – im Vergleich zum Szenario busy_stable_nochurn – signifikant gesunken ist. Auch die in Abbildung 5.17 zu sehende durchschnittliche Geschwindigkeit der Dateiübertragungen in Chord und Re-Chord ist im Durchschnitt in diesem Szenario höher als noch in den vorangegangenen Szenarien. Jedoch erzielt Adaptive-Chord auch in diesem Szenario in der Dauer der Transfers als auch in der Geschwindigkeit der Übertragung deutlich bessere Ergebnisse als Chord und Re-Chord.

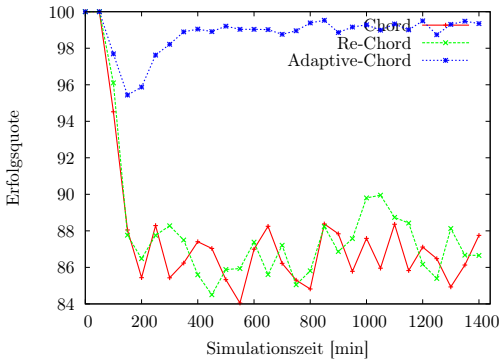


Abbildung 5.18: Durchschnittliche Erfolgsrate eines Downloads.

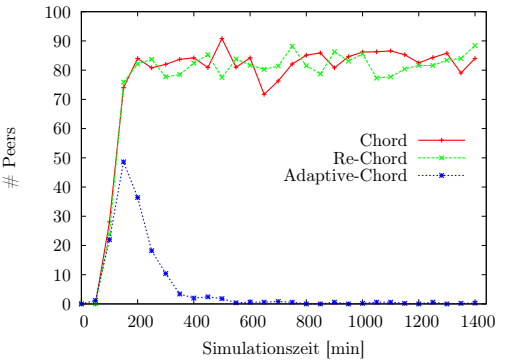


Abbildung 5.19: Anzahl der überlasteten Peers.

Dadurch, dass, wie in Abbildung 5.19 zu sehen, die Anzahl der überlasteten Peers in Chord als auch in Re-Chord geringer als im vorherigen Szenario ausfallen, stieg auch die Erfolgsquote der Dateiübertragungen entsprechend an (vgl. Abbildung 5.18). In diesem Szenario existieren im Adaptive-Chord-Netzwerk nach einer gewissen Zeit keine Peers, die als überlastet eingestuft werden, weshalb die Erfolgsquote von Adaptive-Chord auch in diesem Szenario über der von Chord und Re-Chord liegt.

5.3.6 Szenarien mit Churn

Durch die in den vorangegangenen Auswertungen betrachteten Szenarien ohne Churn konnte eine wichtige Beobachtungen gemacht werden: Es ist deutlich zu erkennen, dass sich ohne Churn die betrachteten Kenngrößen bei den Chord- und Re-Chord-Netzwerken sehr ähnlich verhalten. So ist in keinem Szenario ein klarer Unterschied in der Entwicklung der Kenngrößen von Chord- und Re-Chord über die Zeit festzustellen. Daraus kann der Schluss gezogen werden, dass das durch die Stabilisationsmethoden entstehende Netzwerk von Re-Chord dem Chord-Netzwerk in seiner Qualität sehr ähnlich sein muss. Zusätzlich kann gefolgert werden, dass die Nachrichten, die durch die Stabilisationsmethoden von Re-Chord versendet werden, keinen signifikant negativen Einfluss auf die Qualität des bereitgestellten Dienstes haben.

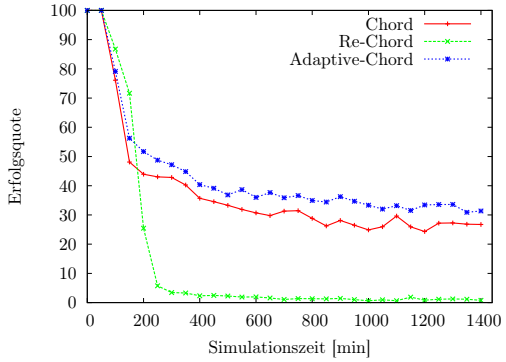


Abbildung 5.20: Typische Entwicklung der durchschnittlichen Erfolgsrate eines Downloads in einem Szenario mit Churn.

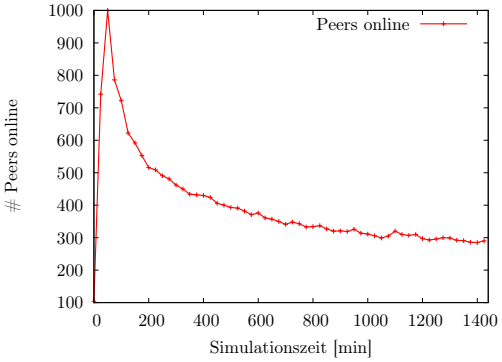


Abbildung 5.21: Anzahl an Peers, die aktiv am Netzwerk teilnehmen.

Die Stabilisierungsmethoden von Re-Chord werden in diesen Experimenten alle 30 Sekunden ausgeführt, was dazu führt, dass einzelne Peers bis zu 100 Service-Nachrichten pro Sekunde nur aufgrund der Stabilisierungsmethoden von Re-Chord versenden. Im Durchschnitt müssen die Teilnehmer eines Re-Chord-Netzwerkes so 6 Service-Nachrichten pro Sekunde versenden, vgl. Abbildung 5.22. Dies entspricht

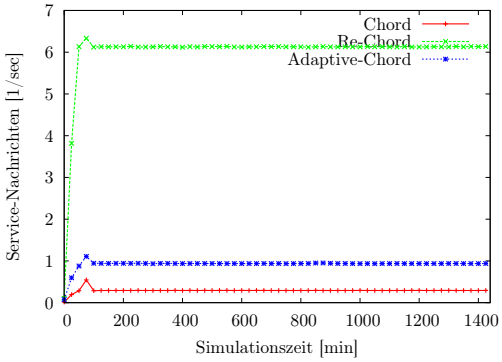


Abbildung 5.22: Anzahl der Service-Nachrichten, die nötig sind, um die Struktur des jeweiligen Netzwerkes aufrechtzuerhalten.

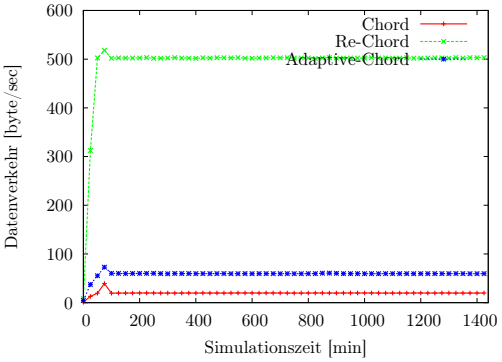


Abbildung 5.23: Datenverkehr der Service-Nachrichten, die nötig sind, um die Struktur des jeweiligen Netzwerkes aufrechtzuerhalten.

einem durchschnittlichen Datenaufkommen von 500 Byte pro Sekunde (siehe Abbildung 5.23). Würden die Stabilisierungsmethoden häufiger ausgeführt werden, so hätte dies zur Folge, dass die Leistungsfähigkeit von schwachen Peers, die mit langsamen Verbindungen an das Netzwerk angeschlossen sind, auf Grund der Stabilisierungsnachrichten gemindert wäre. Bei den Experimenten mit Szenarien mit Churn wurde jedoch festgestellt, dass ein Intervall von je 30 Sekunden nicht ausreichend ist, um das Re-Chord-Netzwerk in einem Zustand zu halten, in dem dieses ausreichend verbunden ist, um Dateianfragen zu routen. Dies wird besonders deutlich, wenn Abbildung 5.20 betrachtet wird. Die Erfolgsrate für Dateitransfers sinkt innerhalb von kurzer Zeit auf einen Wert unter 10% und geht von dort aus strikt gegen 0%. Die Stabilisierungsmethoden von Re-Chord können auch in den folgenden Stunden das Netzwerk nicht ausreichend wiederherstellen, damit die Erfolgsquote steigt. Doch auch ein Ausführungsintervall der Stabilisierungsmethoden von 10 Sekunden zeigt keine signifikante Besserung der Erfolgsquote. Da sich dadurch jedoch die Anzahl von versendeten Nachrichten jeweils verdreifacht, wird das Ausführungsintervall in dieser Arbeit nicht noch weiter gesenkt, da es durch den Overhead, der durch die Stabilisations-Nachrichten entsteht, bereits bei einem Intervall von 10 Sekunden schwachen Peers nicht mehr möglich ist, ausreichend Nachrichten zu versenden, um am Netzwerk teilnehmen zu können.

Daher wird im Rahmen dieser Arbeit bei den nachfolgenden Szenarien Re-Chord nicht weiter betrachtet. Es wird jedoch darauf hingewiesen, dass, mit einem ausreichend kurzen Intervall für die Stabilisierungsmethoden von Re-Chord, die Ergebnisse, die Re-Chord erzielt, nicht signifikant von denen von Chord abweichen werden. Dies jedoch nur unter der Voraussetzung, dass die Menge an Stabilisations-Nachrichten keine negativen Auswirkungen auf die Leistungsfähigkeit des Netzwer-

kes haben wird. Letzteres ist jedoch nur der Fall, wenn die Teilnehmer des Netzwerkes über eine entsprechende Netzanbindung verfügen.

Abbildung 5.21 zeigt die Entwicklung der Peers, die sich im Netzwerk mit Churn befinden. Dadurch, dass sich durch das Churn nicht mehr alle Peers gleichzeitig im Netzwerk befinden, sinkt ebenfalls die Last, die im Netzwerk entsteht. In den in dieser Arbeit betrachteten Versionen von Chord, Re-Chord und Adaptive-Chord kopieren Peers die ihnen zugeordneten Dateien nicht auf andere Peers, wenn diese das Netzwerk verlassen. In der Folge wird eine Datei unerreichbar, wenn der Peer, der für diese zuständig ist, das Netzwerk verlässt. Diese Tatsache muss daher in die Interpretation der durchschnittlichen Erfolgsrate eines Downloads mit einbezogen werden. Denn ein Downloadversuch gilt ebenfalls als gescheitert, wenn der Peer, der für die Datei verantwortlich ist, nicht kontaktiert werden kann. Dies wird besonders deutlich, werden die Erfolgsquoten der Downloads in Szenarien mit Churn betrachtet. Abbildung 5.20 zeigt die typische Entwicklung dieser Erfolgsquoten. Wird in diese Quote jedoch die Anzahl der Dateien mit einbezogen, die sich zu jedem Zeitpunkt im Netzwerk befinden, so wird erkennbar, dass die Erfolgsquote der Downloads von Dateien im Adaptive-Chord-Netzwerk, die sich zum Zeitpunkt der Dateianfrage auch im Netzwerk befinden, um einen Wert von im Durchschnitt 87% bewegen, wohingegen Chord nur eine Erfolgsquote von 70% im Durchschnitt erzielen kann. Siehe hierzu Abbildung 5.24.

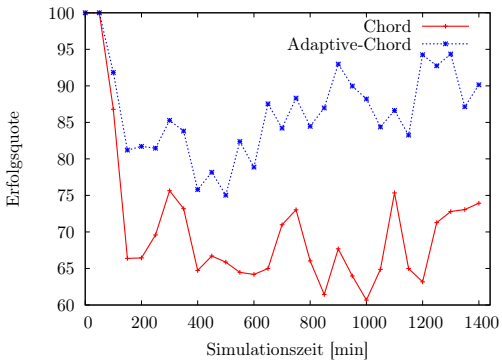


Abbildung 5.24: Typische Entwicklung der durchschnittlichen Erfolgsrate eines Downloads in einem Szenario mit Churn, bezogen auf die Anzahl der Dateien, die sich im Netzwerk befinden.

5.3.7 Szenario busy_unstable_churn & busy_stable_churn

Wie bereits erwähnt, beeinflusst das Churn die Entwicklung der Kenngrößen, weshalb es in Szenarien mit Churn nicht möglich ist, kleinste Veränderungen aus der

Entwicklung der Kenngrößen zu erkennen. Daher war es zu erwarten, dass sich der Unterschied zwischen einem kurzen und einem langen Intervall der Veränderung der Popularität einzelner Dateien in einem Szenario mit Churn nicht signifikant auf die Entwicklung der Kenngrößen niederschlägt. Der so entstehende Effekt wird vollständig durch die Effekte des Churn unkenntlich gemacht. Aus diesem Grund werden an dieser Stelle nur die Ergebnisse des Szenarios busy_unstable_churn präsentiert. Die Ergebnisse des Szenarios busy_stable_churn sind diesen sehr ähnlich.

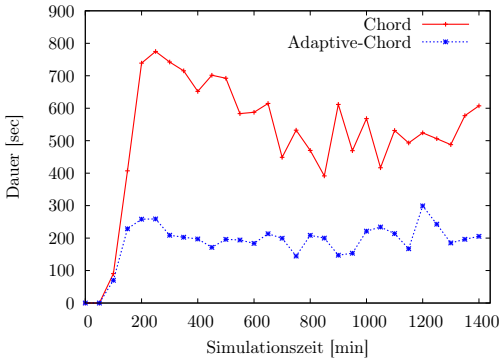


Abbildung 5.25: Durchschnittliche Dauer eines Downloads.

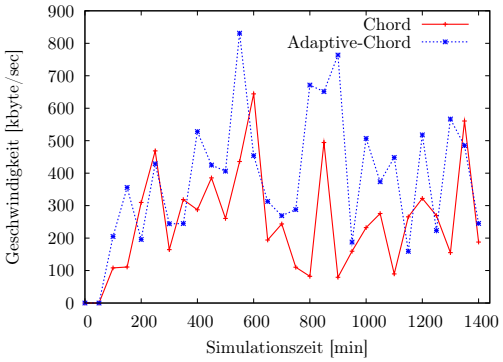


Abbildung 5.26: Durchschnittliche Geschwindigkeit eines Downloads.

Wird Abbildung 5.25 betrachtet, so fällt auf, dass sich die durchschnittliche Dauer eines Dateitransfers im Chord-Netzwerk im Vergleich zum selben Szenario ohne Churn deutlich verkürzt hat. Dies liegt maßgeblich an der Tatsache, dass sich, aufgrund des Churn, deutlich weniger Peers im Netzwerk befinden, vgl. Abbildung 5.21. In der Folge sinkt die gesamte Last im Netzwerk, was auch sehr gut in Abbildung 5.28 zu sehen ist. Diese sinkende Last bewirkt anschließend, dass jeder Peer im Netzwerk weniger Dateianfragen bedienen muss, was dazu führt, dass die Geschwindigkeit, mit der Dateien versendet werden können, steigt und somit die durchschnittliche Dauer eines Transfers sinkt.

In Abbildung 5.27 ist die Erfolgsrate der Downloads zu sehen. Diese beschreibt für das Chord- sowie für das Adaptive-Chord-Netzwerk eine $c \cdot \frac{1}{\log(x)}$ -Kurve, die gegen 30% konvergiert. Diese Erfolgsquote lässt sich durch das Churn erklären: Wie bereits erwähnt, werden Dateien im Netzwerk unerreichbar, wenn der Peer, der die Verantwortung für diese Dateien besitzt, nicht weiter am Netzwerk teilnimmt. Da die Anzahl der Teilnehmer im Netzwerk, wie in Abbildung 5.21 zu sehen, ebenfalls eine $c \cdot \frac{1}{\log(x)}$-Kurve beschreibt, die gegen 30% der einstigen Teilnehmerzahl konvergiert, beschreibt auch die Anzahl der sich im Netzwerk befindlichen Dateien eine solche Kurve. Um die Qualität, mit der Daten im Netzwerk erreichbar sind, zu vergleichen, ist dies jedoch nicht von Bedeutung. Aus Abbildung 5.27 geht klar hervor, dass

die durchschnittliche Erfolgsquote der Dateitransfers bei Adaptive-Chord zu jedem Zeitpunkt über der von Chord liegt. Adaptive-Chord besitzt somit einen klaren Vorteil in diesem Szenario im Vergleich zu Chord.

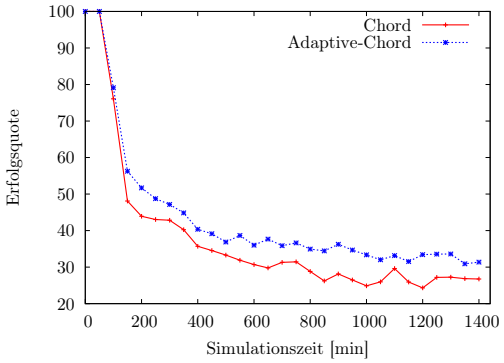


Abbildung 5.27: Durchschnittliche Erfolgsrate eines Downloads.

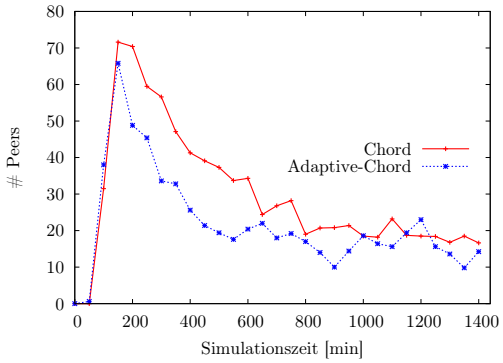


Abbildung 5.28: Anzahl der überlasteten Peers.

Trotzdem ist gut erkennbar, dass der Churn einen negativen Einfluss auf das Adaptive-Chord-Netzwerk besitzt. War in einem vergleichbaren Szenario ohne Churn die durchschnittliche Dauer eines Dateitransfers noch bis zu 9-mal kürzer als in dem betrachteten Chord-Netzwerk, so ist in Abbildung 5.25 zu sehen, dass die Dauer eines Dateitransfers im Adaptive-Chord-Netzwerk mit Churn nur noch in etwa halb so kurz ist wie im Chord-Netzwerk. Dies liegt insbesondere daran, dass die Anzahl der Peers, die im Netzwerk aktiv sind, stark unter Churn zurückgeht. Da somit ebenfalls weniger Dateianfragen im Netzwerk gestellt werden, existiert somit auch insgesamt weniger Last, weshalb die Anzahl der überlasteten Peers stark zurückgeht, wie in Abbildung 5.28 zu sehen. Zusätzlich werden durch das Churn bereits aufgestellte Spiegel aus dem Netzwerk entfernt, woraufhin diese auf anderen Peers erneut aufgestellt werden müssen. Unabhängig davon ist jedoch durch die dargelegten Zahlen ersichtlich, dass die Qualität der Dateibereitstellung auch in diesem Szenario im Adaptive-Chord-Netzwerk höher ist als in dem Chord-Netzwerk.

5.3.8 Szenario idle_unstable_churn & idle_stable_churn

Die Szenarien idle_unstable_churn und idle_stable_churn mit geringer Dateianfrage und mit Churn zeichnen sich dadurch aus, dass in diesen das Datenaufkommen sehr gering ist. Wird Abbildung 5.32 betrachtet, so fällt auf, dass kaum Peers im Chord-Netzwerk existieren, die überlastet sind. Entsprechend gering fällt in diesem Szenario der Vorteil für Adaptive-Chord aus.

Abbildung 5.29 zeigt die durchschnittliche Dauer eines Dateitransfers in diesem Szenario. Dadurch, dass durch das Churn mit der Zeit auch die Anzahl an Dateian-

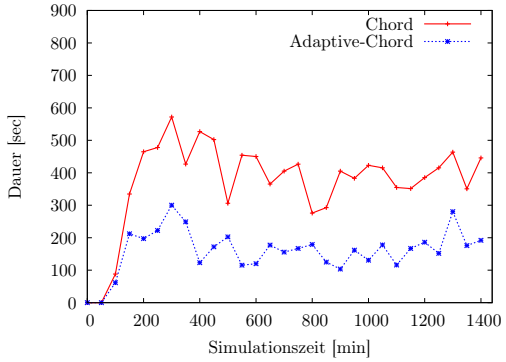


Abbildung 5.29: Durchschnittliche Dauer eines Downloads.

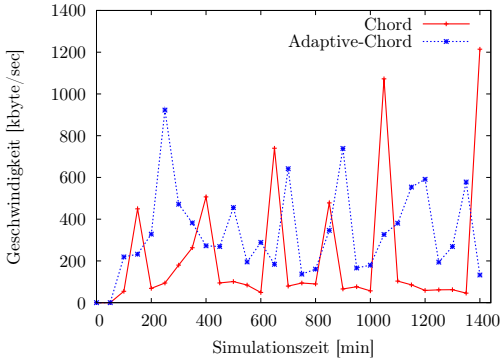


Abbildung 5.30: Durchschnittliche Geschwindigkeit eines Downloads.

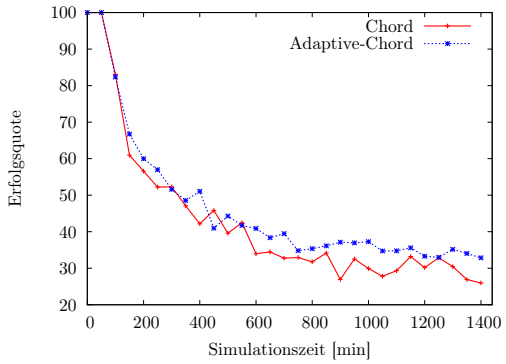


Abbildung 5.31: Durchschnittliche Erfolgsrate eines Downloads.

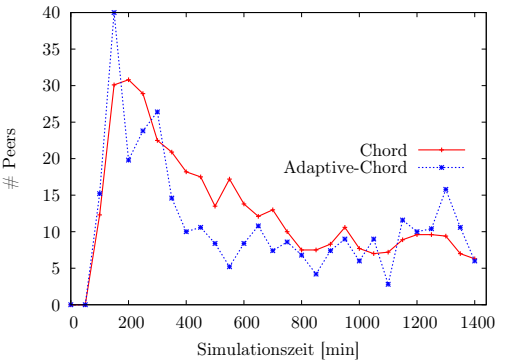


Abbildung 5.32: Anzahl der überlasteten Peers.

fragen abnimmt, sinkt die Last im Netzwerk mit der Zeit. Dies geschieht soweit, dass im Chord-Netzwerk nach Ablauf der betrachteten 24 Stunden nur 8 Peers existieren, die überlastet sind, vgl. Abbildung 5.32. Vor diesem Hintergrund ist es nicht verwunderlich, dass aus Abbildung 5.29 hervorgeht, dass die durchschnittliche Dauer eines Dateitransfers in Chord nur noch halb so lang ist, wie die in Adaptive-Chord. Dies lässt sich auch an der durchschnittlichen Geschwindigkeit erkennen, mit der Dateien übertragen werden: Immer, wenn eine der meist nachgefragten Dateien im Chord-Netzwerk von einem Peer mit hoher Uploadkapazität bereitgestellt wird, übersteigt die durchschnittliche Geschwindigkeit der Downloads die von Adaptive-Chord. Abbildung 5.31 zeigt die durchschnittliche Erfolgsquote der Dateiübertragungen: Wie zu erkennen ist, ist diese Quote in Adaptive-Chord ab $t = 600\ min$ zu jedem Zeitpunkt höher als in Chord.

Durch dieses Szenario lässt sich erkennen, dass Adaptive-Chord deutlich bessere Ergebnisse erzielen kann, wenn im Netzwerk genug Last entsteht, damit ein großer

Teil der Peers in dem Chord-Netzwerk überlastet sind. Dies ist auch nicht verwunderlich, balanciert Adaptive-Chord die Last im Netzwerk und sorgt somit dafür, dass weniger überlastete Peers im Netzwerk existieren.

5.4 Ergebnisse

Szenario	**v** [KByte/s]	**t** [s]	σ **t** [s]	# **O**	**Erfolg**
busy_unstable_nochurn					
Chord	118.6	806.7	1471.9	158.6	71.8 %
Re-Chord	125.3	791.9	1441.1	157.5	72.4 %
Adaptive-Chord	275.4	85.9	154	18.2	97.1 %
idle_unstable_nochurn					
Chord	126.5	619.9	1152.8	76.4	87.7 %
Re-Chord	111.9	576.4	1106.5	71.4	88.3 %
Adaptive-Chord	243.8	72.8	108.1	5.3	98.8 %
busy_stable_nochurn					
Chord	116.2	810.2	1467	160.4	72.2 %
Re-Chord	125.8	796.4	1437.7	158.8	72.3 %
Adaptive-Chord	258.2	87.5	156.3	18.9	97 %
idle_stable_nochurn					
Chord	133.2	611.8	1143.4	75.1	87.8 %
Re-Chord	132.7	599.9	1133.2	74.1	88.3 %
Adaptive-Chord	229.9	73.9	110.1	5.3	98.8 %
busy_unstable_churn					
Chord	249.2	514.4	958.4	29.4	38.1 %
Adaptive-Chord	379.8	186.1	493.5	21.8	43.6 %
idle_unstable_churn					
Chord	207.8	370.3	577.9	12.7	43.2 %
Adaptive-Chord	333.7	159.1	340.1	11	46.8 %
busy_stable_churn					
Chord	231.7	511.6	966.6	30.3	39.4 %
Adaptive-Chord	352.8	185.3	480.3	24.5	44.4 %
idle_stable_churn					
Chord	216.5	321.8	509.7	11.8	43.2 %
Adaptive-Chord	323.7	194.5	423.4	12.9	45.6 %

Tabelle 5.2: Durchschnittliche Werte der Kenngrößen aller Szenarien. **v** bezeichnet die durchschnittliche Downloadgeschwindigkeit, **t** die durchschnittliche Dauer eines Downloads und σ **t** die durchschnittliche Standardabweichung der Dauer eines Downloads. Die durchschnittliche Anzahl der überlasteten Peers ist mit # **O** gekennzeichnet.

Tabelle 5.2 fasst die Ergebnisse der betrachteten Szenarien zusammen. Wie bereits aus der Diskussion der einzelnen Szenarien hervorgeht, unterstreichen die hier präsentierten Durchschnittswerte noch einmal den qualitativen Unterschied zwischen Chord und Adaptive-Chord. So ist die von Adaptive-Chord erzielte durchschnittliche Geschwindigkeit einer Dateiübertragung in jedem Szenario höher als die von Chord erzielte Geschwindigkeit. Zusätzlich ist deutlich zu erkennen, dass die Zeit, die zwischen dem Absenden einer Dateianfrage und dem vollständigen Erhalt der Datei vergeht, in Adaptive-Chord deutlich niedriger ist als in Chord. So wird hierfür in jedem der im Rahmen dieser Arbeit betrachteten Szenarien in Chord mindestens doppelt so viel Zeit benötigt wie in Adaptive-Chord. Durch die Standardabweichung der Dauer eines Downloads wird zusätzlich deutlich, wie sehr die Qualität der Dateibereitstellung in Chord schwankt. Diese starke Schwankung wird durch die in Adaptive-Chord zum Einsatz kommenden Methoden deutlich verringert.

5.4.1 Bewertung von Re-Chord

Wie in allen Szenarien ohne Churn zu sehen, weichen die betrachteten Kenngrößen der Simulationen mit Re-Chord nicht signifikant von den Simulationen mit Chord ab. Somit ist die Qualität der Bereitstellung der Daten für den Benutzer eines Re-Chord-Netzwerkes ohne Churn nicht von der Benutzung eines Chord-Netzwerkes zu unterscheiden. Werden jedoch Szenarien mit Churn betrachtet, fällt auf, dass die von Re-Chord eingesetzten Stabilisationsmechanismen nicht in der Lage sind, das Netzwerk in einem Zustand zu halten, in dem dieses funktionsfähig ist. Nachdem das Netzwerk in den hier betrachteten Szenarien circa zwei Stunden lang dem Churn ausgesetzt ist, sind so viele Einträge in den Routingtabellen der Peers eines Re-Chord-Netzwerkes ungültig, dass dieses nicht mehr in der Lage ist, Anfragen im Netzwerk korrekt zu routen, vgl. hierzu Abbildung 5.20. Die Stabilisationsmethoden sind, während das Netzwerk weiterhin dem Churn ausgesetzt ist, nicht in der Lage, die durch den Churn zerstörte Topologie des Netzwerkes wieder in einen Zustand zu überführen, in dem das Netzwerk funktionstüchtig ist. Dies ist maßgeblich der Tatsache zuzuschreiben, dass das Re-Chord-Overlay bis zu $\mathcal{O}(n \log(n))$ Runden benötigt, um aus einem unzusammenhängenden Netzwerk den Chord-typischen Chord-Ring herzustellen. Da dieser Prozess jedoch immer wieder durch das Churn gestört wird, ist es Re-Chord nicht möglich, im Netzwerk einen Chord-Ring zu konstruieren.

Durch die in dieser Arbeit gesammelten Informationen über das Verhalten von Re-Chord in der Praxis unter Bedingungen, wie diese in einem P2P-Netzwerk herrschen, das maßgeblich von privaten Heimcomputern über das Internet aufgespannt wird, ist der Schluss zu ziehen, dass sich Re-Chord unter diesen Umständen nicht als praxistauglich erwiesen hat. Die Anzahl der zu versendenden Koordinationsnachrichten während der Stabilisationsmethoden ist zu hoch, um die Frequenz, mit der

diese Methoden ausgeführt werden, auf ein Niveau zu heben, auf dem das Netzwerk in der Lage ist, die durch das Churn ungültig gewordenen Einträge in den Routingtabellen der Teilnehmer zu korrigieren. In der Folge zerfällt das Netzwerk soweit, dass sich der Chord-Ring auflöst und das Netzwerk somit nicht mehr funktionstüchtig ist. Von einem Einsatz von Re-Chord in einem Szenario mit einer solch hohen Fluktuation unter den Teilnehmern ist somit abzuraten.

Eine Möglichkeit, den Re-Chord-Ring vor dem Zerfallen zu bewahren, ohne die Stabilisierungsmethoden zu oft ausführen zu müssen, sind Ersatzzeiger für die Predecessor- und Successor-Einträge in der Routingtabelle. Das normale Chord-Netzwerk schützt den Chord-Ring vor dem Zerfall, indem jeder Peer nicht nur einen, sondern mehrere dieser Zeiger besitzt, vgl. Kapitel 2.4.5. Dadurch wäre Re-Chord in der Lage, unabhängig von den Stabilisationsmethoden, den nächsten Peer auf dem Re-Chord-Ring zu benennen, auch wenn der Zeiger auf den direkten Nachfolger nicht mehr gültig ist.

5.4.2 Bewertung von Adaptive-Chord

Adaptive-Chord zeigt in allen Szenarien im Vergleich zu Chord eine klare Verbesserung der Qualität der Bereitstellung der Dateien im Netzwerk. In den Szenarien ohne Churn wurde besonders deutlich, dass die von Adaptive-Chord genutzten Mittel dazu in der Lage sind, die durchschnittlichen Datenraten in einem Chord-Netzwerk zu steigern. Dies zeigt sich besonders in den Szenarien *busy_ unstable_ nochurn* und *busy_ stable_ nochurn*, in denen eine hohe Dateinachfrage herrscht. In diesen Szenarien kann die durchschnittliche Dauer eines Dateitransfers um bis zu Faktor 9 verringert werden. Doch auch in den Szenarien mit weniger hoher Dateinachfrage kann die durchschnittliche Dauer um bis zu Faktor 5 verkürzt werden. Zusätzlich kann aufgrund der Tatsache, dass Adaptive-Chord die Anzahl der überlasteten Peers verringert, eine merklich höhere Erfolgsquote bei den Dateitransfers von Adaptive-Chord verzeichnet werden.

Doch nicht nur unter diesen idealisierten Bedingungen ist Adaptive-Chord in der Lage, bessere Ergebnisse zu liefern, als ein Chord-Netzwerk: So bescheinigen auch unter realistischen Bedingungen mit Churn die betrachteten Kenngrößen Adaptive-Chord durchgängig eine bessere Qualität in der Datenbereitstellung als Chord. Die durchschnittliche Dauer eines Dateitransfers konnte in jedem betrachteten Szenario durch den Einsatz des Lastbalancierungsverfahrens von Adaptive-Chord deutlich gesenkt werden, wobei die Erfolgsquote der Transfers bei Adaptive-Chord auf einem höheren Niveau liegt, als bei Chord. Durch die Evaluation wurde deutlich, dass der in dieser Arbeit vorgestellte Lastbalancierungs-Algorithmus funktionstüchtig ist und die Qualität der Datenbereitstellung in einem strukturierten P2P-Netzwerk signifikant verbessert.

6
Fazit & Ausblick

Strukturierte Peer-to-Peer-Netzwerke werden verwendet, um Daten dezentral ablegen und abrufen zu können. Dabei bestehen diese Netzwerke in der Regel aus einer heterogenen Menge von Teilnehmern, die sich in den zur Verfügung stehenden Ressourcen zum Teil stark unterscheiden. In den meisten strukturierten Peer-to-Peer-Netzwerken wird eine Datei immer nur von einem Peer im Netzwerk angeboten, wobei dieser Peer mit Hilfe einer deterministischen Funktion ausgewählt wird. In diese Funktion fließen jedoch keine Informationen über die von einer Datei erzeugte Last oder über die Leistungsfähigkeit eines Peers ein. In der Folge werden diese Informationen bei der Zuordnung von Dateien zu Peers auch nicht berücksichtigt. Dies hat zur Folge, dass in einem strukturierten Peer-to-Peer-Netzwerk Dateien existieren können, die von Peers zur Verfügung gestellt werden, die über zu geringe Ressourcen verfügen, um die Nachfrage nach der Datei bedienen zu können. In der Folge werden diese Dateien im Netzwerk nur unzureichend zur Verfügung gestellt.

In dieser Arbeit wurde ein Lastbalancierungsverfahren für strukturierte Peer-to-Peer-Netzwerke entwickelt, das die in einem strukturierten P2P-Netzwerk entstehende Last gut auf die Teilnehmer des Netzwerkes verteilen kann, sodass die erreichten durchschnittlichen Datenraten steigen und die durchschnittliche Dauer eines Dateitransfers sinkt. Dies wird erreicht, indem die Bereitstellung einer Datei entkoppelt wird von der Verantwortung für eine Datei. Ist ein Peer nicht in der Lage, die Last einer bestimmten Datei zu tragen, kann dieser Peer die Datei von einem anderen Peer bereitstellen lassen.

Dazu wurde in dieser Arbeit die Last, die ein Peer tragen muss, definiert als

die Menge der Daten, die dieser pro Zeiteinheit versenden muss. Zusätzlich wurden die Ressourcen eines Peers p in Abhängigkeit der von p zur Verfügung gestellten Upload-Kapazität definiert, sodass ein Peer überlastet ist, wenn die von dem Peer zu tragende Last die verfügbaren Ressourcen übersteigt, dieser Peer somit mehr Daten pro Zeiteinheit versenden soll, als es diesem möglich ist. Diese Definition stellt in der Literatur im Zusammenhang mit strukturierten P2P-Netzwerken eine Neuerung dar und macht es möglich, ein P2P-Netzwerk auf Grundlage der tatsächlich im Netzwerk auftretenden Last zu bewerten. Dies stellt gerade im Bezug auf Zipf-verteilte Dateizugriffsmuster und unterschiedliche Dateigrößen der im Netzwerk angebotenen Dateien einen signifikanten Unterschied zu den sonst in der Literatur benutzen Annahmen der gleichen Dateigrößen oder gleichverteilten Zugriffsmustern dar. Auf Grundlage dieser Definition wurde der Lastbalancierung-Algorithmus Adaptive-Chord vorgestellt, der im Rahmen dieser Arbeit entwickelt wurde. Adaptive-Chord verringert die Anzahl der überlasteten Peers in strukturierten P2P-Netzwerken deutlich. Dies hat gewisse Auswirkungen auf die Qualität, mit der Dateien in einem P2P-Netzwerk zur Verfügung gestellt werden: So zeigen Experimente, dass in einem idealisierten Szenario die durchschnittliche Dauer eines Dateitransfers mit einem Adaptive-Chord-Netzwerk um bis zu Faktor 9 im Vergleich zu einem Chord-Netzwerk gesenkt werden kann. Doch auch in realistischen Szenarien, in denen Churn herrscht, wurde gezeigt, dass Adaptive-Chord die durchschnittliche Zugriffszeit halbiert.

Um zu diesem Ergebnis zu gelangen, wurde zunächst in Kapitel 2 ein Überblick über die verschiedenen Techniken der P2P-Overlays gegeben. Anschließend wurden die strukturieren von den untstrukturierten P2P-Overlays abgegrenzt und das Chord-Overlay detailliert vorgestellt. In Kapitel 3 wurden zwei wesentliche Einschränkungen von Chord untersucht. Dazu wurde zunächst die Eigenschaft der Selbst-Stabilisation definiert und festgestellt, dass ein Chord-Netzwerk diese Eigenschaft nicht besitzt. Weiter wurde das Re-Chord-Overlay vorgestellt. Dieses ersetzt die von Chord verwendeten Stabilisationsmethoden durch Methoden, die dazu führen, dass ein Re-Chord-Netzwerk selbst-stabilisierend ist. In Kapitel 3.2 wurde überprüft, inwiefern Chord die Ressourcen eines heterogenen Teilnehmerfeldes ausnutzen kann, um mit diesen mögliche Unterschiede in der erzeugten Last der Dateien zu kompensieren, die daher auftreten, da in vielen Fällen die Dateianfragemuster keiner Gleichverteilung folgen. Dabei wurde festgestellt, dass bei der Zuteilung von Dateien zu Peers in Chord auf Grund der verwendeten Hashfunktion die freien Ressourcen der Peers, die Dateianfragemuster sowie die Dateigrößen keine Beachtung finden. In der Folge bleiben viele Ressourcen der Peers ungenutzt und es existieren Dateien, die von Peers bereitgestellt werden, die nicht über genügend Ressourcen verfügen, um die von den Dateien erzeugten Lasten zu tragen. Somit werden diese Dateien nur mit sehr schlechter Qualität im Netzwerk bereitgestellt, was sich durch eine erhöhte

durchschnittliche Dauer eines Dateitransfers in Chord bemerkbar macht. Kapitel 4 zeigt, wie das im Rahmen dieser Arbeit entwickelte Adaptive-Chord arbeitet. Dazu wurde zunächst die allgemeine Idee von Adaptive-Chord vorgestellt und ein weiteres Overlay-Netzwerk eingeführt, mit dem es möglich ist, effektiv Peers im Netzwerk zu finden, die eine gewisse Anzahl an freien Ressourcen besitzen. Mit Hilfe dieses zusätzlichen Overlay-Netzwerkes wurde anschließend der von Adaptive-Chord verwendete Lastbalancierungs-Algorithmus definiert.

Im Rahmen der in Kapitel 5 durchgeführten Evaluation wurden Chord, Adaptive-Chord und Re-Chord betrachtet. Dabei wurde festgestellt, dass Re-Chord ohne Churn durch die betrachteten Kenngrößen in der Qualität nicht von Chord zu unterscheiden war. Allerdings ist Re-Chord nicht in der Lage, diese Qualität in realistischen Szenarien mit Churn aufrechtzuerhalten: Die Stabilisationsmethoden von Re-Chord müssen zu häufig ausgeführt werden, um dafür zu sorgen, dass die Topologie des Netzwerkes nicht durch das Churn zerstört wird. In der Folge war das Re-Chord-Netzwerk nicht in der Lage, das Routing korrekt zu übernehmen, was dazu führte, dass so gut wie keine Datei im Netzwerk lokalisiert werden konnte. Diese Beobachtung lässt den Schluss zu, dass die in Kapitel 3.1.2 vorgestellte Re-Chord-Variante nicht praxistauglich ist. Weitere Forschung in diesem Bereich könnte sich daher auf die Verschmelzung der Stabilisationsmethoden von Re-Chord und Chord konzentrieren. So ist Re-Chord nachweislich in der Lage, aus jedem beliebigen Netzwerk, das zusammenhängend ist, einen Chord-Ring zu bilden, vgl. [KKS11]. Die in dieser Arbeit durchgeführten Experimente jedoch zeigen, dass die dafür eingesetzten Methoden von Re-Chord nicht in der Lage sind, diesen Chord-Ring auch bei Churn in einem Zustand zu halten, in dem das Routing funktionstüchtig ist. Dies jedoch ist durch die von Chord eingesetzten Stabilisationstechniken möglich, sodass es denkbar ist, dass diese Techniken gemeinsam eingesetzt dafür sorgen können, dass die Zeit, die benötigt wird, bis ein durch Churn beschädigter Re-Chord-Ring durch die Stabilisationsmethoden instandgesetzt wird, deutlich verkürzt werden kann. Insbesondere sollte in diesem Zusammenhang erforscht werden, wie mehrere Successor- und Predecessor-Zeiger pro Peer dabei helfen können, das Ausführungsintervall der Stabilisationsmethoden von Re-Chord zu verringern. Zusätzlich sollte erforscht werden, ob es möglich ist, das Nachrichtenaufkommen, das durch die von Re-Chord eingesetzten Stabilisationsmethoden erzeugt wird, zu reduzieren.

Die Evaluation von Adaptive-Chord zeigte, dass Adaptive-Chord die Anzahl der überlasteten Peers im Netzwerk merklich verringert. Dies führt dazu, dass – unter idealen Bedingungen – die durchschnittliche Zeit für einen Dateitransfer im Vergleich zu Chord um bis zu Faktor 9 verringert wird. Dadurch, dass in Adaptive-Chord deutlich weniger überlastete Peers existieren, steigt auch die Rate der erfolgreichen Transfers von im Durchschnitt 80% in Chord auf einen Durchschnittswert von über 97% in Adaptive-Chord. Doch auch in den realistischen Szenarien, in denen Churn

vorhanden war, konnte Adaptive-Chord die Qualität der Bereitstellung der Dateien im Gegensatz zu Chord verbessern: so war selbst in einem Szenario, in dem wenig Last im Netzwerk entstand, die durchschnittliche Dauer eines Downloads in Adaptive-Chord nur halb so lang wie in Chord.

Um die Datenraten im Adaptive-Chord-Netzwerk weiter zu erhöhen, kann der Adaptive-Chord-Ansatz durch eine Vielzahl von zusätzlichen Verfahren erweitert werden. So kann weiter erforscht werden, inwiefern Techniken, wie *Application-Layer Traffic Optimization* [APY11], dafür genutzt werden können, für stark nachgefragte Dateien mehrere Spiegel aufzustellen, und, je nach Qualität der Verbindung des nachfragenden Peers zu einem der Spiegel, die Anfrage an den Spiegel weiter zu leiten, der die bestmögliche Qualität für diesen Dateitransfer liefern kann.

Da das Aufstellen von mehreren Spiegeln für ein solches Szenario jedoch mehr Zeit benötigt als in dem Fall, in dem jede Datei nur maximal einen Spiegel besitzt, müssen Techniken bereitgestellt werden, die das gleichzeitige Aufstellen von mehreren Spiegeln beschleunigen. Denkbar wäre hier eine Technik, die auch bei dem Peer-to-Peer-Overlay *BitTorrent* zur Anwendung kommt: BitTorrent teilt eine Datei in verschiedene, sogenannte Chunks auf. Diese Chunks werden bei einem Dateitransfer kooperativ von einer Gruppe von Peers zueinander übertragen. Besitzt Peer p die vollständige Datei und existiert eine Reihe von Peers $q_1, ..., q_n$, die diese Datei anfragen, so kooperieren diese in BitTorrent. Dies geschieht, indem jeder Chunk, der von einem Peer q_i empfangen wird, von diesem Peer an die anderen Peers weitergegeben wird. So dient q_i, der bereits einen unvollständigen Teil der angefragten Datei besitzt, für diesen Teil als Spiegel und versendet diesen an die anderen nachfragenden Peers, während dieser bereits weitere Teile der Datei von p oder einem anderen Peer q_j empfängt. Auf diese Art muss p die Datei nicht zu jedem einzelnen der n Peers versenden, sondern versendet jeden Chunk der Datei nur wenige mal, und diese Chunks werden zwischen den Peers $q_1, ..., q_n$ verteilt. Wenn diese Technik in Adaptive-Chord beim Aufstellen von Spiegeln verwendet wird, dann könnte zu jeder Datei mehr als ein Spiegel im Netzwerk entstehen, ohne dass für diesen Prozess deutlich mehr Zeit und Uploadkapazitäten des Peers mit der Verantwortung für die Datei verwendet wird. Die so entstehende Mehrzahl an Spiegeln für eine Datei kann dazu beitragen, die im Netzwerk vorhandenen Kapazitäten der Peers besser auszunutzen, als dies noch bei der in dieser Arbeit vorgestellten Variante von Adaptive-Chord der Fall ist.

Das in dieser Arbeit entwickelte Lastbalancierungsverfahren Adaptive-Chord ist in der Lage, die in einem Chord-Netzwerk auftretenden Last so auf die Teilnehmer des Netzwerkes zu verteilen, dass die durchschnittliche Downloadgeschwindigkeit im Vergleich zu einem unmodifizierten Chord-Netzwerk signifikant steigt. In der Folge sinkt die Zeit, die ein Dateitransfer benötigt, um vollständig übertragen zu werden. Die Qualität der Datenbereitstellung wird daher durch das in Adaptive-Chord

eingesetzte Lastbalancierungsverfahren erhöht. Zusätzlich bietet dieses Verfahren durch die Trennung von Dateiverantwortung und Dateibereitstellung eine Vielzahl von Erweiterungsmöglichkeiten, die in der weiteren Forschung untersucht werden sollten.

Literaturverzeichnis

[AAG+05] K. Aberer, L. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth. The essence of P2P: A reference architecture for overlay networks. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*. IEEE, 2005.

[AB07] M. Abdallah and E. Buyukkaya. Fair Load Balancing under Skewed Popularity Patterns in Heterogeneous DHT-based P2P Systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*. ACTA Press, 2007.

[APY11] R. Alim, R. Penn, and Y. Yan. ALTO Protocol. Internet Draft <draft-ietf-alto-protocol-09.txt>, IETF, June 2011.

[BCM03] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the Annual International Workshop on Peer-To-Peer Systems*. Springer, 2003.

[BKM05] M. Bienkowski, M. Korzeniowski, and F. Meyer auf der Heide. Dynamic Load Balancing in Distributed Hash Tables. In *Proceedings of the Annual International Workshop on Peer-To-Peer Systems*. Springer, 2005.

[BPS06] A. Bharambe, J. Pang, and S. Seshan. Colyseus: a Distributed Architecture for Online Multiplayer Games. In *Proceedings of the Conference on Networked Systems Design & Implementation*. ACM, 2006.

[BSFK06] S. Bianchi, S. Serbu, P. Felber, and P. Kropf. Adaptive Load Balancing for DHT Lookups. In *Proceedings of the Annual IEEE International Conference on Computer Communications and Networks*. IEEE, 2006.

[BSW00] J. Brzezinski, M. Szychowiak, and D. Wawrzyniak. Self-Stabilization in Distributed Systems – A Short Survey, 2000.

[Cli01] Clip2 Distributed Search Service. The Gnutella Protocol Specification v0.4. 2001.

[DHMK10] Z. Despotovic, G. Hofstaetter, M. Michel, and W. Kellerer. An Operator Approach to Popularity-based Caching in DHTs. In *Proceedings of the IEEE International Conference on Communications*. IEEE, 2010.

[Dij74] E. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17, 1974.

[DKK+01] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the ACM symposium on Operating Systems Principles*. ACM, 2001.

[ES05] J. Eberspächter and R. Schollmeier. *P2P Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*, chapter Past and Future. Springer, 2005.

[FSY05] A. Fiat, J. Saia, and M. Young. *Making Chord Robust to Byzantine Attacks*, volume 3669 of *Lecture Notes in Computer Science*. Springer, 2005.

[GDS+03] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-To-Peer File-Sharing Workload. In *Proceedings of the ACM symposium on Operating Systems Principles*. ACM, 2003.

[GGS+11] K. Graffi, C. Groß, D. Stingl, D. Hartung, A. Kovacevic, and R. Steinmetz. LifeSocial.KOM: A Secure and P2P-based Solution for Online Social Networks . In *Proceedings of the IEEE Consumer Communications and Networking Conference*. IEEE, 2011.

[Gra11] K. Graffi. PeerfactSim.KOM: A P2P System Simulator – Experiences and Lessons Learned. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*. IEEE, 2011.

[GTC+08] L. Guo, E. Tan, S. Chen, Z. Xiao, and X. Zhang. The Stretched Exponential Distribution of Internet Media Access Patterns. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 2008.

[HLL+04] F. Hong, M. Li, X. Lu, Y. Wang, J. Yu, and Y. Li. VChord: Constructing Peer-to-Peer Overlay Network by Exploiting Heterogeneity. In *Proceedings of the Erlang User Conference*. EUC, 2004.

[HWW02] S. Hazel, B. Wiley, and O. Wiley. Achord: A Variant of the Chord Lookup Service for Use in Censorship Resistant Peer-to-Peer Publishing

Systems. In *Proceedings of the International Workshop on Peer-to-Peer Systems*. USENIX, 2002.

[HY10] E. Harjula and M. Ylianttila. Load balancing models for DHT-based Peer-to-Peer Networks. Internet Draft <draft-harjula-p2psip-loadbalancing-survey-01.txt>, IETF, March 2010.

[KK03] M. Kaashoek and D. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In *Proceedings of the International Workshop on Peer-to-Peer Systems*. USENIX, 2003.

[KKL+07] A. Kovacevic, S. Kaune, N. Liebau, R. Steinmetz, and P. Mukherjee. Benchmarking Platform for Peer-to-Peer Systems (Benchmarking Plattform für Peer-to-Peer Systeme). *it - Information Technology*, 49, 2007.

[KKS11] S. Kniesburges, A. Koutsopoulos, and C. Scheideler. Re-Chord: A self–stabilizing Chord overlay network. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2011.

[KL07] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 1st edition, 2007.

[KLL+97] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceeding of the ACM Symposium on Theory of Computing*. ACM, 1997.

[KLXH04] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-To-Peer Support for Massively Multiplayer Games. In *Proceedings of the Annual IEEE International Conference on Computer Communications and Networks*. IEEE, 2004.

[KM02] T. Klingenberg and R. Manfredi. Gnutella 0.6 RFC. draft; http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html, June 2002.

[KN07] G. Kunzmann and R. Nagel. Efficient Simulation of Large-Scale P2P Networks: Modeling Network Transmission Times. In *Proceedings of the Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2007.

[KPL+09] S. Kaune, K. Pussep, C. Leng, A. Kovacevic, G. Tyson, and R. Steinmetz. Modelling the Internet Delay Space Based on Geographical Locations. In *Proceedings of the Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2009.

[KR04] D. Karger and M. Ruhl. Diminished Chord: A Protocol for Heterogeneous Subgroup Formation in Peer-to-Peer Networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems*. USENIX, 2004.

[Lew98] D. Lewin. Consistent Hashing and Random Trees : Algorithms for Caching in Distributed Networks. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1998.

[LLD04] B. Leong, B. Liskov, and E. Demain. EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management. In *Proceedings of the International Conference on Networks*. IEEE, 2004.

[LS05] J. Ledlie and M. I. Seltzer. Distributed, Secure Load Balancing with Skew, Heterogeneity and Churn. In *Proceedings of the International Conference on Computer Communications*. IEEE, 2005.

[Man11] S. Manuel. Classification and Generation of Disturbance Vectors for Collision Attacks against SHA-1. *Designs, Codes and Cryptography*, 59, 2011.

[MCvR03] V. Mesaros, B. Carton, and P. van Roy. S-Chord: Using Symmetry to Improve Lookup Efficiency in Chord. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 2003.

[MS07] P. Mahlmann and C. Schindelhauer. *Peer-to-Peer-Netzwerke: Algorithmen und Methoden*. Springer, 1 edition, 2007.

[PaP] The Peers at Play Project. `http://www.peers-at-play.org/`.

[PSK] Documentation for PeerfactSim.KOM. `http://peerfact.kom.e-technik.tu-darmstadt.de/fileadmin/data/Manuals/PeerfactSimKOM_Documentation.pdf`.

[RCFB07] W. Rao, L. Chen, A. Fu, and Y. Bu. Optimal Proactive Caching in Peer-To-Peer Network: Analysis and Application. In *Proceedings of the ACM Conference on Information and Knowledge Management*. ACM, 2007.

[RLS+03] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Structured P2P Systems. In *Proceedings of the International Workshop on Peer-To-Peer Systems*. Springer, 2003.

[RM06] J. Risson and T. Moors. Survey of Research Towards Robust Peer-To-Peer Networks: Search Methods. *The International Journal of Computer and Telecommunications Networking*, 50, 2006.

[RS04] C. Riley and C. Scheideler. Local Load Balancing in Distributed Hash Tables. unpublished manuscript; `http://www.cs.jhu.edu/~scheideler/papers/balancing.ps.gz`, 2004.

[SENB09] M. Steiner, T. En-Najjary, and E. Biersack. Long Term Study of Peer Behavior in the KAD DHT. *IEEE/ACM Transactions on Networking*, 17, 2009.

[SGG02] S. Saroiu, K. Gummadi, and S. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of the Multimedia Computing and Networking*. SPIE, 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM Special Interest Group on Data Communication Conference*. ACM, 2001.

[SW05] R. Steinmetz and K. Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, 2005.

[WY06] J. Wang and Z. Yu. A New Variation of Chord with Novel Improvement on Lookup Locality. In *Proceedings of the International Conference on Grid Computing and Applications*. Springer, 2006.

[Zhu04] Y. Zhu. Towards Efficient Load Balancing in Structured P2P Systems. In *Proceedings of International Parallel and Distributed Processing Symposium*. IEEE, 2004.