Untersuchung von Anonymisierungsverfahren in DHT basierten Overlays

Bachelorarbeit

von

Erol Yildirim

geboren in
Leverkusen

vorgelegt am

Lehrstuhl für Technik sozialer Netzwerke
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

Juli 2015

Betreuer:
Tobias Amft

Abstract

In dieser Arbeit geht es um die Untersuchung von Anonymisierungsverfahren in DHT basierten P2P-Overlays. Unter allen vorgestellten Anonymisierungsverfahren in dieser Arbeit hebt sich das Prinzip der Clouds, welche in dem DHT basierten P2P-Overlay Agyaat benutzt werden, von den anderen Anonymisierungsverfahren hervor, da dieses unterschiedliche Anonymisierungsverfahren vereint, um die Anonymität von Peers zu schützen. Basierend auf Agyaat, wird ein Overlay namens ACE (Anonymous Cloud Environments , zu deutsch: Anonyme Cloud Umgebungen) in den PeerfactSim.KOM Simulator eingebunden und ausgewertet. ACE ist skalierbar und ermöglicht einem Peer Dateien zu publizieren, zu suchen und zu beziehen, ohne dabei seine Identität nach außen zu tragen.

Danksagung

Ich möchte mich bei meinem Betreuer, Tobias Amft, für seine Ratschläge und Anregungen, für die Zeit, die er sich für das wöchentliche Treffen genommen hat, für das Beantworten sämtlicher Emails und für das Korrekturlesen meiner Bachelorarbeit bedanken. Zudem möchte ich mich bei meinem Kommilitonen Mustafa Yilmaz bedanken, der bei Fragen mit hilfsreichen Antworten immer an meiner Seite stand und mich während der Bearbeitungsphase motiviert hat. Des Weiteren möchte ich mich auch bei meinem Kommilitonen Yasin Yazgan für seine Hilfestellungen bedanken.

Inhaltsverzeichnis

1 Motivation **1**

1.1 Was versteht man unter Anonymität ? . . . 2
1.2 Strukturierte und Unstrukturiere P2P-Netzwerke . . . 3
1.3 DHT-basierte P2P-Overlays . . . 4
1.4 Chord . . . 5
1.5 Ausblick . . . 6

2 Verwandte Arbeiten **7**

2.1 Angreifbarkeit der Anonymität in DHT basierten P2P-Overlays . . . 7
2.2 Metriken zur Anonymität . . . 9
2.2.1 Anonymity Set Metrik . . . 9
2.2.2 Entropie Metrik . . . 9
2.3 Anonyme Systeme . . . 10
2.3.1 Crowds . . . 11
2.3.2 Freenet . . . 11
2.4 DHT basierte P2P-Overlays mit Anonymisierungsverfahren . . . 12
2.4.1 Achord . . . 12
2.4.2 Agyaat . . . 14
2.4.3 ACE . . . 16
2.5 Techniken zur Anonymisierung . . . 16
2.5.1 Anonyme DHT . . . 16
2.5.2 Rekursives Routing . . . 17
2.5.3 Random Walks . . . 17
2.5.4 Clouds . . . 18

3 Design der Anonymous Cloud Environments **21**

3.1 Main-Ring . . . 21
3.2 R-Ring . . . 24
3.2.1 Erstellen und Beitreten von Clouds . . . 25
3.2.2 Funkionalitäten des Rendezvous-Knotens und des Pingers . . . 26

3.2.3 Datei Publizierung . . . 30
3.2.4 Dateien suchen und beziehen . . . 34
3.3 Modifikationen zur Erweiterung der Anonymität . . . 38
3.3.1 Freundschaftsbeziehungen in der Cloud . . . 38
3.3.2 Dynamische Dateilagerung . . . 39

4 Implementierung **41**
4.1 Darstellbarkeit von Knoten . . . 41
4.1.1 AbstractAceNode . . . 41
4.1.2 AceNode . . . 42
4.1.3 RringAceNode . . . 44
4.1.4 PingerAceNode . . . 46
4.2 Die globale Cloud Instanz . . . 47
4.3 Nachrichten . . . 49
4.4 Hilfsklassen . . . 51
4.4.1 Verteilung von Knoten auf Clouds . . . 52
4.4.2 Zuweisung von zufälligen Datei Id's auf Knoten . . . 52
4.4.3 Austausch der Kontaktinformationen bei Freundschaftsbeziehungen . . . 52

5 Simulation **53**
5.1 Metriken . . . 53
5.2 Setup . . . 55
5.2.1 Freundschaftsbeziehungen . . . 55
5.2.2 Churn . . . 56
5.3 Ergebnisse . . . 56
5.4 Auswertungen . . . 63
5.4.1 Durchschnittliche Hop Anzahl . . . 63
5.4.2 Random Walk . . . 64
5.4.3 Dateierreichbarkeit . . . 65
5.4.4 Netwerktraffic . . . 65
5.4.5 Nachrichten . . . 66
5.4.6 Freundschaftsbeziehungen . . . 66

6 Ausblick **67**
6.1 Zukünftige Arbeiten . . . 68

Literaturverzeichnis **69**

Kapitel 1

Motivation

In der heutigen Zeit gewinnen P2P Protokolle immer mehr an Bedeutung. Sei es der NSA Skandal oder der Kauf von Whatsapp seitens Facebook, welche die Orthonormalverbraucher dazu anregt, ihre Daten und ihre Privatsphäre in Frage zu stellen. Einen Server in der Mitte während einer Kommunikation über das Internet zu haben, verunsichert viele Orthonormalverbraucher und macht solche Anwendungen unattraktiv und rückt sie in ein schlechtes Licht, welche den Schlagzeilen, die sich in der heutigen Zeit breit machen, zu verbuchen sind. An dieser Stelle werden P2P-Protokolle oder P2P-Overlays interessant, denn diese arbeiten dezentralisiert und haben keinen Server in der Mitte, durch den eine Kommunikation zustande kommt und welcher in der Lage ist die komplette Kommunikation zu verfolgen und festzuhalten. Benutzer können direkt miteinander kommunizieren ohne auf einen Server angewiesen zu sein und brauchen sich zunächst keine Gedanken darüber machen, ob ihre Daten oder Kommunikationsinhalte an Dritte weitergegeben werden können. In Tauschbörsen wie z.B. BitTorrent [Bit], die auf P2P-Protokollen beruhen, ist es möglich Daten mit anderen Benutzern auszutauschen ohne einen zentralen Server für die Abwicklung des Datentransfers haben zu müssen.

Der Verzicht auf einen zentralen Server und stattdessen eine End- zu Endverbindung zu anderen Benutzern führen zu können, mag viele Benutzer auf den ersten Blick erfreuen. Doch reicht es aus nur auf einen zentralen Server zu verzichten, der in der Lage wäre Daten und Inhalte an Dritte weiterzugeben? So wie das Internet sind P2P-Protokolle nicht unter dem Gesichtspunkt Sicherheit und Privatsphäre entwickelt worden. Das Internet wurde bekanntlich für einen kleinen Kreis von Wissenschaftlern entwickelt, die sich gegenseitig vertrauten und Daten austauschen wollten und erforderte somit keine strengen Sicherheitsmaßnahmen, was heutzutage wiederum nicht mehr wegzudenken ist. Ähnlich ist es bei P2P-Overlays. Es wäre also naiv, davon auszugehen, dass alle Benutzer in einem P2P-Netzwerk gute Absichten besitzen und nur zum Austausch von Daten einem P2P-Netzwerk beitreten. Böswillige Benutzer eines P2P-Netzwerkes könnten auch Spionage betreiben und Informationen über bestimme Benutzer an Dritte weitergeben. Benutzer, die nur zum Austausch von Daten einem P2P-Netzwerk beitreten und anonym bleiben möchten, können durch böswillige Benutzer ihre Anonymität und ihre Privatsphäre verlieren. Abgesehen von den oben genannten Problemen sind Anwendungen, die zen-

tralisiert arbeiten bzw. einen Server als zentrale Einheit benutzen, nutzlos, wenn der Server z.B. von Angeifern lahm gelegt wird. Man könnte die Anwendung nicht mehr benutzen, da der Server, der für die Abwicklung jeglicher Funkitonalitäten zuständig ist, nicht mehr erreichbar ist. Konzipiert man jedoch ein P2P-System, wo es keine zentrale Einheit gibt und alle Aufgaben auf die verschiedenen Peers verteilt werden, so ist dieses wesentlich robuster, wenn ein Peer ausfallen sollte und kann in diesem Fall die Aufgaben des ausgefallenen Peers anderen Peers übergeben, wodurch die Anwendung trotz Ausfall einiger Peers sich selbst stabilisieren und weiterhin ihre Dienste anbieten kann.

In dieser Arbeit möchte ich Anonymisierungsverfahren vorstellen und mein eigenes DHT-basiertes P2P Overlay basierend auf einem speziellen Anonymisierungsverfahren implementieren und mithilfe des PeerfactSim.KOM Simulators [FG13] auswerten . Dieses Anonymisierungsverfahren möchte ich an Chord austesten, da Chord das Paradebeispiel für DHT basierte P2P-Overlays ist, wegen seiner einfachen und plausiblen Arbeitsweise, aber dennoch ein effektives Overlay darstellt, welches skalierbar ist und Daten in $O(\log(n))$ beziehen lässt. Dieses Anonymisierungsverfahren wird in dieser Arbeit zwar an Chord ausgetestet, ist aber auf DHT-basierte P2P-Overlays wie z.B Pastry [RD01] und Kademlia [MM02] übertragbar.

1.1 Was versteht man unter Anonymität ?

Zunächst einmal ist es wichtig, darauf einzugehen, was Anonymität ist und wie Anonymität sich in P2P-Netzwerken ausprägt. An dieser Stelle möchte ich ein gängiges Zitat zur Anonymität von P2P-Netzwerken anführen, welches auch in anderen Papern bzgl. der Anonymisierung von P2P-Netzwerken gerne zitiert wird [PK01] **„ Anonymity is the state of being not identifiable within a set of subjects, the Anonymity Set“**. In P2P-Netzwerken ist es üblich, dass Peers Nachrichten an andere Peers senden, um z.B an bestimme Informationen ranzukommen oder auch um Informationen im Netzwerk zu verbreiten. Ein Sender einer Nachricht gilt dann als anonym, wenn er in der Menge der möglichen Sender (*Anonymity Set*) einer Nachricht nicht eindeutig als Sender von einem böswilligen Knoten oder einem Angreifer identifiziert werden kann. In der Menge der möglichen Sender bzw. der *Anonymity Set* befinden sich nur ehrliche Knoten, d.h Knoten die nicht von böswilligen Knoten oder einem Angreifer gesteuert werden und somit trivialerweise nicht in Betracht als mögliche Sender gezogen werden. Es ist offentsichtlich, dass ein Sender als anonymer gilt, wenn die Menge aus möglichen Sendern bzw. die *Anonymity Set* eine größe Anzahl an Peers enthält, denn dadurch ist ein Angreifer nicht in der Lage den Sender eindeutig zu identifizieren, wohingegen ein Sender als weniger anonym gilt, wenn die *Anonymity Set* aus wenigen Knoten besteht. Besteht die *Anonymity Set* nur aus einem möglichen Sender, so kann ein Angreifen diesen eindeutig identifizieren, wodurch der Sender der Nachricht seine Anonymität nicht bewahren kann. Hierbei spricht man von der *Senderanonymität*, die einen Sender einer Nachricht nach außen hin anonym und für andere Peers nicht sichtbar machen

soll. Auf der anderen Seite kann ein Empfänger einer Nachricht nach außen hin auch nicht erkennbar sein wollen, weil er z.B. nicht möchte, dass andere Peers ihn mit den Nachrichteninhalten in Verbindung bringen sollen. Eine Nachricht könnte z.B. eine Dateianfrage enthalten und der Empfänger dieser Anfrage möchte als Beherberger dieser Datei nicht in Verbindung gebracht werden und möchte nicht, dass andere Peers wissen, dass er die Datei im Netzwerk zur Verfügung stellt. Hier spricht man dann von der *Empfängeranonymität*, die den Empfänger einer Nachricht bzw. einer Anfrage nach außen hin nicht sichtbar machen soll. Ausgehend von Chord, könnte ein Benutzer, der Dateien von anderen Benutzern beziehen möchte, nicht wollen, das andere Benutzer wissen oder herausbekommen können, dass er der Initiator einer bestimmen Anfrage einer Datei ist. Im gleichen Atemzug möchte ein Benutzer, der bestimme Dateien anbietet, auch nicht wollen, dass andere Benutzer wissen, dass er bestimme Dateien lagert und anderen Benutzern zur Verfügung stellt.

Allgemein ist es in P2P-Netzwerken nicht üblich, dass diese nur aus Peers bestehen, die sich gegenseitig vertrauen und miteinander befreundet sind. So könnte ein bestimmter Peer, der mit fremden Peers kommunizieren muss, sich in seiner Privatsphäre eingeschränkt fühlen, da dieser Peer z.B. nur mit Peers kommunizieren möchte, mit denen er befreundet ist, da er anderen Peers nicht vertraut und Angst um seine Anonymität hat. Zudem könnte ein Peer wünschen von nur so wenig wie möglich anderen Peers gesehen zu werden und vorzugsweise nur von befreundeten Peers. Es gibt jedoch P2P-Netzwerke wie z.B. „Friend-to-Friend Networks“ [Bri00], welche so konzipiert sind, dass nur bekannte bzw. befreundete Peers miteinander kommunizieren.

Neben der *Empfänger*- und *Senderanonymität* gibts es also die *Anonymität durch Vertrauen*, bei der Peers nur mit befreundeten Peers kommunizieren möchten und sogar möglichst nur von diesen in dem P2P-Netzwerk gesehen werden möchten. Die *Anonymität durch Vertrauen* steht in dieser Arbeit im Gegensatz zur *Empfänger*- und *Senderanonymität* nicht im Fokus und wird erst wieder in den Kapiteln 3 und 4, in welchen mein Overlay ACE vorgestellt wird, wieder aufgegriffen.

1.2 Strukturierte und Unstrukturiere P2P-Netzwerke

Eines der größten Ziele von P2P-Netzwerken ist es, einem Benutzer zu ermöglichen nach Dateien suchen zu können und diese auffindig zu machen und anderen Benutzern seine eigenen Dateien anbieten zu können. Um dies möglich zu machen, werden P2P-Netzwerke unter einem bestimmten Design konzipiert. Ein Id-Space, der in der Regel eine sehr große Anzahl an zu vergebenen Id's erfasst, ist unverzichtbar. Dateien und auch Peers werden auf diesen abgebildet und erhalten ihre Id, wodurch sie eindeutig identifiziert werden können. Der Id-Space muss von den Peers, die dem P2P-Netzwerk beigetreten sind, verwaltet werden, da Dateien, die auch auf den Id-Space abgebildet werden, von diesen zur Verfügung gestellt werden müssen. Um Dateien auffindig zu machen und um sie beziehen

zu können, sind Routing-Strategien erforderlich, anhand welcher Anfragen zu den entsprechenden Beherbergern dieser Dateien geroutet werden können. Routing-Strategien müssen immer auf dem neusten Stand gehalten werden, damit Suchanfragen oder andere Operationen auch beim Verlassen des Netztwerks von anderen Peers funktionieren können.

Es gibt zwei Arten von P2P-Netzwerken, unzwar strukturierte und unstrukturierte P2P-Netzwerke. Diese sind zwar nach den oben aufgeführten Design konzipiert, unterscheiden sich jedoch in ihren einzelnen Funktionen und Arbeitsweisen.

Unstrukturierte P2P-Netzwerke benutzen Id's von Dateien um sie eindeutig zu identifizieren und benutzen sie nicht für Lookups oder für das Routen, da unstrukturierte P2P-Netzwerke nach Dateien oder Objekten suchen, die zu den von einem Peer ausgewählten Kriterien passen. Dateien werden von demjenigen Peer beherbergt und verwaltet, der diese dem Netzwerk zur Verfügung stellen möchte und werden nicht einem anderen Peer zugewiesen. Bei unstrukturierten P2P-Netzwerken ist es also nicht möglich gezielt eine Datei anzufragen, sondern Suchanfragen zu starten, die Dateien und Objekte liefern, die zu einem bestimmten Suchkriterium passen.

Strukturierte P2P-Netzwerke hingegen benutzen Id's von Dateien zwar auch um sie zu identifizieren, benutzen sie aber auch bei einem Lookup oder beim Routen und arbeiten nicht mit ausgewählten Suchkriterien oder Ähnlichem. Peers werden genau wie Dateien auf den selben Id-Space abgebildet. Jede Id des Id-Spaces wird einem Peer zugewiesen und kann beim Verlassen des Netzwerkes eines Peers an einen anderen Peer weitergegeben werden. Jeder Peer ist somit für einen bestimmten Bereich im Id-Space verantwortlich. Im Gegensatz zu unstrukturierten P2P-Netzwerken ist es möglich nach einer gezielten Datei in Form eines Lookups zu suchen.

1.3 DHT-basierte P2P-Overlays

Unter DHT-basierten Overlays (DHT = Distributed Hash Table, zu deutsch „Verteilte Hash Tabelle“) versteht man ein sich selbst organisierendes System, welches Benutzer und Dateien mittels einer konsistenten Hashfunktion auf einen Id-Space abbildet und die Informationen darüber verwaltet, welcher Benutzer welche Dateien lagert. Die Speicherung dieser Informationen sind über alle Knoten des Overlays verteilt und komplett dezentralisiert. Ein Eintrag einer DHT besteht aus einem Paar aus Schlüssel und einem zugehörigen Wert. Hierbei versteht man unter einem Schlüssel die Id einer bestimmten Datei und den zugehörigen Wert als den Beherbeger dieser Datei. DHT-basierte P2P-Overlays bieten zwei Möglichkeiten Dateien zu verwalten, unzwar das Direct- und Indirect Storaging. Beim Direct Storaging ist derjenige Knoten, auf den ein Dateischlüssel abgebildet wird, der Beherbeger dieser Datei. Im Gegensatz dazu wird beim Indirect Storaging ein Benutzer, auf den ein

Dateischlüssel abgebildet wird, die Datei nicht beherbergen, sondern nur die Kontaktinformationen des Beherbergers in seiner DHT verwalten.

DHT-basierte Overlays werden unter den Gesichtspunkten Flexibilität, Skalierbarkeit und Zuverlässigkeit konzipiert. Sie sollten flexibel sein und beim Verlassen von Knoten des Netzwerkes das Netzwerk selbständig stabilisieren können und Verantwortungsgebiete dieses Knoten auf andere anwesenden Knoten übertragen können, damit das Netzwerk weiterhin fehlerlos funktionieren kann. Ein Knoten sollte ohne große Probleme in der Lage sein Dateien bzw. den Beherbeger von Dateien in Form von Lookups auffindig zu machen, weshalb Zuverlässigkeit und Robustheit in DHT-basierten Overlays unabdingbar sind. Zudem sollten Dateien gleichmäßig über alle Knoten des Overlays verteilt werden.

Offensichtlich wird die Bewahrung der Anoynmität von Benutzern von DHT-basierten P2P-Overlays bei der Konzipierung im Allgemeinen nicht berücksichtigt. Deshalb ist es mein Ziel in dieser Arbeit mein eigenes DHT-basiertes Overlay ACE vorzustellen und zu erörtern, welches unter Rücksicht des Gesichtspunktes Anonymität bzw. *Sender*- und *Empängeranonymität* konzipiert worden ist. Zudem bietet ACE die Möglichkeit die *Anonymität duch Vertauen* so weitesgehend wie möglich zu bewahren.

1.4 Chord

Chord [SMK$^+$01] ist ein DHT-basiertes P2P-Overlay welches Peers und Dateien auf den selben Id-Space abbildet und alle Id's in einem Kreis anordnet. Der Id-Space besteht aus allen Id's in dem Intervall $[0, 2^m - 1]$, wobei meistens $m = 160$ verwendet wird, wodurch eine riesige Anzahl an zu vergebenen Id's vorhanden ist. Jede Datei und auch jeder Peer wird durch die konsistente Hashfunktion *SHA-1* auf den Id-Space abgebildet, wobei die eindeutige IP-Adresse von einem Peer und der Dateiinhalt einer Datei gehasht wird. Jeder Peer hat einen Predecessor (zu deutsch: Vorgänger) und einen Successor (zu deutsch: Nachfolger). Der Zuständigkeitsbereich im Id-Space, den ein Peer zugewiesen bekommt, sind alle Id's, die größer als die Id seines Predecessors sind bis einschließlich seiner eigenen Id. Ein Peer ist somit für alle Id's in dem Intervall $(predecessor.id, peer.id]$ verantwortlich.

Dank seiner Fingertabellen, die aus $m := \log(N)$ (N = Anzahl aller Id's im Id-Space) Einträgen besteht, die jeder Peer besitzt und periodisch aktualisiert, ist ein Lookup von Dateien in $O(\log(N))$ möglich. Unter dem Successor einer Datei versteht man denjenigen Peer, in dessen Zuständigkeitbereich die Id dieser Datei fällt. Möchte man also einen Lookup nach einer bestimmten Datei starten, so sucht man den Successor dieser Datei Id. Ein Fingertabelleneintrag besteht aus einer Id, welche die Id des Successor der Id $((n + 2^{i-1}) \bmod 2^m)$ beinhaltet, wobei $i \in \{1, \cdots, m\}$ und i den i-ten Eintrag in

der Fingertabelle bezeichnet und die Kontaktinformationen dieses Successor (kurz: $successor((n + 2^{i-1}) \bmod 2^m)$) in Form einer IP-Adresse und einer Portnummer.

Das Chord-Protokoll ist sehr robust und flexibel und kann auf das Wegfallen von Peers bzw. Knoten leicht reagieren. So müssen z.B. nur $O(\log(1/N))$ (N = Anzahl der Knoten auf dem Chord-Ring) Datei Id's einem anderen Knoten zugewiesen werden, wenn ein Peer das Netzwerk verlässt, damit diese Dateien erreichbar bleiben. Dies ist auf die konsistente Hashfunktion *SHA-1* zurückzuführen, die dafür sorgt, dass alle Keys gleichmäßig auf alle Knoten des Netzwerkes verteilt werden. Zudem verwaltet jeder Knoten eine Successor Liste, damit Knoten beim Wegfallen ihres Successors ihren neuen Successor kontaktieren können, wodurch die Kreisform eines Chord Ringes, die durch das Verketten von allen Knoten mit ihren Predecessoren und Successoren entsteht, erhalten bleibt.

Der Unterschied zwischen Chord und anderen DHT basierten P2P-Overlays liegt in der plausiblen Arbeitsweise, in der beweisbaren Korrekheit und in der beweisbaren Performance [SMK+01], weshalb Chord einen guten Repräsentanten von DHT basierten P2P-Overlays darstellt und als Paradebeispiel für diese gilt, weshalb ich mich dazu entschlossen habe Chord als Repräsentanten fur DHT basierte P2P-Overlays auf Anonymität zu untersuchen und einen Lösungsansatz für diesen mithilfe des Simulators PeerfactSim.KOM [FG13] zu implementieren und auszuwerten.

1.5 Ausblick

In Kapitel 2 werde ich auf bestehende Arbeiten bzgl. der Anonymität eingehen und mein eigenes Overlay vorstellen und darauf eingehen, wieso ich es für notwendig befunden habe, dieses zu entwickeln und zu implementieren. Kapitel 3 befasst sich mit meinem Overlay namens ACE und thematisiert wichtige Funktionen und Arbeitsweisen in Form von Pseudocodes, wonach ich in Kapitel 4 etwas detaillierter auf die Implementierung und einige Erklärungen eingehen werde. Weiterhin werde ich in Kapitel 5 auf die Simulation eingehen und meine Auswertungskriterien darlegen und diese erörten. Schließlich gibt es in Kapitel 6 eine Zusammenfassung meiner Arbeit, Ideen und Anregungen fur zukünftige Ausarbeitungen.

Kapitel 2

Verwandte Arbeiten

Nachdem wir in dem vorherigen Kapitel den Begriff Anonymität und einige Details zu DHT basierten P2P-Overlays behandelt haben, werde ich in diesem Kapitel bekannte Arbeiten zur Anonymisierung in P2P-Netzwerken erörten und deren Vor- und Nachteile aufweisen und darauf eingehen, wieso ich es für nötig befunden habe mein eigenes Overlay ACE zu erstellen. Außerdem werde ich darauf eingehen, welche bestehende Ideen ich zur Anonymisierung von P2P-Netzwerken in meinem Overlay übernommen habe und was diese zur Anonymität beitragen.

2.1 Angreifbarkeit der Anonymität in DHT basierten P2P-Overlays

Allgemein lässt sich sagen, dass DHT-basierte P2P-Overlays in der Regel keinerlei Verschleierung vornehmen um den Beherberger einer Datei anonym zu halten, denn ein Eintrag in einer DHT einer Datei Id oder eines Dateischlüssels enthält immer den Beherbeger einer Datei in Form von IP-Adresse und Portnummer und ist dadurch für jeden Peer im Netzwerk durch ein Lookup aufspürbar. Die *Empfängeranonymität* ist hiermit also im Allgemeinen verletzt. Eine Aussage über die *Senderanonymität* in DHT basierten P2P-Overlays zu treffen ist hingegen wesentlich schwieriger. Das Problem, welches jedoch DHT basierte P2P-Overlays in Punkto *Senderanonymität* besitzen, sind die Routingsstrategien. Diese sind meistens so konzipiert, dass sie eine gute Laufzeit versprechen und effektiv sind, weshalb sie im Allgemeinen keine Rücksicht auf die Bewahrung der *Senderanonymität* nehmen. Zudem sind Routingsstrategien deterministisch, d.h der Pfad, den eine Anfrage bis zu ihrem Ziel durchquert, ist eindeutig bestimmt und somit z.B. von anderen Peers nachvollziehbar.

Ausgehend von Chord ist ein Beherberger einer Datei durch eine *FindSuccessor* Operation, in der man den Successor einer Datei Id bzw. den Beherberger einer Datei sucht, ohne weitere Probleme auffindbar. Wird Direct Storaging verwendet, so ist dieser Successor der Beherberger der angefragten Datei und beim Indirect Storaging liefert dieser Successor den Beherberger der angefragen Datei,

weshalb Chord die *Empfängeranonymität* nicht gewährleisten kann. Die iterative Version von Chord, in welcher der Initiator einer Anfrage jeden Peer auf dem Weg zum Beherberger einer Datei selbst kontaktiert, verletzt offensichtlich die *Senderanonymität*, da Anfragen ohne jeglichen Verschleierung selbst an andere Peers geschickt werden. Im Gegensatz dazu wird bei der rekursiven Version von Chord die Anfrage zwar immer von demjenigen Peer weitergeleitet, der eine Anfrage erhält und die Antwort wird entsprechend von jedem Peer, der ein Teil des Anfrage Pfades gewesen ist, zurückgeleitet bis schließlich der Initiator der Anfrage die Antwort erhält, weshalb der Initiator zunächst anonym geblieben zu sein scheint. Da die Peers in Chord ihre Fingertabellen als Routingwerkzeuge benutzen und diese bei jedem Peer gleich bestimmt werden, ist es möglich Rückschlüsse darüber zu führen, von welchem Peer oder aus welchem Bereich des Chord-Rings die Anfrage stammen könnte auch bei der rekursiven Version von Chord, sofern ein Knoten eine Anfrage mitbekommt. Die Routingsstrategien in Chord basieren darauf, dass man bei einer Anfrage zunächst einen riesen Sprung auf dem Chord-Ring macht, um einen Peer nach einer Datei anzufragen. Besitzt der Peer die angefragte Datei nicht oder ist er nicht für die Datei zuständig, sowird er das Paket in der rekursiven Version von Chord an einen anderen Peer aus seiner Fingertabelle weiterschicken. Die Distanz wird in der Regel jedesmal von Peer zu Peer halbiert, bis sie schließlich den Verantwortlichen Peer für diese Id oder den Beherberger der angefragten Datei erreichen. Ein böswilliger Knoten, der sich in der Nähe eines Beherbergers einer bestimmten Datei im Chord Ring befindet, wird öfter Anfragen zu dieser bestimmten Datei mitbekommen, als wie wenn er weiter entfernt von diesem Beherberger positioniert wäre und ist somit in der Lage Rückrechnungen zu führen, um den Initiator der Anfrage nach dieser bestimmten Datei zu identifizieren [DV04]. Es ist jedoch wichtig zu erwähnen, dass ein böswilliger Knoten nicht immer in der Lage sein wird, den richtigen Initiator zu identifizieren, aber es ist oft möglich einen Initiator auf einen bestimmten Id-Bereich im Chord Ring zu beschränken. Auch wenn die rekursive Version von Chord nicht unter dem Aspekt *Senderanonymität* konzipiert worden ist und die oben erwähnten Lücken in den Routingsstrategien vorhand sind, erschwert es böswilligen Knoten dennoch den Initiator zu identifizieren und besitzt somit einen hohen Grad an *Senderanonymität* [DV04]. Zur *Anonymität duch Vertrauen* in Chord lässt sich sagen, dass ein Peer in Chord seine $O(\log(N))$ Finger, die Peers in seiner Successor Liste, die in der Regel aus $m := \log(N)$ Knoten bestehen, und seinen Predecessor und Successor kennen muss, damit das System von Chord funktionieren kann. Darüber hinaus ist jeder Knoten in der Lage andere Peers aufzusuchen, indem er die *FindSuccessor* Operation nach zufälligen Id's ausführt um ggf. neue Peers kennenzulernen. Ein Knoten bzw. Peer ist somit nicht darauf beschränkt, nur die nötigsten Peers zu kennen. Somit ist die *Anonymität duch Vertrauen* in Chord nicht gewährleistet, da Knoten die z.B. nur von ihren Freunden gesehen werden möchten auch von fremden Peers gesehen werden können. Außerdem können Peers, die nur mit befreundeten Peers kommunizieren wollen, in der Fingertabelle eines fremden Peers sein oder selbst Peers in der Fingertabelle haben, denen sie nicht vertrauen.

Im Allgemeinen lässt sich also festhalten, dass Anfragen in DHT basierten P2P-Overlays nicht iterativ abgehandelt werden dürfen, da man offensichtlich den Initiator nach außen hin sichtbar machen

würde. Rekursives Routen, um Anfragen abzuhandeln, scheint einen höheren Grad an *Senderanonymität* zu gewährleisten und wird in dieser Arbeit noch an vielen Stellen auftauchen und zeigen, dass dieses eine Grundbasis dafür bildet, um die *Senderanonymität* zu gewährleisten.

2.2 Metriken zur Anonymität

Um die in Abschnitt 2.1 aufgeführten Schwachstellen bzgl. der *Senderanonymität* in DHT basierten P2P-Overlays zu messen bzw. zu bewerten, werden im Folgenden zwei Ansätze vorgestellt. Dabei sind die vorgestellten Metriken von den Beobachtungen und dem Wissen eines böswilligen Knotens bzw. Angreifers über das Netzwerk abhängig.

2.2.1 Anonymity Set Metrik

In Abschnitt 1.1 wurde die *Anonymity Set*, welche eine Auskunft über den Grad der *Senderanonymität* gibt, vorgestellt. Je größer die *Anonymity Set*, desto weniger Aussagekraft besitzt ein Angreifer darüber den richtigen Sender zu identifizieren. Im Umkehrschluss wird ein Angreifer einen Sender eher identifizieren können, je kleiner die *Anonymity Set* ist. Betrachten wir Ω als die Menge, die alle potenziellen Sender einer Nachricht enthält und A die Menge aller Peers, die ein Angreifer als potenzielle Angreifer sieht. Dann kann durch:

$$\frac{|A|}{|\Omega|}$$

die Anonymity Set Metrik angegeben werden, die als Minimum $\frac{1}{|\Omega|}$ und als Maximum $\frac{|\Omega|}{|\Omega|} = 1$ animmt [BW05]. Erhält man das Minimum, so kann man einen Sender erfolgreich identifizieren, wohingegen man beim Maximum eine vollkommene Anonymität des Senders erhält.

2.2.2 Entropie Metrik

Die Anonymity Set Metrik beurteilt die *Senderanonymität* eines Senders nach der Anzahl der Peers in der *Anonymity Set*, weshalb sie sich als Metrik nicht gut anbietet, da ein Angreifer durch bestimme Angriffsmechnismen gewisse Informationen erhalten kann, und dadurch den einzelnen möglichen Sendern eine Wahrscheinlichkeit zuordnen kann, inwiefern diese der Sender seien könnten. Folgender Ansatz bedient sich der Entropie der Informationstheorie [Cov91], um die *Senderanonymität* unter den genannten Bedingungen zu erfassen [DSCP03]:

Sei X eine diskrete Zufallsvariable, die nach einer Attacke eines Angreifers den einzelnen Peers in der *Anonymity Set* Wahrscheinlichkeiten zuweist, mit welcher der Angreifer sie für den möglichen Sender hält. Sei $p_i = Pr(X = i)$ die Wahrscheinlichkeitsfunktion (*probability mass function*), die mit jedem i einen Peer aus der *Anonymity Set* darstellt und wobei jedes p_i die die Einschätzung der Wahrscheinlichkeit, dass Peer i der mögliche Sender ist, bestimmt. $H(X)$ bezeichne die Entropie des Systems, nach dem ein Angreifer einen Angriffsmechanismus ausgeführt hat, für die gilt:

$$H(X) = -\sum_{i=1}^{N} p_i \, log_2(p_i)$$

wobei $H_M = log_2(N)$ (N ist die Größe der *Anonymity Set*) die maximale Entropie eines Systems darstellt. Die Information, die ein Angreifer nach einem Angriffsmechanismus erfahren hat, kann durch $H_M - H(X)$ dargestellt werden. Um die *Senderanonymität* angeben zu können wird dieser Wert durch die maximale Entropie H_M geteilt und dadurch normiert. Somit kann die *Senderanonymität* bzw. der Grad der *Senderanonymität* wie folgt angegeben werden:

$$d = 1 - \frac{H_M - H(X)}{H_M} = 1 - \left(1 - \frac{H(X)}{H_M}\right) = \frac{H(X)}{H_M}$$

Für den Fall, dass es nur einen Peer in der *Anonymity Set* gibt, gilt $d = 0$. Für d bzw. für die *Senderanonymität* gilt im Allgemeinen also:

$$0 \leq d \leq 1$$

wobei $d = 0$, wenn ein Angreifer einen Peer mit Wahrscheinlichkeit 1 identifizieren kann und $d = 1$, wenn der Angreifer allen möglichen Sendern die selbe Wahrscheinlichkeit zuordnet ($p_i = \frac{1}{N}$). Die Entropie Metrik definiert den Grad der Senderanoymität also als die Summer aller Informationen, die ein System aufgrund ihrer Systemlücken preisgibt [DSCP03].

2.3 Anonyme Systeme

Die im Folgenden vorgestellten Systeme, werden häufig in Papern bzgl. der Anonymisierung von P2P-Netzwerken erwähnt und gehören zu den Paradebeispielen von anonymen Systemen, an deren Konzept man sich beim Designen eines anonymen P2P-Netzwerkes orientieren kann. Es gibt dennoch weitere anonyme Systeme wie z.B. Tor [DMS04], welches ähnliche Strategien wie die unten

genannten Systeme benutzt, um Anonymität zu gewährleisten. Mixes [Cha81] ist eines der ersten Ansätze zur Anonymisierung von Anwendungen (anonymes Email System) und somit ein Vorreiter eines der unten aufgeführten Systeme.

2.3.1 Crowds

Crowds [RR98] ist anonymes System, welches die Anonymität beim Web Browsing gewährleistet. Crowds besteht aus einer Menge von Peers bzw. Knoten, die sich gegenseitig ihre Webseiten Anfragen untereinander rumschicken, bis schließlich ein Knoten diese Anfrage an den Webserver einer Webseite versendet. Dem Webserver bleibt der Initiator der Anfrage offensichtlich verborgen, da der Webserver lediglich den Peer sieht, der ihn auch direkt kontaktiert. Den Peers selbst, welche die Anfragen weiterleiten, bleibt der Initiator der Anfrage auch verborgen, da ein Peer nicht wissen kann, ob die Anfrage vom demjenigen Knoten stammt, welcher ihm die Anfrage geschickt hat oder ob dieser die Anfrage ebenfalls von einem anderen Peer zugesandt bekommen hat. Jeder Peer merkt sich hierbei von welchem anderen Peer er die Anfrage erhalten hat, damit die Antwort der Anfrage an den Initiator geleitet werden kann. Das Weiterleiten einer Anfrage nach einer Webseite ist einer bestimmten Vorschrift unterworfen. Ein Peer wird eine Anfrage, die er erhält, mit der Wahrscheinlichkeit p_f an einen zufällig ausgewählten Peer und mit Wahrscheinlichkeit $1 - p_f$ an den Webserver schicken, weshalb die Peers auch nicht wissen können vom wem die Anfrage letztlich stammt. Der dadruch entstehende *Random Walk* Pfad hat etwa die Länge $\frac{1}{1-p_f}$ [Bor05]. Die *Senderanonymität* wird in Crowds somit gewährleistet.

2.3.2 Freenet

Freenet [CSWH01] ist ein DHT basiertes P2P-Overlay, welches anonyme Lagerung und anonymen Austausch von Dateien anbietet. Dateien werden in Freenet durch ihren Key, der durch das Hashen des Dateiinhaltes entsteht, angesprochen und gesucht. Wie es bei DHT basierten Overlays üblich ist, werden die Dateien über alle Knoten des Netzwerkes in einer DHT verwaltet. Jeder Knoten führt eine Routingtabelle, in der es andere Knoten und eine Listen mit deren möglichen Keys, für die diese Knoten zuständig sind, abspeichert. Sucht ein Knoten nach einer Datei, so wählt er als nächsten den Knoten in seiner Routingtabelle aus, dessen Key's zu seinem angefragten Key am ähnlichsten sind. Hat dieser Knoten wiederum die Datei auch nicht, so wird die Anfrage entsprechend weitergeleitet, bis letztlich der Beherberger der Datei erreicht wird. Die Datei kann dann durch den rekursiv aufgebauten Anfrage Pfad an den Initiator der Anfrage ausgehändigt werden. Alle Knoten auf dem Anfrage Pfad können die angefragte Datei in ihrem Cache speichern, und bei zukünftigen Anfragen anderen Knoten zur Verfügung stellen. Durch das Caching von Dateien erhöht sich der Grad der *Empfängeranonymi-*

tät, da ein Angreifer z.B. neben dem Beherberger einer Datei noch weitere Knoten als Beherberger der Datei wahrnehmen könnte und somit nicht entscheiden kann, welche der Knoten nun der ursprüngliche Beherberger der Datei ist. Man muss dazu erwähnen dass der Cache sowie die Routingtabellen einen begrenzenden Speicher besitzen und das diese nach bestimmten Vorschriften überschrieben und ersetzt werden. Für den Fall, dass eine Anfrage nicht beantwortet werden kann, weil der Beherberger einer Datei offline ist und diese Anfrage nicht unnötig im Netztwerk herumschwirren soll, haben alle Anfragen eine TTL (time-to-live), durch welche eine Anfrage verworfen wird, wenn ihre TTL auf 0 ist. Routing Schleifen, bei der eine Anfrage im Kreis umher geschickt wird, werden erkannt und entsprechend behandelt.

2.4 DHT basierte P2P-Overlays mit Anonymisierungsverfahren

Neben den in Abschnitt 2.3 vorgestellten Systemen, gibt es bestehende Arbeiten, die bestehende DHT's, welche Lücken in Punkto Anonymität bzw. *Sender*- und *Empfängeranonymität* aufweisen, behandeln und unter dem Gesichtspunkt Anonymität modifizieren. Eines der beiden im Folgenden vorgestellten Ansätze bildet eine Grundbasis meines Overlays ACE.

2.4.1 Achord

Achord [HW] ist ein DHT basiertes P2P-Overlay, welches so konzipiert wurde, um die oben benannten Lücken in Chord auszumerzen. Es ist ein modifiziertes Chord, welches im Grunde alle Funktionalitäten von Chord übernimmt, sie aber an gewissen Stellen einschränkt um die Anonymität bzw. *Sender*- und *Empfängeranonymität* und sogar die *Anonymität durch Vertrauen* weitesgehend zu bewahren. Wie bereits erwähnt, ist es möglich, dass ein Peer in Chord neben den Peers, die er aufgrund des Chord Protokolls kennen muss, auch in der Lage ist andere Peers kennenzulernen. Dies wird in Achord jedoch unterbunden, indem ein Peer nur diejenigen Knoten sieht, die er kennen muss, damit das Chord Prinzip funktioniert und alle Funktionalitäten ausgeübt werden können und ein böswilliger Peer z.B. nicht die Möglichkeit bekommt an Informationen über das Netzwerk zu gelangen, weshalb Achord in Punkto *Anonymität durch Vertrauen* einen hohen Grad besitzt. Die *FindSuccessor* Operation, die es in Chord ermöglichte beliebig viele andere Knoten kennenzulernen, wird in Achord eingeschränkt. In Chord lieferte die *FindSuccessor* Operation den Successor einer bestimmten Id, wodurch ein Peer neue Peers kennenlernen konnte und gleichzeitig den Beherberger einer Datei rausfinden konnte. Dies wird in Achord verhindert, indem Keys bzw. Id's auf Werte (engl. values) abgebildet werden. Gleichzeitig wird dadurch verhindert, dass man den Beherberger einer Datei herausfindet, denn man erhält lediglich einen Wert und nicht die Id oder die Kontaktinformationen des Beherbergers der angefragten Datei. Jeder Peer darf die *FindSuccessor* Operation nur benutzen

um den Successor seiner eigenen Id zu suchen, um beim Beitreten des Netzwerkes seine richtige Position im Chord Ring einnehmen zu können. Darüber hinaus darf man keinen Successor einer anderen Id mittels der *FindSuccessor* Operation suchen. Peers können leicht herausfinden, ob andere Peers eine zulässige *FindSuccessor* Operation ausführen, indem sie die IP-Adresse des Anfragers mittels der Hashfunktion *SHA-1* hashen und die somit erhaltene Id mit der Id des Anfragenden vergleichen. Außerdem darf die *FindSuccessor* Operation nur iterativ ausgeführt werden, da nur ein Peer selbst den Successor seiner Id suchen kann bzw. der *FindSuccessor* Operation seine eigene Id übergeben kann. Möchte man eine *FindSuccessor* Operation nach einer anderen Id durchzuführen, weil man nach bestimmten Dateien sucht und sie beziehen möchte, so führt man die *connect_to_successor* Methode durch, wodurch rekursiv ein Anfrage-Pfad zu dem Beherberger der Datei aufgebaut wird. Bei diesem angekommen, sendet dieser einen bestimmten Wert entlang des Anfrage-Pfades, der rekursiv aufgebaut wurde, zurück, sodass jeder Peer diesen Wert abspeichert und die Anfrage zurück leitet bis der Initiator der Anfrage diesen Wert erhält. Daraufhin ruft der Initiator die *connect_to_successor* Methode auf und übergibt ihr den erhaltenen Wert, wodurch alle Peers, die an der vorherige Anfrage beteiligt waren, wissen, an wen die Anfrage geleitet werden muss bis schließlich der Beherberger dieser Datei die Anfrage erhält und die angefragte Datei entlang des rekursives aufgebauten Pfad an den Initiator senden kann. Möchte ein Peer eine Datei publizieren bzw. dem Overlay zugänglich machen, so ruft dieser die *connect_to_successor* Methode auf, um den Verantwortlich für diese Datei herauszufinden und schickt seinen Wert entlang dieses dieses rekursiv aufgebauten Anfrage Pfades, wobei jeder Knoten auf dem Pfad diesen Wert abspeichert und bei zukünftigen Anfragen weiß an welchen anderen Peer die Anfrage weitergeleitet werden muss, damit sie zum Beherberger der Datei geleitet werden kann. Einem Peer auf einem rekursiv aufgebauten Anfragepfad ist es nicht möglich herauszubekommen, ob der Peer, von dem er unmittelbar eine Anfrage erhält, der Initiator ist, denn dieser könnte die Anfrage auch selbst von einem anderen Peer erhalten haben und einfach weitergeleitet haben.

Um die Sicht auf das Netzwerkes jedes Peers einzuschränken, sieht Achord vor, dass der Predecessor, Successor und die Fingertabellen eines Peers anderen Peers weitestgehend nicht sichtbar sind. Deshalb liegen bei der *Stabilize* Operation, bei der man den Predecessor seines Successor erfahren möchte, und bei der Bestimmung von Fingertabelleneinträgen mittels der *find_best_match* Funktion gewissen Einschränkungen vor, auf die ich hier nicht eingehen möchte, die man aber bei Intresse nachschlagen kann [HW].

Achord benutzt also „Tunneling“ als Verschleierungsstrategie, um Dateien zu beziehen und zu publizieren, womit es versucht die *Sender-*, *Empfängeranonymität* und zusätzlich durch Einschränkungen der Sicht auf das Netzwerk eines Peers einen hohen Grad an *Anonymität durch Vertrauen* zu gewährleisten.

2.4.2 Agyaat

Agyaat [SGL06] ist ein DHT basiertes P2P-Overlay, welches konzipiert wurde, um beidseitige Anonymität bzw. *Sender*- und *Empfängeranoymität* in strukturierten P2P-Netzwerken zu gewährleisten. Die Idee des Overlays beruht darauf, dass unstrukturierte P2P-Netzwerke wie z.B. Gnutella einen hohen Grand an *Sender*- und *Empfängeranonymität* bieten, da Knoten in unstrukturierte P2P-Netzwerken im Allgemeinen nur ihre unmittelbaren Nachbar kennen und mit diesen kommunizieren. Sie haben also eine beschränkte lokale Sicht auf das Netzwerk. Wenn ein Knoten eine Anfrage startet, so wird diese an seinen Nachbarn weitergeleitet, welcher wiederum, wenn er die Datei nicht besitzt, die Anfrage an seinen eigenen Nachbarn weiterleitet bis schließlich der Beherberger der Datei die Anfrage erhält und die Anfrage beantworten kann. Die Antwort der Anfrage kann dann auf dem rekursiv aufgebauten Anfrage Pfad dem Initiator zugesandt werden, ohne das weder die *Sender*- noch die *Empfängeranonymität* verletzt werden. Diese Eigenschaft versucht Agyaat in das Overlay einzubringen, indem es unstrukturierte Netzwerke an strukturierte Netzwerke anhängt. Diese unstrukturierten Netzwerke werden in Agyaat als *Clouds* bezeichnet. Zu dem sieht Agyaat vor, dass der Initiator einer Anfrage nur dem Initiator selbst bekannt ist, und dass eine Anfrage immer bei dem Beherberger einer Datei terminiert. Ein Gegenbeispiel hierzu wäre Chord, denn eine *FindSuccessor* Operation endet nicht bei dem Peer, der für eine bestimmte Datei zuständig ist, sondern bei seinem Predecessor. Solche Topologien, in welcher nur der Initiator weiß wer der Initiator einer Anfrage ist und wo Anfragen bei dem Beherberger einer Datei und nicht bei seinem Predecessor oder bei einem anderen Knoten terminieren, heißen LTI Topologien (LTI = Local Termination and Initiation preserving topology), welche beidseitig Anonymität bzw. *Sender*- und *Empfängeranonymität* gewährleisten sollen [SGL06]. Gnutella stellt wie andere unstrukturierte Netztwerke ebenso eine LTI Topologie dar. LTI Topologien werden in Agyaat als Clouds bezeichnet.

Unter einer Cloud versteht man ein unstrukturiertes Netzwerk von Benutzern, die jeder Cloud ihrer Wahl beitreten können. Clouds werden in Agyaat auf die DHT aufgesetzt und ermöglichen die Anoynmität des Senders und des Empfängers zu gewährleisten, indem Anfragen in einer Cloud gestartet werden und wieder in der Cloud des Beherbergers beim dem Beherberger einer Datei selbst terminieren. Im Gegensatz zu Chord und allgemein DHT basierten P2P-Overlays werden Keys nicht auf Peers, sondern auf Clouds abgebildet. In Chord ist der Beherberger einer Datei nach einem Lookup leicht enttarnt. In Agyaat wiederum wird der Beherberger, der sich in der Cloud aufhält, durch die Cloud nach außen hin verschleiert. Hierdurch wird die *Empfängeranonymität* bewahrt. Die *Senderanonymität* wird ebenfalls durch die Cloud bewahrt, da Anfragen per RandomWalk durch die Cloud wandern, bis schließlich ein Peer diese Anfrage abschickt.

Das Agyaat Overlay besteht aus 2 Ringen, einem üblichen Ring analog zu Chord mit den selben Funktionalitäten, außer dass Lookups oder Dateianfragen nicht über diesen Chord Ring getätigt werden und einmal dem R-Ring (Rendezvous-Node Ring). Der R-Ring besteht lediglich aus Rendezvous-Knoten,

welche einen Eingangspunkt für ihre eigenen Clouds darstellen und selbst Mitglied dieser Clouds sind. Im Gegensatz zu Chord werden Anfragen nicht über den üblichen Chord Ring sondern über den R-Ring geroutet. Clouds werden auf den R-Ring abgebildet und müssen durch ihre Rendezvous-Knoten ansprechbar sein, in dem der Name der Cloud gehasht wird und die daraus resultierende Cloud Id seinem Rendezvous-Knoten zugewiesen wird. Einen Rendezvous-Knoten einer Cloud wiederum bestimmt man, indem man den Successor der Cloud Id auf dem normalen Chord Ring (nicht R-Ring) sucht. Jede Cloud besitzt einen Rendezvous-Knoten und ist durch diesen erreichbar bzw. ansprechbar. Um die Id eines Rendezvous-Knoten auf dem R-Ring zu erhalten, wird der Cloud-Name der zugehörigen Cloud gehasht. Sucht man z.B. eine bestimmte Cloud mit einem bestimmten Namen, so sucht man auf dem R-Ring den Successor der Id, die man durch das hashen des Namens einer Cloud erhält. Möchte man also einer bestimmten Cloud beitreten, so sucht man den Rendezvous-Knoten dieser Cloud auf dem R-Ring. Der R-Ring, bestehend nur aus Rendezvous-Knoten, stellt wieder einen Chord Ring dar und besitzt auch dieselben Funktionalitäten wie ein üblicher Chord-Ring, besteht aber lediglich aus den Rendezvous-Knoten, welche ihre Cloud repräsentieren und zugänglich machen. Die Anzahl der Knoten auf dem R-Ring gleicht somit der Anzahl der vorhandenen Clouds. Auf dem R-Ring werden also alle Anfragen bearbeitet, da die Zugangspunkte bzw. Repräsentanten aller Clouds sich auf diesem verbinden und somit können Anfragen eines Initiators von seiner eigenen Cloud in die Cloud des Beherbergers der angefragten Datei geroutet werden.

Anfragen von Dateien werden zunächst per RandomWalk von dem ursprünglichen Initiator an seine Cloud-Nachbarn weitergereicht. Hierbei bestimmt der Initiator einer Anfrage eine zufällige TTL (time-to-live) und reicht das Paket an einen zufälligen Nachbarn in der Cloud weiter. Erhält ein Nachbar eine Anfrage, so überprüft er die TTL. Wenn diese ungleich 0 ist, verringert er sie um 1 und schickt die Anfrage mit der verringerten TTL an einen zufälligen Nachbarn in der Cloud. Derjenige Nachbar, der ein Paket erhält, wo das TTL 0 ist, ist letztlich für das Absenden der Anfragen außerhalb der Cloud verantwortlich und wird als *Crossover-Peer* bezeichnet. Der Initiator der Anfrage selbst darf aber als *Crossover-Peer* die Anfrage nicht verschicken, da er sich sonst nach außen hin zeigen würde. Agyaat sieht vor, dass nur Rendezvous-Knoten Einsicht über den R-Ring und dessen Routinginformationen besitzen. Möchte ein Crossover-Peer eine Anfrage auf den R-Ring übergeben, so muss dieser zunächst seinen Rendezvous-Knoten nach dem Einstiegspunkt auf dem R-Ring fragen. Somit wird gewährleistet, dass Anfragen auf den R-Ring von nicht Rendezvous-Knoten weitergegeben werden können. Kommt nun die Anfrage bei dem Rendezvous-Knoten an, welcher einen Eingangspunkt für die Cloud darstellt, wo sich der Beherberger der angefragten Datei befindet, so wird die Anfrage vom Rendezvous-Knoten an alle anderen Cloud-Mitglieder gebroadcastet bzw. geflutet. Ein Broadcast ist hier notwendig, da selbst der Rendezvous-Knoten nicht genau weiß, wer die angefragte Datei in seiner Cloud lagert. Erhält der Beherberger der angefragten Datei die Anfrage durch das Broadcasting, so erstellt dieser eine Antwort-Nachricht mit einer zufällig ausgewählten TTL und reicht sie analog per Random Walk an seine Cloud-Nachbarn. Entsprechend wird die TTL der Antwort-Nachricht dann von jedem Cloud-Nachbarn, der die Antwort-Nachricht erhält, um 1 verringert, sofern die TTL ungleich

0 ist und leitet sie an einen zufälligen Nachbarn weiter bis schließlich ein Nachbar das Paket mit der TTL 0 erhält. Dieser wird dann die Antwort-Nachricht direkt an den Crossover-Peer der anfragenden Cloud senden. Kommt die Antwort-Nachricht beim Crossover-Peer an, so wird die Antwort-Nachricht entlang dem Random Walk Pfad, der für die Anfrage vorgenommen wurde, an den ursprünglichen Initiator der Anfrage weitergeleitet. Anfragen werden also in einer Cloud (LTI Topologie) initialisiert und terminieren auch in einer, was den Anfrager und Beherberger einer Datei nach außen hin verschleiert. Auch in der eigenen Cloud bleibt ein Anfrager oder Beherberger einer Datei verschleiert, da man in der Cloud mit Random Walks arbeitet, die verhindern sollen den Anfrager oder Beherberger einer Datei auffindig zu machen.

2.4.3 ACE

ACE (Anonymous Cloud Environments) ist mein eigenes Overlay, welches ich in den PeerfactSim.KOM Simulator [FG13] eingebunden habe. Wie der Name schon deuten lässt, beruht mein Overlay sehr stark auf Agyaat. Man könnte es auch als eine leichte Agyaat Modifikation bezeichnen. Alle Funktionalitäten und Arbeitsweisen, die in Abschnitt 2.4.2 aufgezählt wurden, werden in ACE auch übernommen. Der ausschlaggebende Punkt, wieso ich ACE entwickelt habe ist, dass viele Fragestellung aus dem Agyaat Paper nicht hervorgehen. Wie werden Dateien publiziert, wie genau wird die DHT von den Peers des Netzwerkes verwaltet, wie bestimmt man im Falle von Churn einen neuen Rendezvous-Knoten einer Cloud? Alle diese Fragestellungen werden im Agyaat Paper nicht im Detail behandelt. Es wird lediglich darauf hingewiesen, dass das Wegfallen von Knoten und deren Aufgabengebiete wie es in DHT basierten P2P-Overlays üblich an andere Knoten übergeben werden können. Auf diese Fragestellungen und Probleme werde ich in ACE eingehen und in den Kapiteln 3 und 4 ausarbeiten.

2.5 Techniken zur Anonymisierung

Zum Abschluss dieses Kapitels möchte ich nochmal grob alle erwähnten Anonymisierungsverfahren, die bei einem Design eines anonymen DHT basierten P2P-Overlays berücksichtigt werden sollten, zusammenfassen und deren Vor- und Nachteile aufweisen.

2.5.1 Anonyme DHT

Ein großes Problem von DHT basierten P2P-Overlays ist es, dass ein Beherberger einer Datei durch einen Lookup auffindig gemacht werden kann, da in einem DHT Eintrag zu einem Key bzw. Schlüssel der Beherberger der Datei in Form von IP-Adresse und Portnummer abgelegt wird. Keys werden bei

DHT basierten Overlays auf Peers bzw. Knoten abgebildet, wodurch die *Empfängeranonymität* nicht gewährleistet werden kann. Um den Beherberger einer Datei zu verschleiern, werden in Achord (siehe Abschnitt 2.4.1) Keys bzw. Schlüssel auf Values bzw. Werte abgebildet. In Agyaat wiederum werden Keys auf Clouds abgebildet. Ein DHT Eintrag zu einem bestimmten Key sollte als Value also nicht den Beherberger der Datei beinhalten, sondern eine andere Information anhand der man trotzdem die Möglichkeit besitzt die Datei zu suchen und zu beziehen. Dieser Ansatzt ist unbedingt erforderlich, um die *Empfängeranonymität* zu bewahren und darf beim Design eines anonymen DHT basierten P2P-Overlays nicht fehlen.

2.5.2 Rekursives Routing

Rekursives Routing bzw. Tunneling wird in vielen DHT basierten P2P-Overlays wie z.B Freenet und Achord verwendet und auch in unstrukturierten P2P-Netzwerken wie z.B Gnutella, um sowohl den Sender und Empfänger einer Nachricht zu verschleiern. Es liegt auf der Hand, dass ein iteratives Routing, wo ein Knoten Schritt für Schritt jeden einzelnen Knoten selbst anfragt, nicht in Frage kommt. Die rekursive Routingsstrategie kann in Freenet und Achord Sender und Empfänger einer Nachricht verschleiern, da diese Knoten nur die Knoten in ihrer unmittelbaren Umgebung kennen und eine eingeschränkte Sicht auf das Netzwerk besitzen. In Freenet kennt ein Knoten nur die Knoten aus seiner Routingtabelle und in Achord kennt ein Knoten seinen Predecessor, Successor, die Knoten aus seiner Fingertabelle und die Knoten aus seiner Successor Liste. In Chord ist es trotz rekursivem Routing, welches einen hohen Grad *Senderanonymität* bietet [DV04], möglich, Berechnungen durchzuführen um den Initiator auf einen bestimmten Id Bereich einzuschränken oder ihn sogar zu identifizieren. Rekursives Routing sollte daher mit der Einschränkungen der Sicht auf das Netzwerk eines Knoten kombiniert werden, um den Grad der *Senderanonymität* zu erhöhen.

2.5.3 Random Walks

Random Walks tragen dazu bei, den Initiator einer Nachricht zu verschleiern. Nachrichten werden solange zwischen Knoten hin- und hergeschickt, bis eine bestimmte Bedingung bei einem der Knoten in Kraft getreten ist. Die Anfrage wird dann von diesem Knoten abgesendet. Zusätzlich ermöglichen sie es, wenn der Knoten, welcher die Anfrage rausgesendet hat, eine Antwort erhält, diese dem Initiator der Anfrage zukommen zu lassen an, in dem jeder Knoten die Antwort demjenigen Knoten sendet, von dem er die Nachricht erhalten hat, bis schließlich der Initiator der Anfrage die Datei erhält. Somit stellen sie ein sehr nützliches Hilfswerkzeug auch im rekursiven Routing dar und spielen für die Gewährleistung der *Senderanonymität* eine sehr wichtige Rolle.

In Agyaat, siehe Abschnitt 2.4.2, bestimmt der Initiator einer Datei eine zufällige TTL (time-to-live). Jeder Knoten, der die Anfrage erhält, überprüft zunächst, ob die TTL 0 ist. Wenn nicht, verringert dieser Knoten die TTL um 1 und schickt die Anfrage an einen anderen zufälligen Knoten, bis eine Anfrage einen Knoten mit der TTL 0 erreicht, welcher die Anfrage dann raussendet. Es wird jedoch nicht spezifiziert, wie die zufällige TTL von dem Initiator gewählt werden soll. Sie sollte nicht zu groß und nicht zu klein sein, da z.B. eine zu große Anzahl bedeuten würde, dass eine Antwort einer Dateianfrage von allen Knoten auf dem Random Walk Pfad zu dem Initiator zurückgeleitet werden müsste und würde somit unnötig viel Traffic erzeugen. Auf der anderen Seite sollte sie nicht zu klein sein, da das Hin- und Herschicken einer Anfrage einen Angreifer neben dem Initiator auf andere Knoten aufmerksam macht. Die Wahl der TTL muss an etwas gebunden sein, was andere Knoten nicht erahnen können. Wählt z.B. jeder Knoten eine TTL aus 3 bis 7, so kann der Knoten, der eine Anfrage mit der TTL 3 oder 7 erhält, schließen, dass der Knoten, von dem er die Anfrage erhalten hat, der Initiator ist. Solche Arten von TTL's, die zufällig aus einem gegebenen Intervall gewählt werden, sind also unbrauchbar [Bor05].

Hier bietet sich die Strategie an, die Crowds (siehe Abschnitt 2.3.1) benutzt. Ein Knoten schickt eine Nachricht mit Wahrscheinlichkeit p_f an einen zufälligen anderen Knoten und mit Wahrscheinlichkeit $1 - p_f$ schickt der Knoten die Nachricht raus. Diese Strategie werde ich auch in meinem ACE Overlay übernehmen.

2.5.4 Clouds

Clouds [SGL06] sind untstrukturierte Netzwerke, die an DHT's angehangen werden. Anfragen werden in Clouds initialisiert, per Random Walk wird die Anfrage in der Cloud rumgeschickt, bis ein Knoten, welcher als *Crossover-Peer* bezeichnet wird, die Anfrage raussendet. Angekommen in der Cloud bzw. beim Rendezvous-Knoten der Cloud, wo sich der Beherberger der angefragten Datei befindet, wird die Anfrage geflutet. Der Beherberger der Datei wird eine Antwort Nachricht mit der Angefragten Datei erstellen und die Antwort wird dann ebenfalls per Random Walk in der Cloud rumgeschickt, bis sie von einem Knoten an den *Crossover-Peer* gesendet wird. Schließlich kann die Antwort in der Cloud des Initiators angekommen, durch den aufgebauten Random Walk Pfad an den Initiator der Anfrage geleitet werden.

Viele Anonymisierungsverfahren werden in Clouds vereint und ermöglichen somit einen hohen Grad an *Sender-* und *Empfängeranonymität*. Abgesehen davon, dass durch Clouds ein hoher Grad an Anonymität erreicht wird, fallen durch das Brodcasting bzw. Fluten und die Random Walks zusätzlicher hoher Traffic an. Dieser ist zwingend notwendig, um das Prinzip der Cloud zu verfolgen. In Kapitel 5 werde ich nochmal auf den Traffic, den mein Cloud basiertes DHT Overlay erzeugt, auswerten und erörten.

Im Allgemeinen halte ich Clouds für eine sehr sinvolle Strategie um den Sender und Empfänger einer Nachricht zu verschleiern, weshalb ich mich dazu entschlossen habe ein Cloud basiertes Overlay zu implementieren und auszuwerten. Aufgrund der in Abschnitt 2.4.3 angeführen Problemstellungen, habe ich mich dazu entschlossen, ACE, bzw. eine modifizierte Version von Agyaat zu entwerfen, auf die ich in den folgenden Kapiteln im Detail eingehen werde.

Kapitel 3

Design der Anonymous Cloud Environments

In diesem Kapitel möchte ich die Funktionalitäten des Overlays ACE (Anonymous Cloud Environments) darlegen und die Implementierung von ACE anhand von Pseudocodes verdeutlichen. Unter anderem werde ich die Modifikationen von Agyaat, die ich für ACE vorgenommen habe, vorstellen. In dem Folgekapitel werde ich dann detaillierter auf die einzelnen Klassen und deren Beziehungen zueinander eingehen und wie ich bestimmte Sachen implementiert habe. Vorab möchte ich erwähnen, dass sowohl Agyaat als auch ACE die Funktionalitäten von Chord benutzen und auf Chord basieren. Aus diesem Grund konnte ich mir die Implementierung von EduChord, welche bereits im PeerfactSim.KOM Simulator [FG13] von Tobias Amft eingebunden war und eine Implementierung von Chord darstellt, zu Nutzen machen und für meine Zwecke benutzen und modifizieren.

3.1 Main-Ring

Wie bereits erwähnt, ist ACE ein Cloud- und DHT basiertes P2P-System, welches im Grunde eine Modifikation von Agyaat darstellt. ACE besteht aus 2 Ringen. Der eine Ring wird als Main-Ring und der andere als R-Ring (Rendezvous-Knoten Ring) bezeichnet. Möchte ein Knoten dem ACE Overlay beitreten, so muss er zunächst dem Main-Ring beitreten. Der Main-Ring ist wie ein üblicher Chord Ring zu betrachten, welcher dieselben Eigenschaften und Spezifikationen besitzt. Der Main-Ring dient in ACE nur dazu, um den neuen Rendezvous-Knoten einer Cloud zu bestimmen. Denn schließlich soll ein Rendezvous-Knoten, welcher als Eingangspunkt und Organisator einer Cloud dient, auch mal von einem anderen Knoten abgelöst werden. Dabei wird der Name der Cloud mittels der konsistenten Hashfunktion *SHA-1* gehasht, um die Cloud-Id zu erhalten. Der aktuelle Rendezvous-Knoten wird die *find_successor*( ) Methode (**Algorithmus 1**) mit der Cloud Id als Übergabeparameter aufrufen. Dieser Successor wird dann der neue Rendezvous-Knoten der Cloud. Die Wahl eines neuen Rendezvous-Knotens kann periodisch ausgeführt werden. Wichtig ist, dass die *find_successor*( ) Methode bei Bestimmung eines neuen Rendezvous-Knoten auf dem Main-Ring ausgeführt wird und nicht

auf dem R-Ring. Die oben aufgeführten Methoden, sind aber für beide Ringe, sowohl dem Main-Ring als auch den R-Ring, verwendbar. Die Wahl eines neuen Rendezvous-Knoten wird in **Algorithmus 3** verdeutlicht.

Algorithmus 1 Bestimmung eines Successor einer Id (siehe Chord Paper [SMK+01]).

Voraussetzung: $id \in [0, 2^{160} - 1]$.

```
1: n.find_successor(id)
2: {
3:    if (id ∈ (n, n.successor))
4:       return n.successor;
5:    else
6:       n' ← n.closest_preceding_node(id);
7:       return n'.find_successor(id);
8:    endif
9: }
```

Algorithmus 2 Bestimmung des höchsten Predecessor in der Fingertabelle eines Knotens zu einer gegeben Id (siehe Chord Paper [SMK+01]).

Voraussetzung: $id \in [0, 2^{160} - 1]$.

```
1: n.closest_preceding_node(id)
2: {
3:    for i = m downto 1
4:       if(finger[i] ∈ (n, id))
5:          return finger[i];
6:       end if
7:    end for
8:    return n;
9: }
```

Algorithmus 3 Bestimmung eines neuen Rendezvous-Knoten.

Voraussetzung: *n* ist aktueller Rendezvous-Knoten , *id* entspricht der Cloud Id des Rendezvous-Knoten

```
1: n.find_next_rendezvous_node( )
2: {
3:    new_rendezvous_node ← n.find_successor(n.cloud_id);
4:    return new_rendezvous_node;
5: }
```

Sowohl der Main-Ring als auch der R-Ring sind beides Chord Ringe, die bestimmte Operationen periodisch vornehmen müssen, damit das Netzwerk einen stabilen Zustand erhalten kann. Die *stabilize*( ) und die *fix_fingers*( ) Methode (siehe **Algorithmus 4 und Algorithmus 5**) von Chord sind für beide Ringe unverzichtbar. Da der Main-Ring als ein normaler Chord Ring betrachten werden kann, ist das trivial. Der R-Ring wiederum, welcher auch ein Chord-Ring ist, spielt aber eine wesentlich wichtigere Rolle als der Main-Ring. Alle Anfragen werden über den R-Ring geroutet, da die Clouds an den R-Ring angehangen werden bzw. durch ihre Rendezvous-Knoten auf dem R-Ring vertreten werden. Da der R-Ring analog zu Chord ein effektives Routing in $O(\log(N))$ gewährleisten soll, ist die *fix_fingers*( ) Methode unverzichtbar. Die *stabilize*( ) Methode ist ebenfalls unverzichtbar, da die Rendezvous-Knoten ebenso auf das Wegfallen eines Rendezvous-Knotens reagieren müssen und neue Rendezvous-Knoten, die neu hinzugekommene Clouds repräsentieren, müssen sich stets an ihren entsprechenden Positionen auf dem R-Ring positionieren.

Algorithmus 4 Periodische Operation, die stets den aktuellen Successor eines Knoten bestimmt (siehe Chord Paper [SMK+01]).

```
1:  n.stabilize( )
2:  {
3:     possible_new_successor ← n.successor.predecessor
4:     if (possible_new_successor ∈ (n, n.successor))
5:        n.successor ← possible_new_successor;
6:     end if
7:     n.successor.notify(n);
8:  }
9:
10: n.notify(n')
11: {
12:    if (n is rendezvous-node)
13:       if (n.predecessor is null or n′ ∈ [n.predecessor, n))
14:          n.predecessor ← n′
15:       end if
16:    else
17:       if (n.predecessor is null or n′ ∈ (n.predecessor, n))
18:          n.predecessor ← n′;
19:       end if
20:    end if
21: }
```

Algorithmus 5 Periodische Operation, welche die Fingertabelleneinträge aktualisiert (siehe Chord Paper [SMK+01]).

Voraussetzung: m ist die Anzahl der Fingertabelleneinträge.

```
1: n.fix_fingers( )
2: {
3:     next ← next + 1;
4:     if (next > m)
5:         next ← 1;
6:     end if
7:     finger[next] ← find_successor(n + 2^(next−1));
8: }
```

Die *stabilize*( ) Methode funktioniert auf beiden Ringen im Grunde analog nur mit dem Unterschied, dass ein Rendezvous-Knoten seinen Predecessor auch aktualisiert, wenn ein Knoten ihn mit der selben Id wie seines Predecessors benachrichtigt, um ihm mitzuteilen, dass er sein aktueller Successor sei. Denn Rendezvous-Knoten einer Cloud können sich mit der Zeit ändern, wobei die Id der Rendezvous-Knoten einer Cloud immer die Cloud Id besitzen.

3.2 R-Ring

Hier werden alle Funktionalitäten und Aufgaben des R-Ringes vorgestellt. Außerdem werden die Modifikationen, die an Agyaat vorgenommen wurden, hier im Detail besprochen. Eine Sache, die ACE von Agyaat unterscheidet, ist das in ACE nur eine Hashfunktion (*SHA-1*) benutzt wird. Somit gibt es für jede Cloud zu einem bestimmten Zeitpunkt immer nur einen Rendezvous-Knoten. In Agyaat sind wiederum mehrere Rendezvous-Knoten zur gleichen Zeit zulässig, welche durch Verwendung mehrerer Hashfunktionen möglich ist, um die Last und Aufgaben des Rendezvous-Knotens auf mehrere Knoten zu verteilen. Ein Nachteil ist hier aber, dass das Wegfallen mehrerer Rendezvous-Knoten im Falle von Churn , was bedeutet, dass andere Knoten für diese einspringen müssen und das dies durch aufwendige Protokolle geregelt werden muss. Zudem geht das Agyaat Paper nicht drauf ein wie man das Wegfallen eines oder mehrerer Rendezvous-Knoten behandelt und auch im Allgemeinen wird im Paper auch nicht auf Churn eingegangen. ACE bietet einem Rendezvous-Knoten durch einen sogenannten Pinger, welcher auch ein Knoten aus der selben Cloud ist, Hilfe, indem dieser einige Aufgaben des Rendezvous-Knotens übernimmt. Insbesondere wird der Pinger dafür zuständig sein, zu überprüfen, ob alle Mitglieder der Cloud noch erreichbar sind und wird alle Mitglieder der Cloud darunter auch dem Rendezvous-Knoten mitteilen, welche Nachbarn noch zu erreichen sind. Einfachheitshalber sind außerdem alle Knoten miteinanader vernetzt. D.h alle Knoten aus der Cloud kennen

sich untereinander. Ich habe auch eine weitere Version implementiert, in der sich innerhalb der Cloud nur Freunde kennen. Es ist aber zwingend notwendig, dass der Rendezvous-Knoten und auch Pinger jeden Knoten der Cloud kennen, damit das System hinter Clouds funktionieren kann. Auf weitere Details werde ich in diesem und dem folgenden Kapitel noch eingehen.

3.2.1 Erstellen und Beitreten von Clouds

Bevor ein Knoten eine Cloud erstellt, sollte er zuerst dem Main-Ring beigreteten sein. Dann kann er die *find_successor*( ) Methode aufrufen und ihr die Cloud Id übergeben, die man durch das Hashen des Namens der Cloud erhält. Wichtig ist hier, dass die Operation auf dem R-Ring und nicht auf dem Main-Ring ausgeführt wird. Da Rendezvous-Knoten auf dem R-Ring die Id ihrer Cloud besitzen, kann man die Id des Successor, die man mittels der *find_ successor( )* Methode auf dem R-Ring ermittelt hat, mit der Cloud Id vergleichen. Besitzt der Successor dieselbe Id wie die Cloud Id, so existiert diese Cloud bereits und der Knoten, der die Cloud erstellen wollte, kann dann dieser Cloud beitreten. Ansonsten kann der Knoten die Cloud erstellen, in dem er zunächst seine entsprechende Position auf dem R-Ring einnimmt bzw. dem R-Ring beitritt. Der Knoten wird dann gleichzeitig mit dem Erstellen der Cloud der Rendezvous-Knoten dieser Cloud. Analog funktioniert dies für Knoten, die einer bestimmt Cloud beitreten möchten. Diese suchen zunächst den Rendezvous-Knoten der Cloud ihrer Wahl auf dem R-Ring und kontaktieren diesen. Der Rendezvous-Knoten wiederum gewährt einem Knoten dann den Beitritt und ernennt ihn zum Pinger, falls die Cloud noch keinen Pinger besitzt. Sollte es noch keinen Rendezvous-Knoten auf dem R-Ring zu seiner Cloud geben, so kann dieser Knoten die Cloud erstellen und gilt dann als der Rendezvous-Knoten der Cloud. Das Erstellen und Beitreten von Clouds wird in **Algorithmus 6** und **Algorithmus 7** demonstriert.

Algorithmus 6 Suche einer Cloud auf dem R-Ring.

Voraussetzung: n' ist ein Rendezvous-Knoten auf dem R-Ring, der als Einstiegspunkt dient.

```
1:  n.search_ cloud(n',cloud_ name)
2:  {
3:     n.cloud_ id ← hash(cloud_ name);
4:     successor ← n'.find_ successor(cloud_ name);
5:     if (successor.id is equal to cloud_ id)
6:        n.create_ cloud( );
7:     else
8:        n.rendezvous_ node ← successor;
9:        n.rendezvous_ node.join_ request(n);
10:    endif
11: }
```

Algorithmus 7 Rendezvous-Knoten gewährt Knoten den Beitritt in die Cloud.

Voraussetzung: n' ist ein Rendezvous-Knoten.

```
1:  n.join_request(n')
2:  {
3:      n.neighbor_list.add(n');
4:      if (n.pinger is null)
5:          n.pinger ← n';
6:          n'.become_pinger(n.neighbor_list);
7:      else
8:          n'.join_reply(n.neighbor_list,n.pinger);
9:      end if
10: }
```

Algorithmus 8 Knoten bekommt Eintritt in die Cloud.

```
1:  n.join_reply(neighbor_list,pinger)
2:  {
3:      n.neighbor_list ← neighbor_list;
4:      n.pinger ← pinger;
5:  }
6:
7:  n.become_pinger(neighbor_list)
8:  {
9:      n.neighbor_list ← neighbor_list;
10:     n.pinger ← pinger;
11: }
```

3.2.2 Funkionalitäten des Rendezvous-Knotens und des Pingers

Der Rendezvous-Knoten spielt eine sehr wichtige Rolle. Alle Knoten, die der Cloud beitreten, werden von ihm registriert und an die anderen Cloud Mitglieder mitgeteilt. Zudem verwaltet der Rendezvous-Knoten die Routinginformationen über den R-Ring, welche er seinen eigenen Cloud Mitgliedern zur Verfügung stellt, wenn diese z.B eine Anfrage an eine bestimmte Cloud raussenden möchten, denn nur Rendezvous-Knoten speichern Informationen über den R-Ring. ACE arbeitet mit anonymen DHT's, d.h. dass Keys auf Clouds abgebildet werden. Wie die anonymen DHT's jedoch genau verwaltet werden, geht aus Agyaat nicht hervor. Dies wird in ACE jedoch genau spezifiziert. Wenn ein Key einer Datei auf eine bestimmte Cloud abgebildet wird, so heißt das nicht, dass der Beherberger dieser Datei sich in dieser Cloud befindet, sondern dass diese Cloud dafür zuständig ist zu wissen, wo die Datei

zu erreichen ist bzw. dass diese Cloud Auskunft darüber geben kann, in welcher Cloud die angefragte Datei vorzufinden ist. Ein Knoten kann nämlich viele verschiedene Dateien anbieten, die wiederum alle auf verschiedene Clouds abgebidet werden. Es wird in ACE vorgesehen, dass ein Knoten nur einer Cloud beitritt. So würde ein Knoten, der eine Datei publizieren möchte, zunächst die Cloud bestimmen, die für diese Datei zuständig ist, indem er den Successor dieser Datei Id auf dem R-Ring sucht und somit den Rendezvous-Knoten, dieser Cloud erhält. Dann wird er diesem Rendezvous-Knoten eine Nachricht schicken, dass der bestimmte Key in seiner Cloud vorzufinden ist. Der Rendezvous-Knoten wird dann einen Eintrag in seiner DHT anlegen, dass der bestimmte Key in der bestimmten Cloud vorzufinden ist. Rendezvous-Knoten verwalten also repräsentativ die DHT ihrer Cloud. Damit beim Verlassen des Netzwerkes der neue ausgewählte Rendezvous-Knoten alle DHT Einträge weiterhin verwalten kann, muss der aktuelle Rendezvous-Knoten seine DHT periodisch an alle Knoten der Cloud senden, damit diese bei Verlassen des Netzwerkes die Dateierreichbarkeit dieser Keys ermöglichen können. Außerdem ist es notwendig, dass er periodisch allen Mitgliedern der Cloud mitteilt, welche Knoten der Cloud neulich beigetreten sind, damit alle Knoten untereinander kommunizieren können und voneinander wissen (siehe **Algorithmus 9**).

Wie **Algorithmus 9** und **Algorithmus 10** verdeutlichen , wird eine Cloud durch das Zusammenspiel zwischen Rendezvous-Knoten und Pinger aufrecht erhalten. Der Rendezvous-Knoten wird als erstes den Pinger kontaktieren und ihm seine Liste mit allen Knoten, in der sich auch die neulich beigetreten Knoten befinden, von dem der Pinger und der Rest der Knoten in der Cloud nichts wissen, senden. Außerdem wird der Knoten seine Successor Liste und seine DHT mitschicken, damit beim Verlassen des Netzwerkes des Rendezvous-Knoten diese wichtigen Informationen nicht verloren gehen und vom neuen Rendezvous-Knoten genutzt werden können. Die Successor Liste ist dafür da, dass ein neuer Rendezvous-Knoten mittels der Successor Liste dem R-Ring beitreten kann. Und ebenso spielt die DHT eine wichtige Rolle, damit die Cloud auch weiterhin mit ihrem neuen Rendezvous-Knoten Anfragen für Keys, welche in seinen Aufgabenbereich fallen, beantworten kann. Der Pinger sendet dann jedem Knoten in der Liste, die er vom Rendezvous-Knoten erhalten hat, einen Ping, worauf diese mit einem Pong antworten müssen, damit der Pinger ermitteln kann, welche Knoten erreichbar sind (siehe **Algorithmus 10**, **Algorithmus 11** und **Algorithmus 12**). Anschließend wird der Pinger allen erreichbaren Knoten, die Liste mit allen erreichbaren Knoten in der Cloud, die Successor Liste und DHT des aktuellen Rendezvous-Knoten zusenden und jedem Knoten eine Priorität mitschicken. Die Priorität hat die Aufgabe beim Wegfallen eines Rendezvous-Knotens den restlichen Knoten zu ermöglichen einen neuen Rendezvous-Knoten zu ermitteln. Der Pinger weist dann jedem Knoten einen Wert (Priorität) von 1 bis r zu, wobei r die Anzahl aller Knoten abgesehen von dem Rendezvous-Knoten und dem Pinger ist (siehe **Algorithmus 10**).

Das Wegfallen eines Knotens, der weder Rendezvous-Knoten noch Pinger ist, kann eine Cloud gut verkraften. Fällt z.B der Rendezvous-Knoten oder der Pinger weg, so sind deren Aufgabengebiete umgehend an andere Knoten der Cloud zu verteilen, damit die Cloud weiterhin bestehen und im Netz-

werk erreichbar bleiben kann. Dabei spielen die Prioritäten, die der Pinger jedem einzelnen Knoten zuweist, eine wichtige Rolle. Beim Wegfallen von Knoten wegen Churn, sind folgende Fälle möglich, die wie folgt behandelt werden:

- Fall 1: Ein **normaler Knoten**, der weder Rendezvous-Knoten noch Pinger ist fällt weg. Das Wegfallen eines normalen Knotens wird allerspätestens vom Pinger erkannt, wenn dieser jeden Knoten anpingt, um zu überprüfen, welche Knoten erreichbar sind und kann den restlichen Knoten der Cloud mitteilen, dass dieser nicht mehr erreichbar ist. Des Weiteren muss nichts vorgenommen.

- Fall 2: Der **Pinger** einer Cloud fällt weg. Sofern der Rendezvous-Knoten nicht weggefallen ist, kann dieser einen neuen zufälligen Pinger bestimmen. Ein Rendezvous-Knoten wird dann darauf aufmerksam, dass der Pinger weggefallen ist, wenn er keine Antwort auf seine *cloud_update* Operation bekommt, in welcher er die Liste aller erreichbaren Knoten vom Pinger erwartet. Wenn der Rendezvous-Knoten jedoch auch weggefallen ist, so kommen die Prioritäten ins Spiel, die der Pinger den einzelnen Knoten periodisch zugewiesen hat. So wird jeder Knoten in der Cloud einen Timer stellen und $priorit\ddot{a}t \times waitForPinger$ Sekunden warten bis der Pinger sich meldet, wobei *waitForPinger* ein Parameter ist, den der Pinger den einzelnen Knoten mitteilt und dazu dient bis wann spätestens der Pinger erneut in der nächsten Periode diesen Knoten kontaktieren wird. Dieser Parameter sollte unter anderem von der Anzahl der Knoten in einer Cloud und der Performance des Pingers abhängen. Meldet sich der aktuelle Pinger in dieser Zeitspanne nicht, so wird sich der Knoten mit der niedrigsten Priorität beim Rendezvous-Knoten in Form eines Pinges melden. Erhält der Knoten kein Pong zurück, so kann dieser eine Nachricht an alle Knoten in der Cloud fluten, dass sowohl der aktuelle Pinger als auch der Rendezvous-Knoten weggefallen sind, so dass alle anderen Knoten ihre Timer stoppen können. Dieser Knoten wird dann der Rendezvous-Knoten und wählt dann einen zufälligen neuen Pinger.

- Fall 3: Der **Rendezvous-Knoten** einer Cloud fällt weg. Sofern der Pinger der Cloud nicht weggefallen ist, kann dieser bemerken, dass der Rendezvous-Knoten weggefallen ist, da er periodisch auf den Rendezvous-Knoten wartet, dass dieser ihm eine Liste mit allein Knoten, darunter auch neuhinzugekommen Knoten sendet, damit der Pinger überprüfen kann, ob alle Knoten erreichbar sind. In diesem Fall kann der Pinger den neuen Rendezvous-Knoten bestimmen. Sollte der Pinger ebenfalls weggefallen sein, so verfahren die Knoten wie in Fall 2, dass derjenige Knoten mit der niedrigsten Priorität den Rendezvous-Knoten anpingen wird, da er den Pinger als nicht mehr erreichbar vermutet. Gleichzeitig wird der Knoten dann feststellen, dass der Rendezvous-Knoten auch weggefallen ist und kann das der Cloud mitteilen. Dieser Knoten wird dann auch der neue Rendezvous-Knoten und kann zufällig einen neuen Pinger ermitteln.

Algorithmus 9 Periodische Operation eines Rendezvous-Knotens, um die Cloud mit notwendigen Informationen zu versorgen.

Voraussetzung: *n* ist ein Rendezvous-Knoten.

```
1:  n.update_cloud( )
2:  {
3:     n.pinger.update_information(n.neighbor_list, n.successor_list, n.responsible_keys_dht);
4:  }
5:
6:  n.update_response(neighbor_list);
7:  {
8:     recently_joined_nodes_list ← n.recently_joined_nodes;
9:     n.neighbor_list ← neighbor_list;
10:    n.neighbor_list.add(recently_joined_nodes_list);
11: }
```

Algorithmus 10 Operation des Pingers, in welcher alle Knoten der Cloud über alle erreichbaren Knoten und allen wichtigen Informationen informiert werden.

Voraussetzung: *n* ist ein Pinger.

```
1:  n.update_information(neighbor_list,successor_list,responsible_keys_dht)
2:  {
3:     list_of_neighbors ← neighbor_list;
4:     n.successor_list ← successor_list;
5:     n.responsible_keys_dht ← responsible_keys_dht;
6:     n.neighbor_list.clear( );
7:     for node in list_of_neighbors
8:        node.ping( );
9:     end for
10:    rendezvous_node_priority ← 1 ;
11:    for node in n.neighbor_list
12:       node.retrieve_pinger_updates(n.neighbor_list, n.successor_list, n.responsible_keys_dht,
13:                                    rendezvous_node_priority)
14:       rendezvous_node_priority ← rendezvous_node_priority + 1;
15:    end for
16:    n.rendezvous_node.update_response(n.neighbor_list);
17: }
```

Algorithmus 11 Pinger überprüft, welche Knoten der Cloud immer noch erreichbar sind.

Voraussetzung: *n* ist ein Pinger.

```
1: n.wait_for_pong(n')
2: {
3:     n.neighbor_list.add(n');
4: }
```

Algorithmus 12 Knoten, die einen Ping erhalten, antworten mit einem Pong, um mitzuteilen, dass sie erreichbar sind.

```
1: n.ping( )
2: {
3:     n.pinger.wait_for_pong(n);
4: }
5:
6: n.retrieve_pinger_updates(neighbor_list,successor_list,n.responsible_keys_dht,
7:                          rendezvous_node_priority)
8: {
9:     n.neighbor_list ← neighbor_list;
10:    n.successor_list ← successor_list;
11:    n.responsible_keys_dht ← responsible_keys_dht;
12:    n.rendezvous_node_priority ← rendezvous_node_priority;
13: }
```

3.2.3 Datei Publizierung

Dateien bzw. Keys von Dateien werden auf Clouds abgebildet. Möchte ein Knoten also eine Datei publizieren und dem Overlay zugänglich machen, so muss er zunächst die Cloud bestimmen, die für diesen Key zuständig ist bzw. auf die dieser Key abgebildet wird. Es muss also eine *find_successor* mit dem Key als Übergabeparameter ausgeführt werden. Somit würde man den Successor dieses Keys bzw. dieser Id auf dem R-Ring bestimmen und den Rendezvous-Knoten der zuständigen Cloud erhalten. Wenn der Knoten, welcher die Datei publizieren möchte, die *find_successor* ausführen würde, so könnte er seine Anonymität verlieren, da man sofort schließen könnte, dass er die Datei publizieren möchte. Aus diesem Grund wird die *find_successor* Methode ein anderer Knoten für ihn ausführen (siehe **Algorithmus 13**). Zunächst wird der Knoten, der eine Datei publizieren möchte, eine Random Walk Nachricht erstellen, die wiederum eine FindSuccessor Nachricht enthält, welche den Successor seiner zu verbreitenden Id, bestimmen soll. Diese Random Walk Nachricht wird dann an einen zufällig ausgewählten Knoten der Cloud gesendet. Dieser wird die Random Walk Nachricht mit ei-

ner Wahrscheinlichkeit $\frac{(L-1)}{L}$ an einen anderen zufällig ausgewählten Knoten in der Cloud senden und mit Wahrscheinlichkeit $\frac{1}{L}$ wird er die FindSuccessor Nachricht auf den R-Ring senden (siehe **Algorithmus 16**). Der dabei entstehende Random Walk Pfad wird in etwa die Länge L besitzen. Da der Knoten, welcher die FindSuccessor Nachricht auf den R-Ring routen möchte und den Einstiegspunkt auf dem R-Ring braucht, wird dieser den Rendezvous-Knoten seiner Cloud kontaktieren (siehe **Algorithmus 15**). Die FindSuccessor Nachricht wird ebenfalls einen Anfrage Pfad auf dem R-Ring aufbauen, sodass alle Knoten auf dem Anfrage Pfad die Antwort zurückleiten können, bis schließlich der Knoten, der die FindSuccessor Nachricht auf den R-Ring gesendet hat, die FindSuccessor Antwort erhält. Hat der Knoten nun eine FindSuccessor Antwort erhalten, so wandert diese Antwort rekursiv auf dem Random Walk Pfad zum Initiator der Anfrage zurück. Dieser weiß dann, welche Cloud auf dem R-Ring für seine Datei bzw. für die Id zuständig ist und schickt eine DistributeData Nachricht raus, welche die Id der zu verbreitenden Datei und seine Cloud Id enthält, die wieder mittels Random Walk in der Cloud herumgeschickt wird, bis schließlich ein Knoten die Nachricht an den Rendezvous-Knoten der verantwortlichen Cloud für diese Id schickt, denn die DistributeData Nachricht enthält die Kontaktinformationen von dem Rendezvous-Knoten, an welchen die Nachricht gesendet werden muss (siehe **Algorithmus 17**). Wenn die Nachricht beim Rendezvous-Knoten der verantwortlichen Cloud ankommt, so legt dieser Rendezvous-Knoten einen DHT Eintrag für den Key bzw. der Id an und speichert in welcher Cloud sich die Datei zu dieser Id befindet (siehe **Algorithmus 18**). Somit kann eine Datei im Netzwerk publiziert werden, ohne dass ein Knoten bzw. Peer seine Identität als Beherberger dieser Datei preisgeben muss.

Algorithmus 13 sollte periodisch ausgeführt werden, da mit der Zeit neue Clouds dazukommen können, auf welche dann bestimmte Datei Id's abgebildet werden und diese somit verantwortlich für diese Datei Id's sind. Abgesehen davon könnten alle Knoten einer Cloud wegfallen und eine neue Cloud wäre dann verantwortlich für die Datei Id's, für die die entfallenen Cloud verantwortlich war, so wie es in DHT basierten P2P üblich ist, wenn ein Knoten entfällt und dessen Aufgabenbereich an einen andere Knoten vergeben werden muss.

Algorithmus 13 Peridoische Operation, in der ein Knoten seine Dateien dem Netzwerk zugänglich macht, indem es den verantwortlichen Clouds mitteilt, dass diese in seiner Cloud gelagert werden.

```
1:  n.distribute_data( )
2:  {
3:    for data in n.list_of_data
4:      find_successor_message ← create_find_successor_message( );
5:      find_successor_message.target_id ← data.id;
6:      random_walk_message ← create_random_walk_message( );
7:      random_walk_message.payload ← find_successor_message:
8:      responsible_rendezvous_node ← n.random_neighbor( ).
9:                                          random_walk(random_walk_message);
10:     distribute_data_message ← create_distribute_data_message( );
11:     distribute_data_message.data_id ← data.id;
12:     distribute_data_message.cloud_id ← n.cloud_id;
13:     distribute_data_message.
14:                 responsible_rendezvous_node ← responsible_rendezvous_node;
15:     random_walk_message ← create_random_walk_message( );
16:     random_walk_message.payload ← distribute_data_message;
17:     n.random_neighbor( ).random_walk(random_walk_message);
18:   end for
19: }
```

Algorithmus 14 Crossover-Peer bestimmt Successor einer Id auf R-Ring.

```
1: n.find_successor_rring(id)
2: {
3:   entry_point_rring ← n.rendezvous_node.get_entry_point(id);
4:   successor ← entry_point_rring.find_successor( );
5:   return successor;
6: }
```

Algorithmus 15 Rendezvous-Knoten bestimmt Einstiegspunkt bzw. Eingangsknoten auf dem R-Ring.

Voraussetzung: n ist ein Rendezvous-Knoten.

```
1: n.get_entry_point(id)
2: {
3:   return n.closest_preceding_node(id);
4: }
```

Algorithmus 16 Random Walk Operation eines Knoten.

```
1: n.random_walk(random_walk_message)
2: {
3:   random_number ← get_random_number_between_1_and_L( );
4:   if (random_number < L)
5:     return n.random_neighbor( ).random_walk(random_walk_message);
6:   else
7:     message ← random_walk_message.payload;
8:     if (message is find_successor_message)
9:       return n.find_successor_rring(message.id);
10:    else if (message is distribute_data_message)
11:      n.send_distribute_data_message(message);
12:    else if (message is lookup_message)
13:      return n.send_lookup_message(message);
14:    else if (message is get_data_message)
15:      return n.send_get_data_message(message.target_id,message.cloud_id);
16:    else if (message is get_data_response)
17:      return n.send_get_data_response(message);
18:    end if
19:  end if
20: }
```

Algorithmus 17 Crossover-Peer schickt einem Rendezvous-Knoten einen DHT-Eintrag zu.

```
1: n.send_distribute_data_message(distribute_data_message)
2: {
3:   id ← distribute_data_message.id;
4:   cloud_id ← distribute_data_message.cloud_id;
5:   responsible_rendezvous_node ← distribute_data_message.responsible_rendezvous_node
6:   responsible_rendezvous_node.receive_distribute_data(id,cloud_id);
7: }
```

Algorithmus 18 Rendezvous-Knoten nimm Eintrag in seine DHT bzw. in die DHT seiner Cloud auf.

Voraussetzung: *n* ist ein Rendezvous-Knoten.

```
1: n.receive_distribute_data(id,cloud_id)
2: {
3:   n.responsible_keys_dht.add(id,cloud_id);
4: }
```

3.2.4 Dateien suchen und beziehen

Dateien zu suchen und zu beziehen funktioniert ähnlich wie Dateien zu publizieren. Um eine bestimmte Datei zu suchen bzw. einen Lookup nach einer bestimmten Id zu starten, wird der Initiator des Lookups eine Lookup Nachricht erstellen (siehe **Algorithmus 19**). Diese wird per Random Walk in der Cloud herumgeschickt, bis der Crossover-Peer den Lookup auf den R-Ring sendet (siehe **Algorithmus 16**). Den Einstiegspunkt bzw. Einstiegsknoten auf dem R-Ring bekommt der Crossover-Peer von seinem Rendezvous-Knoten (siehe **Algorithmus 15**). Die *Lookup* Operation wird von den Rendezvous-Knoten auf dem R-Ring ausgeführt und bestimmt den Rendezvous-Knoten, der einen Eintrag in seiner DHT für die angefragte Id enthält. Sie funktioniert analog zur *find_successor* Methode (siehe **Algorithmus 1**), nur dass sie beim Rendezvous-Knoten selbst terminiert und nicht bei seinem Predecessor (siehe **Algorithmus 21**). Die *Lookup* Operation muss beim Rendezvous-Knoten terminieren, damit dieser in der Lookup Antowort die Cloud, in der sich die Datei der angefragten Id befindet, angeben kann. Die Lookup Antwort enthält nur die Cloud Id, in der sich der Beherberger der angefraten Datei befindet, da Rendezvous-Knoten in ihrer DHT zu einer gegebenen Id die Cloud Id speichern und nicht den Rendezvous-Knoten dieser Cloud, denn es könnte ja sein,dass der Rendezvous-Knoten dieser Cloud mit der Zeit wegfällt. Mit der Cloud Id hingegen, kann ein Knoten dann den aktuellen Rendezvous-Knoten bestimmen und ist stets in der Lage die Cloud des Beherberges einer Datei zu kontaktieren. Die Lookup Antwort wird dann entlang des rekursiv aufgebauten Lookup Pfades auf dem R-Ring zum Crossover-Peer weitergeleitet. Dieser kann dann wiederum die Lookup Nachricht auf dem rekursiven aufgebauten Random Walk Pfad dem Initiator des Lookups zukommen lassen. Durch den Lookup weiß der Initiator nun, in welcher Cloud die Datei der angefragten Id zu finden ist. Um die Datei zu beziehen, erstellt der Knoten eine GetData Nachricht, welche die angefragte Datei Id und die Cloud Id des Beherbergs der Datei enthält und lässt sie analog zu den Lookups per Random Walk in der Cloud herumschicken (siehe **Algorithmus 22**). Der Crossover-Peer, der repräsentativ für den Initiator die Datei beziehen muss und die Datei dann entsprechend entlang des Random Walk Pfades an den Initiator leitet, muss zunächst die *find_successor* Methode mit der Cloud Id des Beherbergers der angefragten Datei als Übergabeparameter ausführen um den aktuellen Rendezvous-Knoten der Cloud herauszufinden (siehe **Algorithmus 20**). Hat der Crossover-Peer die Find Successor Antwort mit dem aktuellen Rendezvous-Knoten der Cloud des Beherbergers erhalten, so kann er die GetData Nachricht direkt an den Rendezvous-Knoten versenden. Der Rendezvous-Knoten wiederum flutet die GetData Nachricht in seiner eigenen Cloud, da er auch nicht weiß, welcher der Knoten seiner Cloud die Datei lagert (siehe **Algorithmus 24**). Der Beherberger der Datei wird dann nachdem er die GetData Nachricht erhalten hat, eine GetData Antwort mit der angefragten Datei erstellen, die an den Crossover-Peer der GetData Nachricht adressiert ist und wird diese Antwort in eine RandomWalk Nachricht packen, die wiederum einen Random Walk in der Cloud des Beherberges vornimmt (siehe **Algorithmus 25**). Der Knoten, welcher die Antwort rausschicken muss, schickt dann die Get Data Antwort an den Crossover Peer, an den die Antwort

gerichtet ist (siehe **Algorithmus 26**). Hat der Crossover Peer nun die GetData Antwort erhalten, wird sie dem Initiator der Anfrage entlang des Random Walk Pfades gesendet.

Algorithmus 19 Knoten startet einen Lookup.

```
1: n.start_lookup(id)
2: {
3:     lookup_message ← create_lookup_message( );
4:     lookup_message.target_id ← id;
5:     random_walk_message ← create_random_walk_message( );
6:     random_walk_message.payload ← lookup_message;
7:     cloud_id ← n.random_neighbor( ).random_walk(random_walk_message);
8:     return cloud_id;
9: }
```

Algorithmus 20 Crossover-Peer sendet eine Lookup Nachricht auf den R-Ring.

```
1: n.send_lookup_message(lookup_message)
2: {
3:     target_id ← lookup_message.target_id;
4:     entry_point_rring ← n.rendezvous_node.get_entry_point(target_id);
5:     cloud_id ← entry_point_rring.lookup(target_id);
6:     return cloud_id;
7:
8: }
```

Algorithmus 21 Rendezvous-Knoten überprüft, ob er verantworlich für die gegebene Id ist und leitet sie ggf. auf dem R-Ring weiter.

```
1: n.lookup(target_id)
2: {
3:     if (target_id in (n, n.successor))
4:         return n.successor.lookup(target_id);
5:     else if (n.responsible_keys_dht.contains(target_id)
6:         cloud_id ← n.n.responsible_keys_dht.get(target_id);
7:         return cloud_id;
8:     else
9:         n′ ← n.closest_preceding_node(target_id);
10:        return n'.lookup(target_id);
11:    endif
12: }
```

Algorithmus 22 Knoten führt die GetData Operation durch um eine Datei zu beziehen.

```
1: n.start_get_data(target_id)
2: {
3:     cloud_id ← n.start_lookup(target_id);
4:     get_data_message ← create_get_data_message( );
5:     get_data_message.target_id ← target_id;
6:     get_data_message.cloud_id ← cloud_id;
7:     random_walk_message ← create_random_walk_message( );
8:     random_walk_message.payload ← get_data_message;
9:     get_data_response ←n.random_neighbor( ).random_walk(random_walk_message);
10:    return get_data_response.data;
11: }
```

Algorithmus 23 Crossover-Peer schickt GetData Message raus.

```
1: n.send_get_data_message(target_id,cloud_id)
2: {
3:     get_data_message ← create_get_data_message( );
4:     get_data_message.sender ← n;
5:     get_data_message.target_id ← target_id;
6:     get_data_message.cloud_id ← cloud_id;
7:     rendezvous_node ← n.find_successor_rring(cloud_id);
8:     return rendezvous_node.receive_get_data_message(get_data_message);
9: }
10:
11: n.receive_get_data_response(get_data_response)
12: {
13:    return get_data_response;
14: }
```

Algorithmus 24 Rendezvous-Knoten bekommt eine Anfrage nach einer Datei, die sich in seiner Cloud befindet.

Voraussetzung: *n* ist ein Rendezvous-Knoten.

```
 1: n.receive_get_data_message(get_data_message)
 2: {
 3:    return n.broadcast_get_data_message(get_data_message);
 4: }
 5:
 6: n.broadcast_get_data_message(get_data_message)
 7: {
 8:    for node in n.neighbor_list
 9:       get_data_response ← node.get_data_request(get_data_message);
10:       if(get_data_response is not null)
11:          return get_data_response;
12:       end if
13:    end for
14:    return null;
15: }
```

Algorithmus 25 Knoten bekommt eine Anfrage nach einer Datei, die sich in seiner Cloud befindet. Lagert der Knoten die angefragte Datei, so erstellt er eine GetData Antwort und lässt sie per Random Walk verschicken.

```
 1: n.get_data_request(get_data_message)
 2: {
 3:    target_id ← get_data_message.target_id;
 4:    receiver ← get_data_message.sender;
 5:    if(n.list_of_data.cointains(target_id)
 6:       get_data_response ← create_get_data_response( );
 7:       get_data_response.data ← n.list_of_data.get(target_id);
 8:       get_data_response.receiver ← receiver;
 9:       random_walk_message ← create_random_walk_message( );
10:       random_walk_message.payload ← get_data_response;
11:       return n.random_neighbor( ).random_walk(random_walk_message);
12:    else
13:       return null;
14:    end if
15: }
```

Algorithmus 26 Knoten schickt GetData Antwort an den Crossover-Peer der anfragenden Cloud.

```
1: n.send_get_data_response(get_data_response)
2: {
3:     crossover_peer ← get_data_response.receiver;
4:     return crossover_peer.receive_get_data_response(get_data_response);
5: }
```

3.3 Modifikationen zur Erweiterung der Anonymität

3.3.1 Freundschaftsbeziehungen in der Cloud

In ACE ist es vorgesehen, dass alle Knoten in der Cloud alle miteinander vernetzt sind bzw. sich gegenseitig kennen. ACE bietet optional an, Freundesbeziehungen in Clouds zu simulieren. Jeder Knoten ist dann in seiner Cloud nur für seine Freunde sichtbar und routet alle möglichen Nachrichten auch nur über seine Freunde. Es ist aber trotzdem notwendig, dass ein Knoten immer mit dem Rendezvous-Knoten und dem Pinger vernetzt ist, denn der Rendezvous-Knoten muss Dateianfragen an jeden Knoten der Cloud fluten, damit die Anfrage an den Beherberger der Datei gelangen kann. Der Pinger unterstützt den Rendezvous-Knoten und überprüft welche Knoten zu erreichen sind und sollte deshalb auch jeden Knoten in der Cloud kennen. Ein großer Nachteil an Freundschaftsbeziehungen in Cloud ist, dass das gleichzeitige Wegfallen von Rendezvous-Knoten und Pinger zwar von den Knoten erkannt werden könnte und ein neuer Rendezvous-Knoten dezentralisiert von den Knoten gewählt werden könnte (analog zur Bestimmung eines Rendezvous-Knoten und Pingers in ACE ohne Freunschaftsbeziehungen), jedoch kennt der neu bestimmte Rendezvous-Knoten nur seine Freunde und kann nur diesen mitteilen, dass er der neue Rendezvous-Knoten ist. Alle anderen Knoten müssten, wenn sie das Wegfallen des Rendezvous-Knoten bemerken, der Cloud erneut beitreten, indem sie den Rendezvous-Knoten auf dem R-Ring suchen und eine Anfrage an diesen schicken.

Freundschaftsbeziehungen aus der Realität z.B. anhand von Facebook Freundschaftsgraphen [ACPP12], lassen sich in ACE nicht umsetzen, da zwei Knoten, die befreundet sind, auch in der selben Cloud sein müssen. Demnach könnte es Freundschaftsbeziehungen in der Realität geben, die sich nicht umsetzten lassen, weil der eine Knoten z.B zu Cloud1 und der andere Knoten zu Cloud2 gehört. Interessant wäre es aber trotzdem Freundschaftsbeziehungen Cloud übergreifend zu betrachten, da hier Cloud übergreifende Random Walks möglich wären und einem Angreifer nochmal zusätzlich erschweren würden, den Initiator einer Anfrage zu identifizieren. Somit könnte man die Senderanonymität zusätzlich erhöhen. Durch die Freundschaftsbeziehungen in ACE kann die *Anonymität durch Vertrauen*, welche beabsichtigt, dass nur Freunde miteinander kommunizieren, zum großen Teil erfüllt werden.

Der Rendezvous-Knoten und der Pinger müssen aber nicht zwingend Freunde von allen Knoten der Cloud sein, weshalb die *Anonymität durch Vertrauen* auch nicht vollständig gewährleistet werden kann.

3.3.2 Dynamische Dateilagerung

Um die Empfängeranonymität bzw. die Anonymität eines Beherbergers einer Datei zu erhöhen, kann man eine dynamische Lagerung von Dateien vornehmen. Dabei wird ein Beherberger einer Datei diese einen anderen Knoten übergeben, ohne zu wissen, wer die Datei zukünftig lagern wird. Dies kann z.B. durch einen Random Walk bewerkstelligt werden. Ein Knoten, der auf dem Random Walk Pfad die Datei erhält, könnte mit einer zufälligen Wahrscheinlichkeit die Datei an einen anderen Knoten weiterleiten oder entsprechend selbst lagern. Somit wäre es weder dem ursprünglichen Beherberger der Datei noch den anderen Knoten auf dem Random Walk Pfad möglich, zu wissen welcher Knoten nun die Datei lagert. Ein Knoten müsste bei diesem Ansatz jedoch darauf achten, dass der Knoten, dem er die Datei weiterleitet, auch wirklich erreichbar ist, indem er diesen z.B. anpingt. Um ganz sicher zu gehen, könnten Knoten eine Nachricht an ihren Vorgänger zurücksenden, damit diese wissen, dass die Datei bei einem Knoten angekommen ist. Denn wenn eine Datei an einen Knoten weitergeleitet wird, welcher offline ist, so würde der Random Walk unterbrochen werden und niemand würde die Datei lagern und dem Netzwerk weiterhin zur Verfügung stellen können.

Kapitel 4

Implementierung

Nachdem wir im vorherigen Kapitel ACE im Detail und in seinen Arbeitsweisen in Form von Pseudocodes und Erklärungen betrachtet haben, werde ich in diesem Kapitel noch etwas detaillierter auf die Implementierung zu sprechen kommen, indem ich die einzelnen Klassen und deren Beziehungen zueinander darlege und erkläre, wie ich bestimmte Sachverhalte in der Implementierung umgesetzt habe. Die Implementierung von ACE wurde in dem PeerfactSim.KOM Simulator [FG13] vorgenommen.

4.1 Darstellbarkeit von Knoten

Es gibt 3 Klassen von Knoten. `AceNode`, `PingerAceNode` und `RringAceNode`. Jeder Knoten im Overlay ist ein `AceNode`. Ist ein Knoten zusätzlich Rendezvous-Knoten, so hat es ebenso eine `RringAceNode` Instanz oder im Falle dass er ein Pinger sein sollte, besitzt es eine `PingerAceNode` Instanz. Da sich die `AceNode` und `RringAceNode` Klasse in vielerlei Hinsicht gleicht, z.B in ihren periodischen Operationen, erben diese beiden Klassen von `AbstractAceNode`, welche die Gemeinsamkeiten der beiden Klassen darstellt. Der `PingerAceNode` erbt jedoch nicht von `AbstractAceNode`, da er andere Funktionalitäten besitzt.

4.1.1 AbstractAceNode

Die `AbstractAceNode` Klasse, welche die Oberklasse der `AceNode` und `RringAceNode` Klasse ist und somit eine Grundbasis beider Klassen darstellt, besitzt folgende Instanzvariablen:

- `AceContact nodeContact`
  Enthält die IP-Adresse und Portnummer des Knoten.

- `AceContact successor`
  Enthält die Kontaktinformationen des *Sucessors* eines Knoten.

- `AceContact predecessor`
  Enthält die Kontaktinformationen des *Predecessor* eines Knoten.

- `AceContact[] fingerTable`
  Enthält die Fingertabelle, die für Routing Zwecke benutzt wird.

- `LinkedList<AceContact> successorList`
  Enthält die *Successor Liste*, welche beim Wegfallen des akutellen *Successors* nützlich ist.

- `Cloud cloud`
  Enthält die Cloud Instanz, zu welcher der Knoten zugehörig ist.

Außerdem enthält die `AbstractAceNode` Klasse folgende Methode:

- `AceContact getClosestPrecedingNode(BigInteger id)`
  Diese Methode bestimmt den höchsten Predecessor zu einer gegebenen Id in der Fingertabelle des Knoten. Sie wird von einem Knoten auf dem Main-Ring als auch von einem Rendezvous-Knoten auf dem R-Ring ausgeführt um den nächsten Knoten aus der Fingertabelle zu bestimmen, an den die Anfrage gesendet werden soll.

4.1.2 AceNode

Die `AceNode` Klasse stellt einen Knoten in dem ACE Overlay dar und spielt eine zentrale Rolle, da viele Aufgaben über diese Klasse verrichtet werden. Eine Instanz von der Klasse `AceNode` enthält folgende Instanzvariablen:

- `StabilizeOperation stabilizeOperation`
  Stabilize Operation von Chord, die auf dem Main-Ring ausgeführt wird.

- `FixFingersOperation fixFingersOperation`
  FixFingers Operation von Chord, die auf dem Main-Ring ausgeführt wird.

- `PeriodicLookupOperation periodicLookupOperation`
  Eine Operation, die periodisch Lookups nach zufälligen Id’s ausführt.

- `LinkedList<AceContact> cloudNeighbors`
  Enthält alle Knoten der Cloud, außer dem Pinger und dem Rendezvous-Knoten.

- `LinkedHashMap<BigInteger,BigInteger> responsibleDataKeys`
  Stellt die DHT der Cloud dar, die von jedem Knoten gespeichert wird.

- `LinkedList<BigInteger> offeredDataKeys`
  Enthält die Liste der Datei Id's, die der Knoten lagert.

- `DistributeDataOperation distributeDataOperation`
  Periodische Operation, welche Dateien im Netzwerk publiziert.

- `RringAceNode rRingNode`
  Ist nur initialisiert, wenn der Knoten ein Rendezvous-Knoten ist, ansonsten immer `null`.

- `LinkedList<AceContact> rRingSuccessorList`
  Enthält die *Successor Liste* des aktuellen Rendezvous-Knotens.

- `PingerAceNode pingerNode`
  Ist nur initialisiert, wenn der Knoten ein pinger ist, ansonsten immer `null`.

- `ArrayList<AceContact> friends`
  Eine Liste der Kontaktinformationen der Freunde eines Knotens. Wird nur initialisiert, wenn Freundschaftsbeziehungen simuliert werden.

- `PingFriendsOperation pingFriendsOperation`
  Periodische Operation, welche die Freunde eines Knotens periodisch anpingt, um zu überprüfen, ob diese erreichbar sind. Wird nur verwendet, wenn Freundschaftsbeziehungen simuliert werden.

Zu den wichtigen Methoden der `AceNode` Klasse zählen:

- `void join()`
  Mit dieser Methode versucht ein Knoten dem Main-Ring beizutreten.

- `void joinOperationSuccess(AceContact successor)`
  Beim Aufruf dieser Methode ist ein Knoten dem Main-Ring erfolgreich beigetreten. Diese Methode aktiviert dann wichtige periodische Operationen wie z.B. die `stabilizeOperation` und `fixFingersOperation`. Zudem wird mit der `joinCloudOperation` Methode der Join-Prozess eines Knotens seiner Cloud gestartet.

- `void joiningSuccessful()`
  Beim Aufruf dieser Methode ist ein Knoten seiner Cloud erfolgreich beigetreten. Operationen wie die `distributeDataOperation` und auch die `periodicLookupOperation` können nun gestartet werden. Werden Freundschaftsbeziehungen simuliert, so wird die `pingFriendsOperation` ebenfalls gestartet.

- `void joinCloudAgain()`
  Diese Methode wird von einem Knoten aufgerufen, wenn dieser nachdem er das Netzwerk verlassen hat, zu seiner Cloud wiederkehrt. Dann werden seine periodischen Operationen hiermit wieder gestartet.

- `messageArrived(final TransMsgEvent receivingEvent)`
  Alle eingehende Nachrichten werden von dieser Methode verarbeitet.

- `AceContact getNextHop()`
  Bestimmt den nächsten Knoten auf dem Random Walk Pfad. Wird mit Freundschaftsbeziehungen gearbeitet, so wird nur ein zufälliger Knoten aus den Freunden eines Knoten ausgewählt. Simuliert man jedoch keine Freunschaftsbeziehungen, so wird ein zufälliger Nachbarknoten ausgewählt.

- `AceContact isCrossOverPeer()`
  Im Allgemeinen gilt in ACE $L := 5$. Ein Knoten schickt bei einem Randon Walk eine Anfrage mit der Wahrscheinlichkeit $\frac{L-1}{L}$ an einen anderen Knoten und mit Wahrscheinlichkeit $\frac{1}{L}$ schickt der Knoten die Anfrage raus. Dies wird in dieser Methode simuliert. Muss ein Knoten die Anfrage weiterleiten, so wird der zufällige Knoten durch die `getNextHop()` Methode bestimmt und dieser als Rückgabewert zurückgegeben. Wenn der Knoten jedoch die Anfrage selbst absenden muss, so wird diese Methode als Rückgabewert die Kontaktinformationen des Knotens selbst zurückgeben, damit dieser Bescheid weiß, dass er die Anfrage versenden muss.

- `void connectivityChanged(ConnectivityEvent ce)`
  Im Falle von Churn wird diese Methode jegliche periodische Operationen eines Knoten stoppen. Kommt ein Knoten wieder online, so wird ebenfalls die Methode aufgerufen und ein Knoten kann wieder sowohl dem Main-Ring als auch seiner Cloud beitreten.

4.1.3 RringAceNode

Die `RringAceNode` Klasse repräsentiert einen Rendezvous-Knoten und ist als Instanzvariable der `AceNode` Klasse vertreten. Der Redenzvous-Knoten hat im Gegensatzt zu einem normalen Knoten

gewisse Aufgabenbereiche und Operationen, die man in eine eigene Klasse auslagern kann und somit einen besseren Überblick erhält. Bei einem Knoten, der Rendezvous-Knoten ist, könnten FindSuccessor Nachrichten sowohl vom R-Ring als auch vom Main-Ring eingehen. Um diese Problematik aus dem Weg zu räumen, warten Rendezvous-Knoten auf einem anderen Port als normale Knoten auf Nachrichten. Somit kann ein Knoten, der gleichzeitig Rendezvous-Knoten ist, entscheiden, von welchem Ring die Anfrage stammt.

Die `RringAceNode` Klasse enthält folgende Instanzvariablen:

- `AceNode mainRingNode`
  Ein Rendezvous-Knoten ist ein `AceNode` mit besonderen Fähigkeiten und Aufgabenbereichen.

- `static final short RRING_PORT`
  Der Port, unter welchem ein Rendezvous-Knoten seine eingehenden Nachrichten erwartet.

- `LinkedList<AceContact> neighborList`
  Ein `RringAceNode` führt eine Liste über alle Knoten in der Cloud. Der Pinger wird jedoch nicht in die Liste hinzugefügt, da er separat abgespeichert wird.

- `AceContact pinger`
  Der aktuelle Pinger der Cloud.

- `LinkedHashMap<BigInteger,BigInteger> responsibleDataKeys`
  Die DHT des Rendezvous-Knoten bzw. seiner Cloud, für dessen Id's die Cloud zuständig ist.

- `StabilizeOperation stabilizeOperation`
  Periodische Operation, die auch auf dem R-Ring nötig ist, um auf das Wegfallen von anderen Rendezvous-Knoten anderer Clouds reagieren zu können.

- `FixFingersOperation fixFingersOperation`
  Periodische Operation, welche die Fingertabellen aktualisiert und ein effektives Routing auch auf dem R-Ring ermöglichen soll.

- `UpdateCloudOperation updateCloudOperation`
  Diese Operation teilt dem Pinger der Cloud alle wichtigen Informationen mit, z.B. welche neuen Knoten der Cloud beigetreten sind, schickt diesem seine *DHT* bzw. die seiner Cloud und seine *Successor Liste* mit und bittet den Pinger darum zu überprüfen, welche Knoten noch erreichbar sind, um ihm dann eine Liste mit den erreichbaren Knoten schicken zu können.

Zu den wichtigsten Methoden der `RringAceNode` Klasse gehören:

- `void join()`
  Ein Rendezvous-Knoten tritt mit dieser Methode dem R-Ring bei.

- `void joinOperationSuccess(AceContact successor)`
  Beim Aufruf dieser Methode ist der Rendezvous-Knoten dem R-Ring erfolgreich beigetreten. Anschließend können dann wichtige periodische Operationen gestartet werden.

- `messageArrived(final TransMsgEvent receivingEvent)`
  Alle eingehenden Nachrichten werden von dieser Methode verarbeitet.

- `broadcastGetDataMessage(GetDataMessage getDataMessage)`
  Erhält ein Rendezvous-Knoten eine Anfrage nach einer Datei, die sich in seiner Cloud befinden soll, so flutet er diese Nachricht an jeden Knoten seiner Cloud.

- `void churnOccured()`
  Im Falle von Churn werden die periodischen Operationen dieses Rendezvous-Knotens gestoppt, wenn dieser wegfällt.

4.1.4 PingerAceNode

Eine `PingerAceNode` Instanz stellt den Pinger einer Cloud dar. Ein Knoten, der ein Pinger ist, besitzt eine `PingerAceNode` Instanz, die somit eine übersichtlichere Verarbeitung der Funktionalitäten und Operationen eines Pinger gewährleistet. Ein Pinger wartet ebenfalls auf einem speziellen Port auf eingehende Nachrichten, da ein Pinger über seine `PingerAceNode` Instanz angesprochen wird.

Die `PingerAceNode` Klasse enthält folgende Instanzvariablen:

- `AceNode ownerNode`
  Jeder Pinger ist auch ein `AceNode`.

- `Cloud cloud`
  Die Cloud-Instanz der Cloud des Pingers.

- `AceContact rendezvousNode`
  Der Rendezvous-Knoten der Cloud wird separat abgespeichert.

- `LinkedList<AceContact> successorList`
  Die *Successor Liste* seines Rendezvous-Knoten.

- `LinkedHashMap<BigInteger,BigInteger> responsibleDataKeys`
  Die *DHT* seines Rendezvous-Knoten bzw. seiner Cloud.

- `static final short pingerPort`
  Der Port, unter welchem ein Pinger auf eingehende Nachrichten wartet.

- `AceContact pingerContact`
  Die Kontaktinfomationen eines Pingers in Form von IP-Adresse und Portnummer.

- `PingerOperation pingerOperation`
  Die Operation, welche alle erreichbaren Knoten ermittelt und diesen alle wichtigen Informationen zusendet. Operation wird erst dann ausgeführt, wenn der Pinger eine `UpdateCloudMessage` von seinem Rendezvous-Knoten erhält und wird diesem per `UpdateCloudResponse` alle erreichbaren Knoten zukommen lassen.

Zu den wichtigsten Methoden der `PingerAceNode` Klasse gehören:

- `void messageArrived(final TransMsgEvent receivingEvent)`
  Diese Methode verarbeitet alle eingehenden Nachrichten.

- `void churnOccured()`
  Im Falle von Churn wird die `pingerOperation` gestoppt.

4.2 Die globale Cloud Instanz

Die `Cloud` Klasse repräsentiert eine Cloud, welche hauptsächlich die Aufgabe besitzt im Falle von Churn das Wegfallen von Knoten zu behandeln. Es ist offensichtilich, dass es eine solche globale Instanz in P2P-Systemen nicht geben kann. Da die Anonymität in meiner Arbeit im Vordergrund steht, habe ich das Churn Verhalten dahingehend abstrahiert, dass Knoten nicht selbst das Wegfallen von ihren Cloud Nachbarn behandeln, sondern das dies durch die Cloud Instanz erledigt wird. Außerdem wird die `Cloud` Instanz dazu genutzt, den Rendezvous-Knoten einer Cloud zu erhalten, anstatt ihn auf dem R-Ring zu suchen. Im Falle von Churn bedarf dies nämlich vieler Spezialfälle, die abgedeckt werden müssen, wenn ein Knoten zu seiner Cloud zurückkehren möchte, weshalb ich dies auch durch die Hilfe der `Cloud` Instanz abstrahiert habe.

Die Instanzvariablen der Cloud Klasse sind folgende:

- String cloudName
  Enthält den Namen der Cloud.

- BigInteger cloudId
  Enthält die Id der Cloud.

- LinkedHashMap<BigInteger,AceNode> dataStorage
  Enthält alle Id's der Dateien, die in dieser Cloud gelagert werden und deren Beherberger.

- LinkedList<AceNode> nodeList
  Eine Liste mit allen Knoten außer dem Rendezvous-Knoten und dem Pinger, denn diese werden separat abgespeichert.

- RringAceNode rendezvousNode
  Der aktuelle Rendezvous-Knoten der Cloud.

- LinkedList<AceContact> successorList
  Enthält die *Successor Liste* des aktuellen Rendezvous-Knotens.

- PingerAceNode pingerNode
  Der aktuelle Pinger der Cloud.

Zu den wichtigsten Methoden der Cloud Klasse gehören:

- void registerNode(AceNode node)
  Nimmt einen Knoten in die Liste aller Knoten der Cloud auf.

- void unregisterNode(AceNode node))
  Entfernt einen Knoten aus der Liste aller Knoten, wenn dieser offline geht.

- LinkedList<AceContact> getAvailableNodes()
  Gibt die Liste aller erreichbaren Knoten zurück. Diese wird vom Pinger in seiner PingerOperation benutzt, um alle ereichbaren Knoten zu erhalten. Das Anpingen der einzelnen Knoten vom Pinger wird ebenso durch diese Methode abstrahiert.

- void connectivityChanged(AceNode node)
  Im Falle von Churn wird ein Knoten, der wegfällt, diese Methode aufrufen und somit seiner Cloud-Instanz mitteilen, dass er offline geht. Die Cloud Instanz kann dann je nachdem ob der

wegfallende Knoten ein Rendezvous-Knoten oder Pinger gewesen ist, einen neuen Rendezvous-Knoten oder Pinger bestimmen.

4.3 Nachrichten

Die Implementierung von ACE enthält viele verschiedene Nachrichten, die für bestimmte Zwecke und Operationen vorgesehen sind. Alle Nachrichten in ACE erben von der Klasse `EduBaseMessage` von der EduChord Implementierung des PeerfactSim.KOM Simulators. Als Instanzvariable enthält die `EduBaseMessage` Klasse die `TransInfo's` von dem Sender und Empfänger der Nachricht. Die `TransInfo` Klasse enthält die IP-Adresse und die Portnummer, unter welcher ein Knoten erreichbar ist. Somit bildet eine `EduBaseMessage` Instanz die Grundbasis einer Nachricht.

ACE enthält folgende Nachrichten:

- `FindSuccessorMessage`
  Wird sowohl auf dem Main-Ring als auch auf dem R-Ring benutzt, um den *Successor* zu einer gegebenen Id zu suchen.

- `FindSuccessorResponse`
  Enthält den Successor zu einer gegebenen Id.

- `GetPredecessorMessage`
  Wird sowohl auf dem Main-Ring als auch auf dem R-Ring benutzt, um den Predecessor während einer `StabilizeOperation` anzufragen.

- `GetPredecessorResponse`
  Enthält den *Predecessor* und die *Successor Liste* eines Knotens.

- `NotifyMessage`
  Wird sowohl auf dem Main-Ring als auch auf dem R-Ring benutzt, um einen Knoten mitzuteilen, das dieser nun der *Successor* eines Knotens ist.

- `NotifyResponse`
  Dient dazu einem Knoten mitzuteilen, dass man erreichbar ist und damit der aktuelle *Successor* dieses Knotens bleibt.

- `GetEntryPointMessage`
  Wird von Knoten, die keine Rendezvous-Knoten sind, benutzt, um von ihrem Rendezvous-Knoten den Einstiegspunkt für eine gegebene Id auf dem R-Ring zu erhalten.

- `GetEntryPointResponse`
  Enthält den Einstiegspunkt zu einer gegebenen Id auf dem R-Ring.

- `DeliverDataMessage`
  Diese Datei enthält eine `GetDataResponse`, die von einem Beherberger einer Datei bei Nachfrage erstellt wird und per Random Walk durch die Cloud geschickt wird.

- `DistributeDataMessage`
  Dient dazu eine Datei zu publizieren und wird an den verantwortlichen Rendezvous-Knoten der gegebenen Datei Id gesendet.

- `GetDataMessage`
  Mit dieser Nachricht werden Anfragen nach Dateien gestartet.

- `GetDataResponse`
  Enthält eine angeforderte Datei.

- `LookUpMessage`
  Dient dazu, die Cloud Id des Beherbergers einer bestimmten Datei nachzuschauen.

- `LookUpResponse`
  Enthält die Cloud Id, in welcher sich der Beherberger einer bestimmten Datei befindet.

- `BecomePingerMessage`
  Ein Rendezvous-Knoten teilt einem Knoten, welcher seiner Cloud beitreten möchte, mit, dass die Cloud keinen Pinger besitzt und dass dieser als Pinger ausgewählt wurde.

- `BecomePingerResponse`
  Knoten teilt dem Rendezvous-Knoten mit, dass er von nun an Pinger ist und schickt ihm seine Kontaktinformationen, unter denen er als Pinger erreichbar ist.

- `JoinCloudMessage`
  Knoten schickt diese Nachricht an den Rendezvous-Knoten einer Cloud, um dieser Cloud beizutreten.

- `JoinCloudResponse`
  Enthält die Kontaktinformationen des Rendezvous-Knotens und des Pingers, falls dieser existiert.

- `PingerNotifyMessage`
  Enthält die *DHT* und *Successor Liste* des Rendezvous-Knoten und eine Liste aller erreichbaren Knoten und wird an alle Knoten der Cloud gesendet, außer dem Rendezvous-Knoten.

- `RandomWalkMessage`
  Enthält eine Nachricht, die per Random Walk in der Cloud des Initiators einer Anfrage versendet wird.

- `RandomWalkResponse`
  Enthält die Antwort einer Anfrage, die per Random Walk durch die Cloud gesendet wird.

- `UpdateCloudMessage`
  Eine Nachricht, die ein Rendezvous-Knoten an den Pinger sendet, welche alle Knoten der Cloud, darunter auch neu hinzugekommene Knoten, enthält, die der Pinger nicht kennt.

- `UpdateCloudResponse`
  Enthält alle erreichbearen Knoten der Cloud und wird an den Rendezvous-Knoten einer Cloud gesendet.

- `PingFriendMessage`
  Beim Simulieren von Freundschaftsbeziehungen benutzt ein Knoten diese Nachricht, um zu sehen, ob sein Freund erreichbar ist.

- `PongFriendMessage`
  Bekommt ein Knoten eine `PingFriendMessage`, so antwortet er mit einer `PongFriendMessage`, um mitzuteilen, dass er erreichbar ist.

4.4 Hilfsklassen

Hier werden die wichtigsten Hilfsklassen vorgestellt, mit deren Hilfe es möglich ist, bestimmte Operationen und Sachverhalte wie z.B Lookups nach zufälligen Id's und Dateianfragen zu starten oder Freundschaftsbeziehungen zwischen Knoten innerhalb der eigenen Cloud während der Simulation zu realisieren.

4.4.1 Verteilung von Knoten auf Clouds

die `CloudStartSettingsTool` Klasse bestimmt eine zufällige Verteilung der zu simulierenden Knoten auf die ausgewählten Clouds. Zudem bestimmt sie den Rendezvous-Knoten jeder einzelnen Cloud zufällig. Beim Simulieren von Freundschaftsbeziehungen bestimmt diese Klasse auch zufällig die Freundschaftsbeziehungen der einzelnen Knoten in einer Cloud. Beim Erstellen des ersten Knoten in der `AceNodeFactory` wird die `generateInfo(int numberOfClouds,int numberOfNodes)` Methode ausgeführt, welche dann die oben aufgezählten Aufgaben erledigt.

4.4.2 Zuweisung von zufälligen Datei Id's auf Knoten

Um die `PeriodicLookupOperation` oder die `PeriodicDataOperation` simulieren zu können, muss im Vorfeld festgelegt werden, welche Knoten welche Dateien bzw. Datei Id's lagern. Diese Aufgabe übernimmt die `DataKeyGenerator` Klasse, indem sie jedem Knoten die ausgewählte Anzahl an zufälligen Datei Id's zuweist. Wenn ein Knoten eine Datei bzw. Datei Id im Netztwerk publiziert hat und der verantwortliche Rendezvous-Knoten einen Eintrag in der DHT abgelegt hat, wird diese Id in der `availableDataKeys` Liste der `DataKeyGenerator` Klasse abgelegt, damit die periodischen Lookup Operationen zufällige Id's aus dieser Liste auswählen können.

4.4.3 Austausch der Kontaktinformationen bei Freundschaftsbeziehungen

Die `CloudStartSettingsTool` Klasse bestimmt zwar die Freundschaftsbeziehungen unter den einzelnen Clouds, jedoch existieren zu dem Zeitpunkt nicht alle Kontaktinformationen bzw. die `AceContact's` der einzelnen Knoten, da zu dem Zeitpunkt nur der erste Knoten erstellt wurde und nur dieser eine `AceContact` Instanz besitzt. Um jeden Knoten aber die Kontaktinformationen seiner Freunde zugänglich zu machen, wird die `FriendshipOrganizer` Klasse die Kontaktinformationen der Freunde jedes einzelnen Knotens aktualisieren, wenn diese ihrer Cloud erfolgreich beigetreten sind.

Kapitel 5

Simulation

Nachdem in den vorherigen Kapiteln Anonymität in P2P Netzwerken im Detail betrachtet wurde und mein eigenes Overlay ACE, welches unter dem Konzept der Anonymität konzipiert worden ist und in seinen Funktionalitäten und Arbeitsweisen vorgestellt wurde, möchte ich nun einige Simulationen ausführen und mein Overlay ACE unter gewissen Metriken hinsichtlich Funktionalität und Performance betrachten und mittels des Peerfact-Sim.KOM Simulators auswerten.

5.1 Metriken

Um eine Aussage über die Funktionalität und Performance von ACE treffen zu können, werden bestimmte Metriken an ACE ausgetestet und erörtert. Da ACE eine DHT darstellt, an welche unstrukturierte Netzwerke bzw. Clouds angehangen werden, ist es interessant zu sehen, wie die im Folgenden vorgestellten Metriken ausfallen werden.

- **Durschnittliche Hop Anzahl**
  Durchschnittliche Hop Anzahl aller `PeriodicDataOperation` während der gesamten Simulation. Auf der x-Achse werden die Anzahl der Knoten und auf der y-Achse die durschnittliche Hop Anzahl angegeben.

- **Durschnittliche Random Walk Pfadlänge**
  Die durchschnittliche Random Walk Pfadlänge aus allen Random Walk's, die während einer Simulation ausgeführt werden. Auf der x-Achse wird die Anzahl der Knoten und auf der y-Achse die Länge des Random Walk Pfades in Form von Anzahl der besuchten Hops angegeben.

- **Netzwerktraffic**
  Der Netztwerktraffic, der während der gesamten Simulation erzeugt wird. Auf der x-Achse werden die Anzahl der Knoten und auf der y-Achse der enstandene Netzwerktraffic in Bytes angegeben.

- **Anzahl an Nachrichten**
  Anzahl aller Nachrichten, die während einer Simulation durch das Netzwerk geschickt werden. Auf der x-Achse werden die Anzahl der Knoten und auf der y-Achse die Anzahl der Nachrichen angegeben.

- **Dateierreichbarkeit**
  Alle erfolgreichen Dateianfragen im Verhältnis zu den gesamten Dateianfragen während einer Simulation. Auf der x-Achse werden die Anzahl der Knoten und auf der y-Achse die Dateierreichbarkeit in % angegeben.

Die durchschnittliche Hop Anzahl der `PeriodicDataOperation` dient zum Nachweis der korrekten Arbeitsweise der Routingsstrategien von ACE, da die Anzahl der Hops, die beim Beziehen einer Datei durchlaufen werden, ungefähr kalkulierbar ist. Außerdem erhält man dadurch eine Aussage über die Performance und Effektivität von ACE. Das Prinzip der Clouds, welches mit Random Walks arbeitet, um Dateien von dem Beherberger einer Datei bis hin zum Initiator einer Dateianfrage zu übertragen, lässt die Vermutung aufstellen, dass der Netzwerkttraffic von ACE im Vergleich zu anderen DHT basierten P2P-Overlays wie z.B. Chord, die keinerlei Verschleierung vornehmen, um Knoten anonym zu halten, um einiges vervielfacht wird, weshalb es interessant ist den Netzwerktraffic auszuwerten. Zudem lässt sich durch die Auswertung der Anzahl der Nachrichten und des Netzwerktraffics zeigen, inwiefern die an die DHT angehangenen unstrukturierten Netzwerke bzw. Clouds, die beiden Metriken beeinflusst, denn es ist klar, dass in ACE der meiste Netzwerktraffic durch Random Walks zum Übertragen einer Datei von einer in die andere Cloud verursacht werden, in welcher jeder Knoten auf dem Random Walk Pfad die Datei an den nächsten Knoten weiterletet, bis schließlich der Initiator der Dateianfrage die angefragte Datei erhält. Da alle Anfragen in ACE auch über den R-Ring geroutet werden, und der R-Ring somit im Fokus steht, wird der Großteil aller Nachrichten von Knoten in den einzelnen Clouds initialisiert. Die Dateierreichbarkeit zeigt auch, ob das Prinzip hinter den Clouds funktioniert bzw. ob Knoten in Clouds verschleiert werden und trotzdem Dateien problemlos publizieren und beziehen können. Eine 100 prozentige Dateierreichbarkeit würde beim Simulieren ohne Churn die Korrektheit der Funktionalität von ACE und die Idee Clouds durch ihre Rendezvous-Knoten auf dem R-Ring zu repräsentieren, Anfragen und Nachrichten über den R-Ring zu routen, um Dateien zu publizieren, suchen und beziehen, bestätigen. Bei Simulationen mit Churn kann man zusätzlich einen Einblick darüber bekommen, inwiefern Dateien dann immer noch erreichbar sind. Die Auswertung der durchschnittlichen Random Walk Pfadlänge verschafft einen Einblick darüber, ob die durchschnittlich erwartete Pfadlänge tatsächlich angenommen wird.

5.2 Setup

Um die verschiedenen Einflüsse von bestimmten Ereignissen bzw. Modifikationen an ACE austesten, werden im Folgenden die verschiedenen Setups vorgestellt. Alle Simulationen haben eine Simulationszeit von **180 min**. Spätens kurz nach der **20 min** sollten alle Knoten dem Main-Ring beigetreten sein. Alle Knoten, abgesehen von den Rendezvous-Knoten, treten ab der **25 min** ihrer Cloud bei. Rendezvous-Knoten werden zufällig bestimmt und werden nachdem sie dem Main-Ring erfolgreich beigetreten sind, ihre Cloud erstellen und dem R-Ring beitreten. Alle periodische Operationen, die mit dem Beziehen oder Publizieren von Dateien zu tun haben, werden ab der **30 min** ausgeführt, da zu diesem Zeitpunkt alle Knoten ihrer Cloud beigetreten sein sollten. Jeder Knotet startet periodisch in jeder Minute einen Lookup nach einer zufälligen Id mittels der `PeriodicLookupOperation`. Erfolgt ein erfolgreicher Lookup nach einer gegebenen Id, so wird die die Datei zu dieser Id mittels der `PeriodicDataOperation` bezogen. Soll Churn simuliert werden, so wird Churn von der **50 min** bis zur **80 min** eingeschaltet und danach wieder abgeschaltet, sodass alle weggefallenen Knoten dem Netzwerk erneut beitreten.

5.2.1 Freundschaftsbeziehungen

Freundschaftsbeziehungen werden als Setup verwendet, um auszutesten, ob die *Anonymität durch Vertrauen* sich weitestgehend umsetzen lässt, ohne die Funktionalitäten und Performance der einzelnen Knoten in der Cloud einzuschränken. So soll gezeigt werden, dass zwar nur mit Freunden innerhalb der Cloud kommuniziert werden kann, ein Knoten aber in den vorgestellten Metriken keine Nachteile verzeichnet. Ein Knoten wird nur dann Dateien publizieren, suchen und beziehen, wenn er Freunde hat, die online sind und seine Anfragen und Nachrichten per Random Walk weiterreichen können.

- **F1**
  Freundschaftsbeziehungen werden während der Simulation nicht berücksichtigt. Alle Knoten einer Cloud kennen sich untereinander und jeder Knoten routet über den anderen Knoten.

- **F2**
  Freundschaftsbeziehungen werden während der Simulation benutzt. Knoten routen nur über ihre Freunde und kennen auch nur diese in der Cloud (abgesehen vom Rendezvous-Knoten und dem Pinger).

5.2.2 Churn

Churn bietet sich als Setup gut an, da es interessant ist zu sehen, wie Knoten auf das Wegfallen ihrer Cloud Nachbarn reagieren und wie Churn sich auf die oben genannten Metriken auswirkt. Zudem kann dadurch gezeigt werden, dass ein System auch beim Wegfallen von einigen Knoten wieder einen stabilen Zustand annehmen kann, welches ein wichtiges Merkmal von DHT basierten P2P-Overlays darstellt.

- **C1**
  Simulation wird ohne Churn ausgeführt. Alle Knoten bleiben während der Simulation online.

- **C2**
  Churn ist während der Simulation eingeschaltet. Knoten können während der Simulation offline gehen und wieder online kommen.

5.3 Ergebnisse

Im Folgenden werden die Ergebnisse der Simulation der oben angeführten Setups in allen möglichen Kombinationen vorgestellt und anschließend unter den vorgestellten Metriken ausgewertet und erörtert.

F1-C1

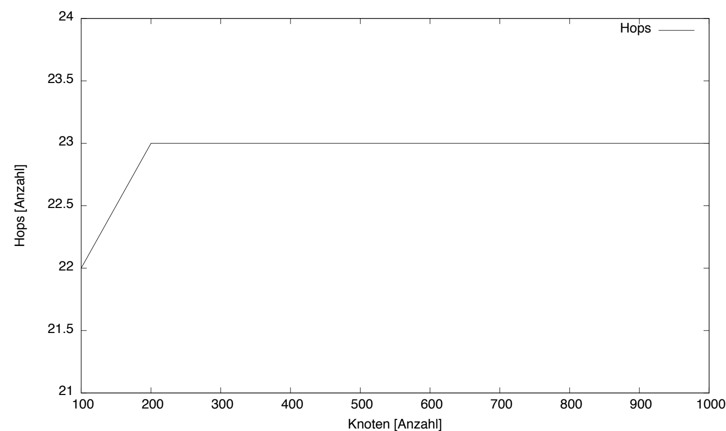


Abbildung 5.1: Durchschnittliche Hop Anzahl der `PeriodicDataOperation`

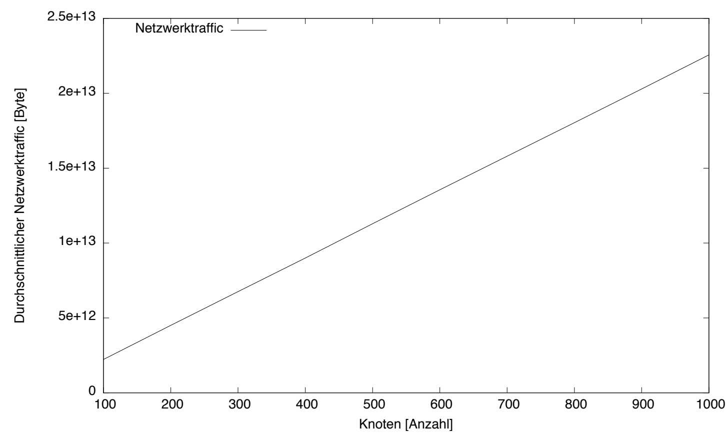


Abbildung 5.2: Durchschnittlicher Netzwerktraffic einer gesamten Simulation

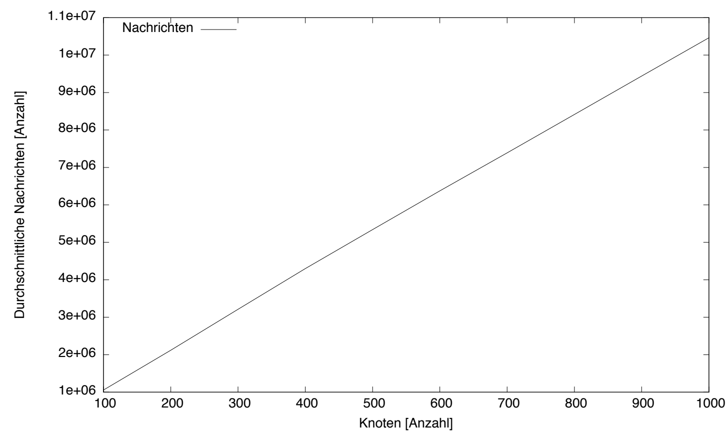


Abbildung 5.3: Durchschnittliche Anzahl aller Nachrichten einer Simulation

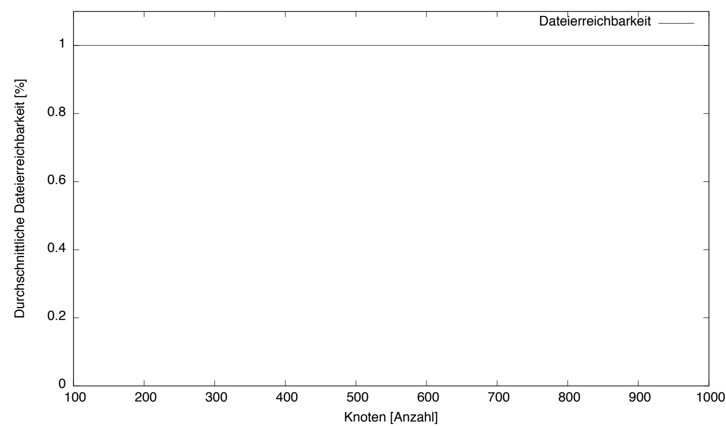


Abbildung 5.4: Durchschnittliche Dateierreichbarkeit von allen Dateianfragen einer Simulation

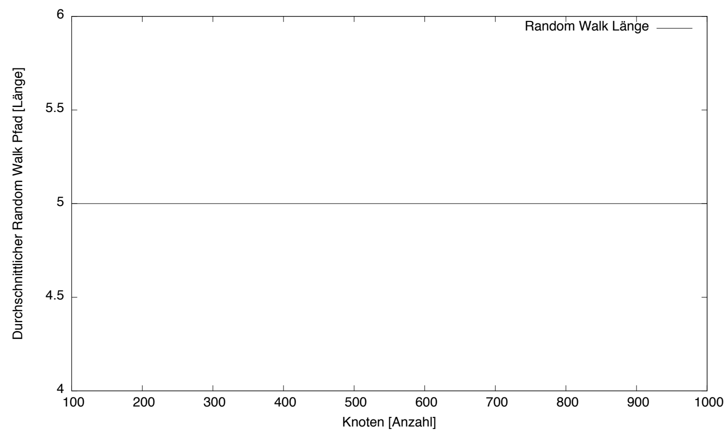


Abbildung 5.5: Durchschnittliche Länge eines Random Walk Pfades

F1-C2

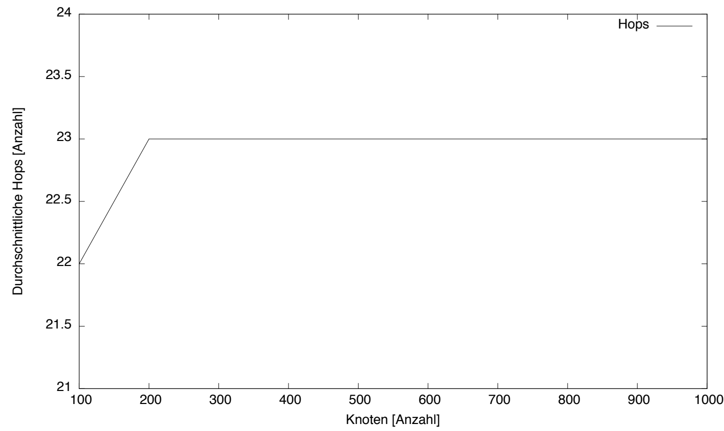


Abbildung 5.6: Durchschnittliche Hop Anzahl der `PeriodicDataOperation`

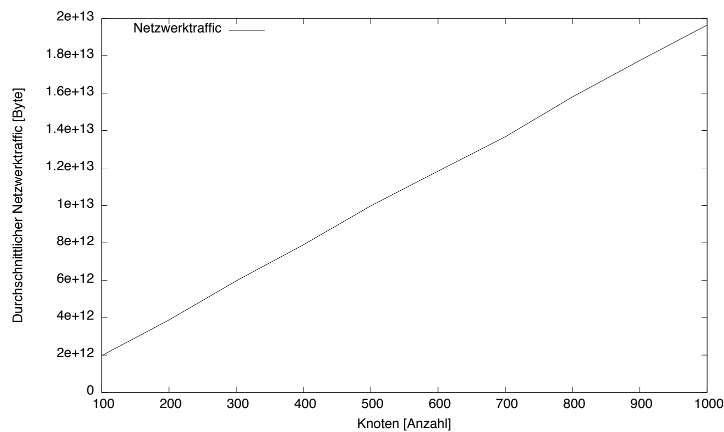


Abbildung 5.7: Durchschnittlicher Netzwerktraffic einer gesamten Simulation

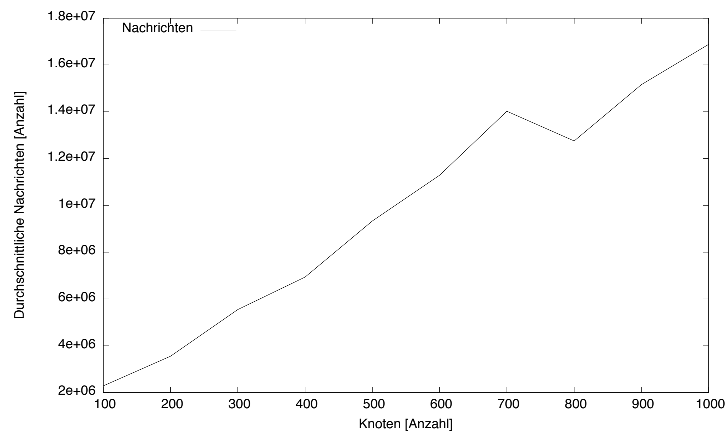


Abbildung 5.8: Durchschnittliche Anzahl aller Nachrichten einer Simulation

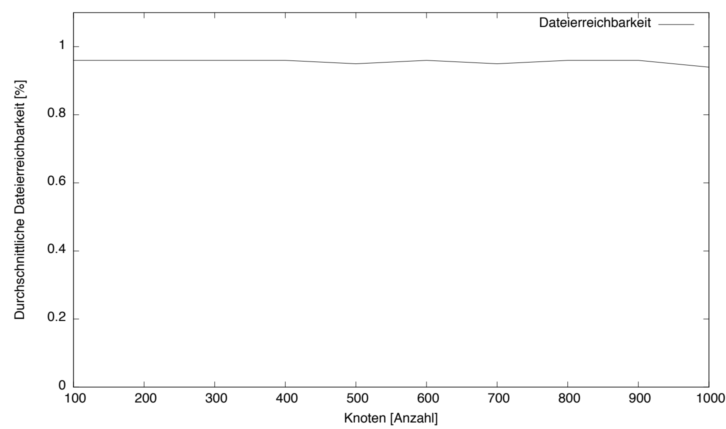


Abbildung 5.9: Durchschnittliche Dateierreichbarkeit von allen Dateianfragen einer Simulation

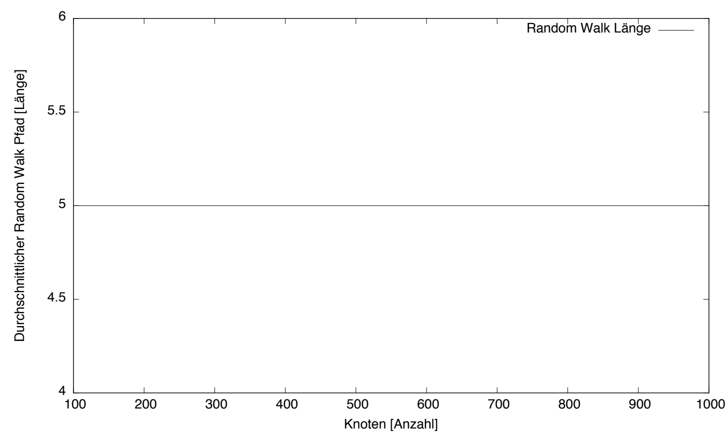


Abbildung 5.10: Durchschnittliche Länge eines Random Walk Pfades

F2-C1

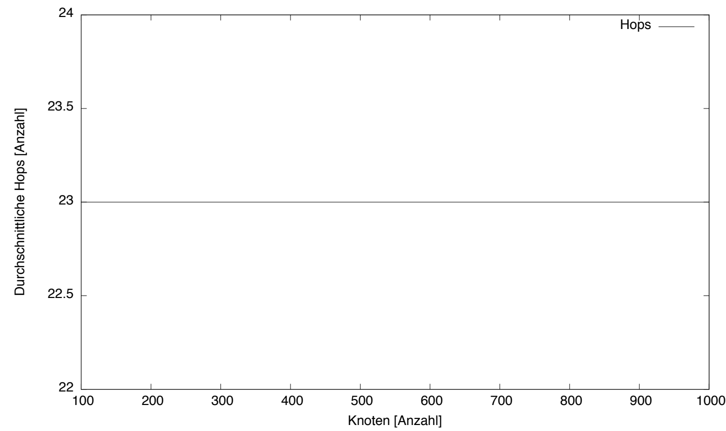


Abbildung 5.11: Durchschnittliche Hop Anzahl der `PeriodicDataOperation`

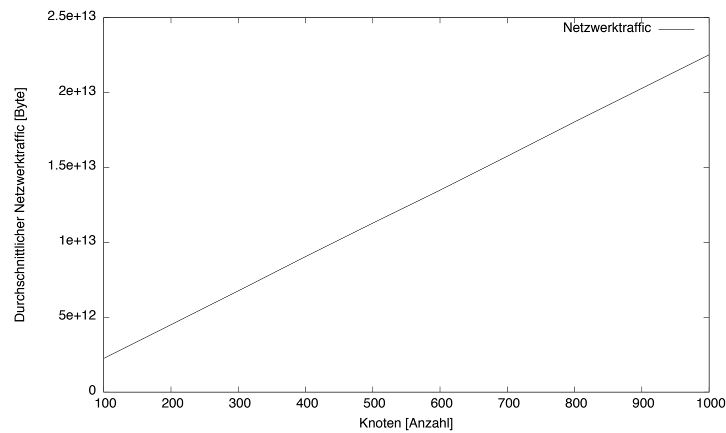


Abbildung 5.12: Durchschnittlicher Netzwerktraffic einer gesamten Simulation

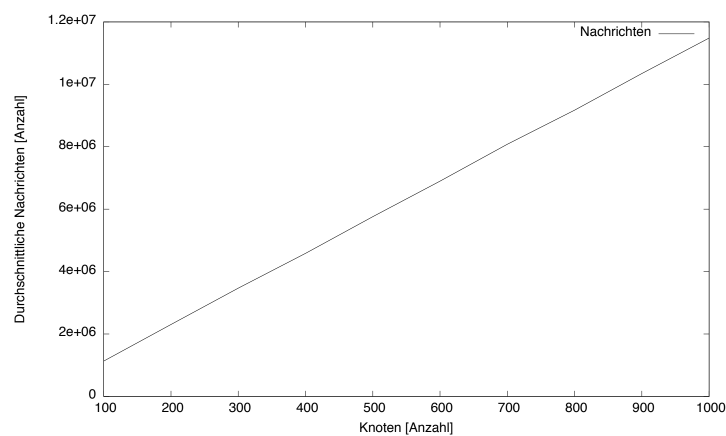


Abbildung 5.13: Durchschnittliche Anzahl aller Nachrichten einer Simulation

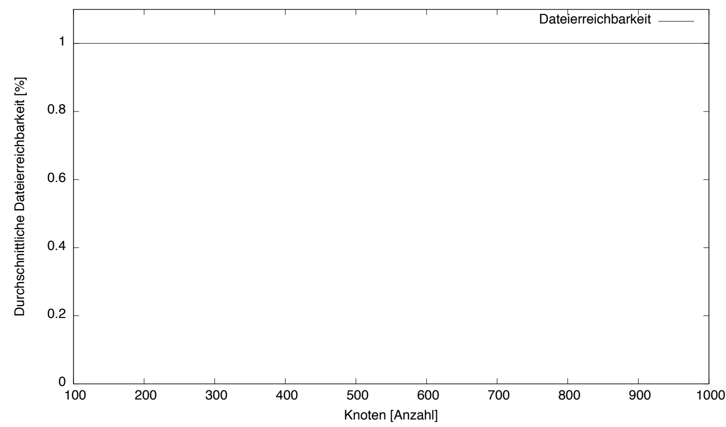


Abbildung 5.14: Durchschnittliche Dateierreichbarkeit von allen Dateianfragen einer Simulation

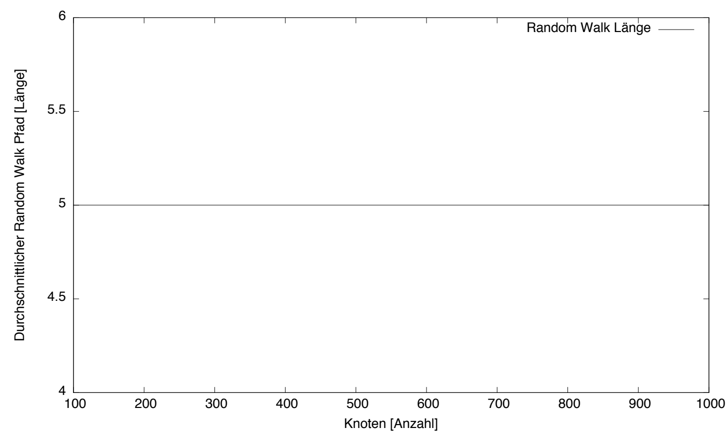


Abbildung 5.15: Durchschnittliche Länge eines Random Walk Pfades

F2-C2

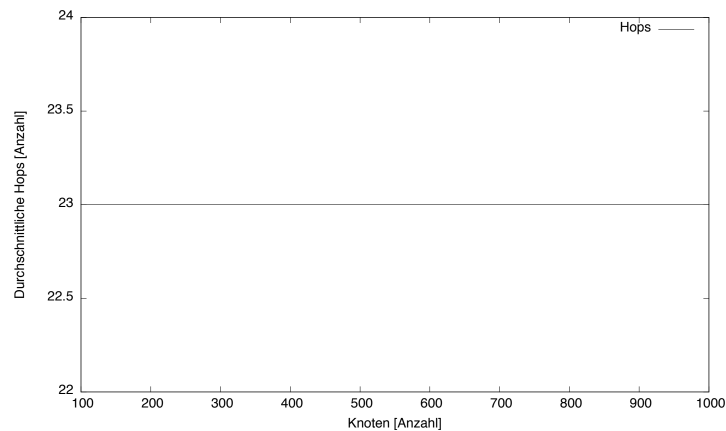


Abbildung 5.16: Durchschnittliche Hop Anzahl der `PeriodicDataOperation`

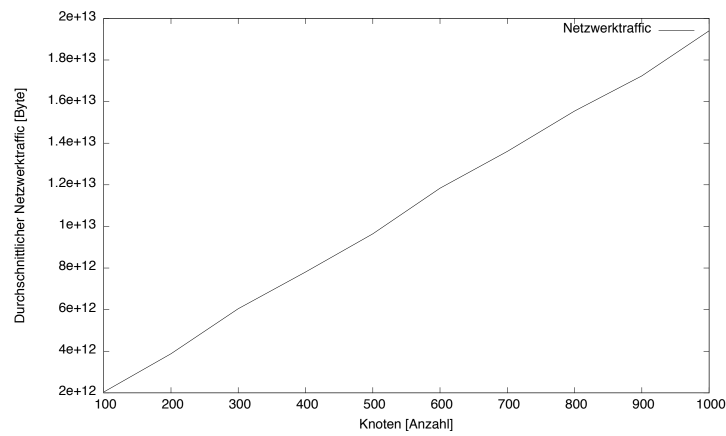


Abbildung 5.17: Durchschnittlicher Netzwerktraffic einer gesamten Simulation

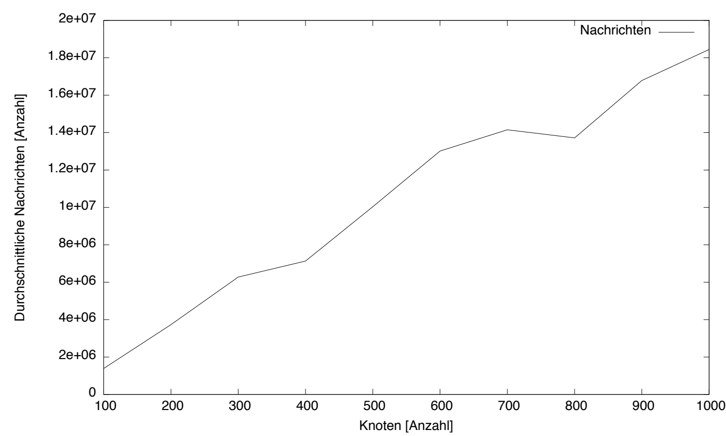


Abbildung 5.18: Durchschnittliche Anzahl aller Nachrichten einer Simulation

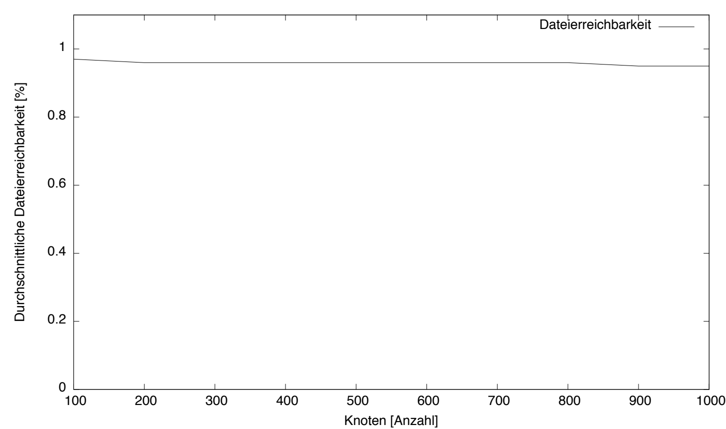


Abbildung 5.19: Durchschnittliche Dateierreichbarkeit von allen Dateianfragen einer Simulation

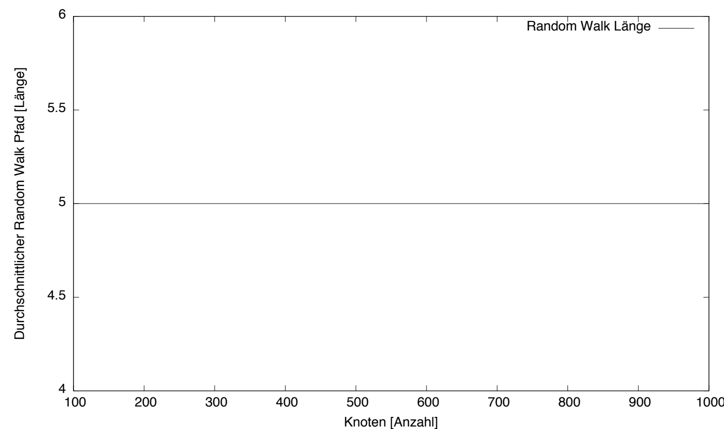


Abbildung 5.20: Durchschnittliche Länge eines Random Walk Pfades

5.4 Auswertungen

Die in Abschnitt 5.1 vorgestellten Metriken und die in Abschnitt 5.2 vorgestellten Setups werden anhand der Ergebnisse aus Abschnitt 5.3, die mithilfe des PeerfactSim.KOM Simulators [FG13] erzeugt wurden, erörtert, um die Performance und Funktionalität von ACE zu beurteilen.

5.4.1 Durchschnittliche Hop Anzahl

Wie anhand der Abbildungen 5.1, 5.6, 5.11 & 5.16 zu erkennen ist, beträgt die durchschnittliche Hop Anzahl, welche die `PeriodicDataOperation` benötigt, um eine Datei zu beziehen, in fast allen Fällen konstant **23 Hops**. Diese konstante Hop Anzahl liegt daran, dass bei allen Simulationen eine konstante Cloud Anzahl von 20 Clouds benutzt wurde. Bei der Verwendung von 100 Knoten kann es auch vorkommen, dass nach der zufälligen Gleichverteilung der 100 Knoten auf die 20 Clouds einige Clouds leer bleiben bzw. keine Knoten enthalten und somit weniger als 20 Clouds simuliert werden, weshalb die durchschnittliche Hop Anzahl von den **23 Hops** abweichen kann. Die **23 Hops** kommen wie folgt zustande:

- Wenn die `PeriodicLookupOperation` einen Lookup einer zufälligen Id erfolgreich ausgeführt hat, wird sie die `PeriodicDataOperation` aufrufen, die wiederum die Datei dieser zufälligen Id beziehen soll.
- Ein Lookup hat eine durchschnittliche Hop Anzahl von **9 Hops**. Dabei sind **5 Hops**, diejenigen Hops, die auf dem Random Walk Pfad in der Cloud des Initiators der Anfrage vorkommen, da der Initiator des Lookups verschleiert bleiben soll und der Lookup von einem anderen Knoten

in der Cloud ausgeführt werden soll. Weitere **4 Hops** kommen durch das Raussenden einer Anfrage auf den R-Ring zustande, da sich auf dem R-Ring 20 Rendezvous-Knoten befinden sollten, da bei jeder Simulation 20 Clouds simuliert werden ($\log(20) \approx 4,32$).

- Nachdem der Initiator eines Lookups weiß, in welcher Cloud sich der Beherberger der Datei zu einer gegebenen Id befindet, wird er die Datei wieder von einem anderen Knoten in der Cloud beziehen lassen, in dem er eine GetData Nachricht in seiner Cloud per Random Walk rumschicken lässt. Auf diesem Random Walk werden etwa **5 Hops** durchlaufen. Der Crossover-Peer wird die GetData Nachricht an die Cloud schicken, in welcher sich der Beherberger der Datei befindet. Dafür muss er zunächst den Rendezvous-Knoten dieser Cloud auf dem R-Ring suchen. Dies beansprucht ungefähr **4 Hops**, da der R-Ring aus 20 Rendezvous-Knoten bestehen sollte ($\log(20) \approx 4,32$). Nachdem der Crossover-Peer den Rendezvous-Knoten der Cloud kennt, wird er diesen direkt kontaktieren. Der Rendezvous-Knoten wird wiederum die GetData Nachricht in seiner Cloud fluten. Der Beherberger der Datei wird dann eine GetData Antwort, welche die angefragte Datei enthält, erstellen und sie per Random Walk durch die Cloud wandern lassen, bis schließlich ein Knoten aus der Cloud die GetData Antwort an den anfragenden Crossover-Peer sendet. Dieser Random Walk verläuft auch über ca. **5 Hops**.

- Insgesamt:

$$\underbrace{\underbrace{\textbf{9 Hops}}_{\text{Lookup}} + \underbrace{\textbf{5 Hops}}_{\text{Random Walk GetData Nachricht}} + \underbrace{\textbf{4 Hops}}_{\text{R-Ring}} + \underbrace{\textbf{5 Hops}}_{\text{Random Walk GetData Antwort}}}_{\textbf{23 Hops}}$$

5.4.2 Random Walk

Die Abbildungen 5.5, 5.10, 5.15 & 5.20 zeigen, dass sich das Random Walk Design von ACE, welches an das Random Walk Design von Crowds angelehnt ist, bewährt. Die durschnittliche Random Walk Pfadlänge beträgt 5, so wie es das Design vorsieht. Im Allgemeinen gilt in ACE $L := 5$. Ein Knoten wird eine Anfrage mit der Wahrscheinlichkeit $\frac{L-1}{L}$ an einen anderen Knoten in der Cloud und mit der Wahrscheinlickeit $\frac{1}{L}$ wird er die Anfrage raussenden. $L := 5$ soll bezwecken, dass die Random Walk Pfade ungefähr die Länge 5 haben sollen. Dies wird offensichtlich erfüllt.

5.4.3 Dateierreichbarkeit

Aus den Abbildungen 5.4 & 5.14 geht hervor, dass beim Simulieren ohne Churn alle Dateien, die beantragt werden, auch erfolgreich bezogen werden können. Das System hinter den Clouds scheint somit zu funktionieren. Knoten können sich in ihren Clouds verschleiern und dennoch erfolgreich Datei dem Netzwerk zur Verfügung stellen und beziehen. Unter Verwendung von Churn ist eine Dateierreichbarkeit von 100% nicht erreichbar, da zwischen der **50min** und **80min** Churn eingeschaltet ist und einige Knoten zufällig das Netzwerk verlassen, und die Dateien, welche diese dem Netzwerk zur Verfügung stellen, nicht mehr erreichbar sind. Insgesamt sind aber bei einem 30 minütigen Ausfall von Knoten noch etwa 95 Prozent aller Dateianfragen erfolgreich. Dazu muss jedoch erwähnt werden, dass ein Knoten in dem Churn Intervall zwar offline gehen kann, aber dennoch im Churn Intervall zurückkehren kann. D.h. ein Knoten muss nicht über das komplette Churn Intervall ausfallen. Ein weiterer Nachteil liegt darin, dass Dateien nicht nur erreichbar sind, wenn deren Beherbeger offline gehen. Es ist in ACE dennoch möglich, dass eine Datei per Random Walk durchgereicht wird in der Cloud des Beherbergers und ein Knoten, an den eine Datei weitergeleitet wird, offline geht. Somit kann die angefragte Datei auch nicht an den Initiator der Dateianfrage ausgehändigt werden. Um dieses Problem zu vermeiden, könnte man einen Knoten vorher anpingen, um zu sehen, ob dieser noch erreichbar ist. Um die Dateierreichbarkeit zu erhöhen, könnte ein Beherberger einer Datei einen zweiten zufälligen Beherberger dieser Datei per Random Walk analog zur dynamischen Dateilagerung in Abschnitt 3.3.2 bestimmen. Somit wäre eine Datei immernoch erreichbar, wenn der originale Beherberger offline geht und der zweite Beherberger der Datei noch online ist. Ein Nachteil liegt jedoch darin, dass wenn sowohl der originale als auch der zweite Beherberger der Datei online sind. Man müsste hier entweder extra Protokolle entwickeln, sodass nur einer dieser beiden die Dateianfrage benantwortet oder man lässt zu, dass beide eine Anfrage beantwortet können. Dieser Ansatz würde jedoch den erheblichen Netzwerktraffic, den ACE ohnehin schon erzeugt, drastisch vermehren.

5.4.4 Netwerktraffic

Im Allgemeinen lässt sich sagen, dass ein lineares Wachstum durch den Anstieg der Anzahl der Knoten verzeichnet wird (siehe Abbildungen 5.2, 5.7, 5.12 & 5.17). Das liegt daran, dass es periodische Operationen wie z.B die `PeriodicLookupOperation`, `PeriodicDataOperation`, `DistributeDataOperation` usw. gibt, die periodisch von allen Knoten in gleicher Anzahl und gleichen periodischen Abständen ausgeführt wird, wodurch eine proportionale Zuordnung vorliegt. Grundsätzlich lässt sich über ACE sagen, dass ein sehr hoher Netzwerktraffic erzeugt wird. Dies ist unter anderem den Random Walks zur Ausstellung von angefragten Dateien zu verschulden, die vom Beherberger einer Datei bis hin zum Initiator einer Anfrage einer Datei in etwa **10 Knoten** durchlaufen. In üblichen DHT's würde ein Anfrager einer Datei hingegen die Datei direkt vom Beherberger

einer Datei beziehen und somit viel weniger Traffic als in ACE verursachen. Abgesehen davon, gibt es einige Operationen, die ein Knoten sowohl auf dem Main-Ring als auch auf dem R-Ring periodisch ausführt. Die `UpdateCloudOperation` und `PingerOperation` verursachen auch jede Menge Traffic und tragen ihren Teil zu einem hohen Netzwerktraffic bei, welche periodisch ausgeführt werden, um die Cloud aufrechtzuerhalten und mit allen nötigen Informationen zu versorgen.

5.4.5 Nachrichten

Aus den Abbildungen 5.3, 5.8, 5.13 & 5.18 ist ebenfalls ein lineares Wachstum bei Erhöhung der Knotenanzahl zu erkennen. Je mehr Knoten im Netzwerk vorhanden sind, desto öfter werden diese in ihren periodischen Operationen Nachrichten erstellen und verschicken, weshalb auch hier eine proportionale Zuordnung wegen der periodischen Operationen vorliegt. Wie man leicht in Abschnitt 4.3 sehen kann, werden in ACE viele verschiedene Nachrichten verwendet, um die umfangreichen Funktionalitäten von ACE zu bewerkstelligen. Viele dieser Nachrichten werden auch periodisch versendet, weshalb ACE eine hohe durchschnittliche Anzahl an Nachrichten während einer Simulation verzeichnet.

5.4.6 Freundschaftsbeziehungen

Wie man den Ergebnissen entnehmen kann, stellen Freundschaftsbeziehungen bzgl. der ausgewählten Metriken keine signifikanten Nachteile gegenüber den Simulationen ohne Freundschaftsbeziehungen dar. In jeder Simulation hat ein Knoten etwa 5 Freunde in seiner Cloud, die zufällig ausgewählt werden. Anfragen oder Antworten werden beim Simulieren von Freundschaftsbeziehungen nur über zufällig ausgewählte Freunde mittels Random Walk durchgereicht. Ein Knoten wird eine Anfrage oder Antwort einer Dateianfrage nicht per Random Walk rausschicken, wenn alle seine Freunde offline sind. Etwa 5 Freunde innerhalb einer Cloud zu besitzen, scheint auszureichen, um Anfragen und Antworten von Dateianfragen erfolgreich zu verschicken, sodass die Dateierreichbarkeit trotz Freundschaftsbeziehungen mit und ohne Churn solide Werte liefern kann (siehe Abbildungen 5.14 & 5.19).

Kapitel 6

Ausblick

Das in dieser Thesis vorgestellte Overlay ACE, welches eine Modifikation von Agyaat darstellt, und mit Clouds arbeitet, die unstrukturierte Netzwerke darstellen und an DHT basierte P2P-Overlays drangehängt werden können, benutzt viele in dieser Arbeit vorgestellten Anonymisierungsverfahren, welche auch in bekannten und bewährten Systemen benutzt werden. Clouds verschleiern den Sender und Empfänger einer Anfrage um *Sender*- bzw. *Empfängeranonymität* zu gewährleisten. Die *Empfängeranonymität* in ACE wird durch das Benutzen von anonymen DHT's bewerkstelligt, in dem Keys nicht wie gewohnt auf Knoten abgebildet werden, sondern auf Clouds. Durch einen Lookup eines Keys erhält man also nicht wie es in DHT's im Allgemeinen üblich ist, den Beherberger der Datei des angefragten Keys, sondern nur die Cloud, in der sich der Beherberger der Datei befindet. Random Walks werden innerhalb der Clouds genutzt, um Anfragen und Antworten von Dateianfragen zu verschleiern, bzw. anderen Knoten nicht zu offenbaren, dass die Anfrage oder Antwort von einem bestimmten Knoten initialsiert wurde. Die durch die Random Walks enstehenden Pfade, werden genutzt, um Antworten von jeglichen Anfragen mittels rekursivem Routing zurück an den Initiator einer Anfrage zu leiten. Das rekursive Routing verschleiert, wie bei unstrukturierten Netzwerken, wie z.B Gnutella, den Sender und Empfänger einer Anfrage. Die *Anonymität duch Vertrauen* kann in ACE bis zu einem gewissen Grad geschützt werden, indem man Freundschaftsbeziehungen innerhalb der Clouds simuliert. Jedoch ist es unverzichtbar, dass alle Knoten mit dem Rendezvous-Knoten und dem Pinger vernetzt sein müssen, auch wenn diese keine Freunde sind, damit das Prinzip hinter den Clouds funktionieren kann, weshalb die *Anonymität duch Vertrauen* in ACE auch nicht komplett gewährleistet werden kann, welche das Ziel hat, dass innerhalb einer Cloud Knoten nur von ihren Freunden gesehen werden und auch nur mit diesen kommunizieren. Bei der Wahl von Rendezvous-Knoten sollte man, wenn möglich Super-Peers [YGM03] wählen, da ein Rendezvous-Knoten viele Aufgaben bewerkstelligen muss und somit eine hohe Performance erfordert. Rendezvous-Knoten mit weniger Performance könnten leicht überfordert sein und das System damit verlangsamen. ACE ist skalierbar, da der R-Ring, über welchen alle Dateien angefragt und verbreitet werden, eine DHT ist (der R-Ring ist ein Chord Ring bestehend aus Rendezvous-Knoten) und Anfragen von Dateien neben dem Routen auf dem R-Ring in $O(\log(N))$ lediglich ein paar Hops in der Cloud des Initiators einer Anfrage und ein paar Hops in der Cloud des

Beherbergers durchlaufen. Die durchschnittliche Hop Anzahl, um eine Datei zu beziehen, hängt in Chord von der Anzahl der teilnehmenden Knoten ab, wohingegen bei ACE die durchschnittliche Hop Anzahl nicht an der Anzahl aller Knoten im Netzwerk abhängt, sondern an der Anzahl der Clouds bzw. der Anzahl Rendezvous-Knoten auf dem R-Ring.

6.1 Zukünftige Arbeiten

ACE ist ein Overlay, welches unter dem Aspekt Anonymität in Form von *Sender*- und *Empfängeranonymität* konzipiert wurde. Es wäre für zukünftige Arbeiten intressant, ACE auf Anonymität zu überprüfen, indem man Angriffsmechanismen von böswilligen Knoten simuliert und erörtert, um zu sehen, was das System trotzt des Designs, welches auf Anonymität abgezielt ist, an Informationen durchsickern lässt, die ein böswilliger Knoten bzw. Angreifer herausbekommen kann, um Sender und Empfänger von Anfragen zu identifizieren. Zusätzlich könnte man die dynamische Lagerung in ACE einbinden und auch erörtern, inwiefern das zur Verbesserung der Empfängeranonymität dient oder auch nicht. Man könnte ebenfalls überprüfen, inwiefern die Datei Duplizierung bzw. das Lagern von einer Datei von dem originalen und einem zweiten Beherberger zur Dateierreichbarkeit im Falle von Churn beiträgt. Außerdem könnte man die Clouds, welche in ACE als Anonymisierungsverfahren verwendet werden, in andere DHT basierte P2P-Overlays wie z.B. Pastry oder Kademlia einbinden und Vergleiche zu ACE ziehen, welches auf Chord beruht.

Literaturverzeichnis

[ACPP12] ARNABOLDI, Valerio; CONTI, Marco; PASSARELLA, Andrea; PEZZONI, Fabio: Analysis of ego network structure in online social networks. In: *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)* IEEE, 2012, S. 31–40.

[Bit] BITTORRENT, Inc: *BitTorrent*. Website. `http://www.bittorrent.com`.

[Bor05] BORISOV, Nikita: *Anonymous routing in structured peer-to-peer overlays*, University of California, Berkeley, Diss., 2005

[Bri00] BRICKLIN, David: *Friend-to-Friend Networks Friend-to-Friend Networks*. Website. `http://www.bricklin.com/f2f.htm`. Version: 2000.

[BW05] BORISOV, Nikita; WADDLE, Jason: *Anonymity in structured peer-to-peer networks*. Computer Science Division, University of California, 2005

[Cha81] CHAUM, David L.: Untraceable electronic mail, return addresses, and digital pseudonyms. In: *Communications of the ACM* 24 (1981), Nr. 2, S. 84–90.

[Cov91] COVER, Thomas M.: *JA Thomas Elements of information theory*. 1991.

[CSWH01] CLARKE, Ian; SANDBERG, Oskar; WILEY, Brandon; HONG, Theodore W.: Freenet: A distributed anonymous information storage and retrieval system. In: *Designing Privacy Enhancing Technologies* Springer, 2001, S. 46–66.

[DMS04] DINGLEDINE, Roger; MATHEWSON, Nick; SYVERSON, Paul: Tor: The second-generation onion router / DTIC Document. 2004. Forschungsbericht.

[DSCP03] DIAZ, Claudia; SEYS, Stefaan; CLAESSENS, Joris; PRENEEL, Bart: Towards measuring anonymity. In: *Privacy Enhancing Technologies* Springer, 2003, S. 54–68.

[DV04] DONNELL, Charles W.; VAIKUNTANATHAN, Vinod: Information leak in the Chord loo-

kup protocol. In: *Peer-to-Peer Computing, 2004. Proceedings. Proceedings. Fourth International Conference on* IEEE, 2004, S. 28–35.

[FG13] FELDOTTO, Matthias; GRAFFI, Kalman: Comparative Evaluation of Peer-to-Peer Systems Using PeerfactSim.KOM. In: *In Proc. of IEEE HPCS'13*, 2013.

[HW] HAZEL, Steven; WILEY, Brandon: Achord: A Variant of the Chord Lookup Service for Use in Censorship Resistant Peer- to- Peer Publishing Systems.

[MM02] MAYMOUNKOV, Petar; MAZIERES, David: Kademlia: A peer-to-peer information system based on the xor metric. In: *Peer-to-Peer Systems*. Springer, 2002, S. 53–65.

[PK01] PFITZMANN, Andreas; KÖHNTOPP, Marit: Anonymity, Unobservability, and Pseudonymity–A Proposal for Terminology. (2001).

[RD01] ROWSTRON, Antony; DRUSCHEL, Peter: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: *Middleware 2001* Springer, 2001, S. 329–350.

[RR98] REITER, Michael K.; RUBIN, Aviel D.: Crowds: Anonymity for web transactions. In: *ACM Transactions on Information and System Security (TISSEC)* 1 (1998), Nr. 1, S. 66–92.

[SGL06] SINGH, Aameek; GEDIK, Bugra; LIU, Ling: Agyaat: mutual anonymity over structured P2P networks. In: *Internet Research* 16 (2006), Nr. 2, S. 189–212.

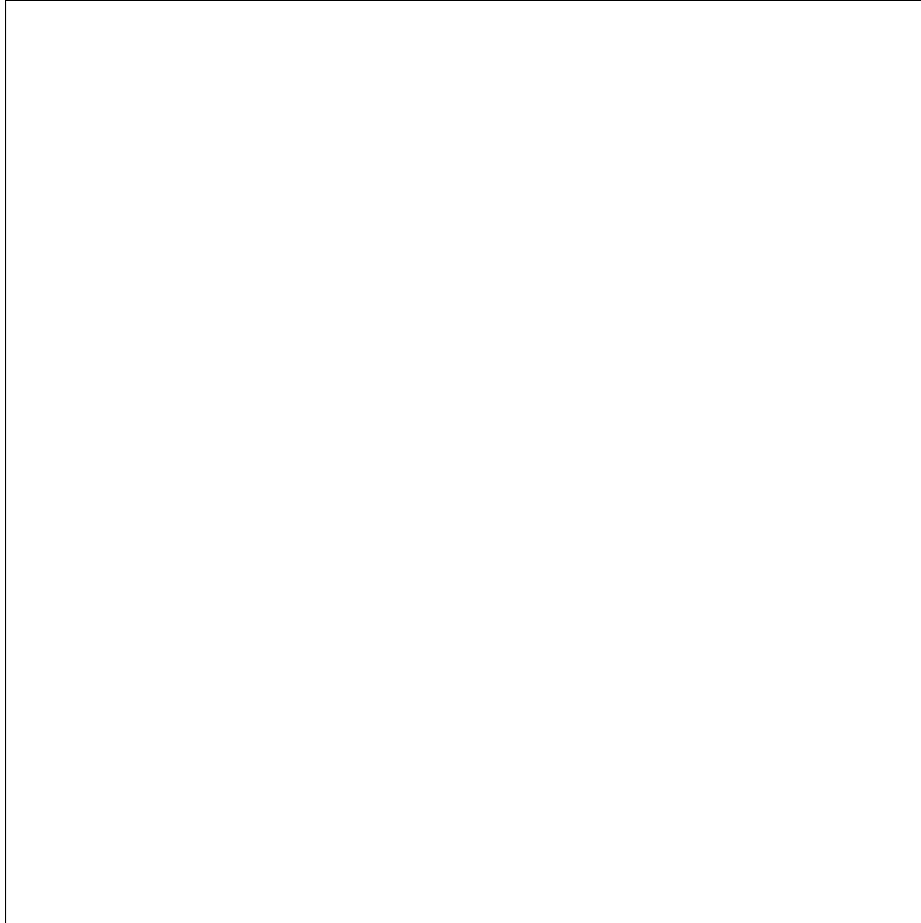
[SMK+01] STOICA, Ion; MORRIS, Robert; KARGER, David; KAASHOEK, M F.; BALAKRISHNAN, Hari: Chord: A scalable peer-to-peer lookup service for internet applications. In: *ACM SIGCOMM Computer Communication Review* 31 (2001), Nr. 4, S. 149–160.

[YGM03] YANG, Beverly; GARCIA-MOLINA, Hector: Designing a super-peer network. In: *Data Engineering, 2003. Proceedings. 19th International Conference on* IEEE, 2003, S. 49–60.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt 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.

Düsseldorf, 22.Juli 2015 Erol Yildirim



Diese CD beinhaltet:

- eine PDF-Version der Bachelorarbeit
- alle LaTeX-Dateien und eingebundene Grafiken
- den Quellcode der während dieser Arbeit entstandenen Software
- die Messdaten der durchgeführten Evaluationen
- die in dieser Arbeit referenzierten Paper