
# Implementierung und Evaluierung eines Freenet basierten Protokols zu Anonymisierungszwecken in Peer-to-Peer Netzwerken

Bachelorarbeit
von

Timm Kenfenheuer

geboren in
Bergisch Gladbach

eingereicht bei

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

August 2015

Betreuer:
Tobias Amft

# Abstract

Das Internet ist anonym und jeder kann schreiben, was er möchte. Leider trifft dies in einigen Ländern der Welt nicht zu. In diesen Ländern werden Personen, die, in den Augen des Staats, falsche Meinungen kund tun, verfolgt und bestraft. Dort gibt es keine Meinungs- oder Pressefreiheit und man kann per IP-Adresse ausfindig gemacht werden. Für diese Menschen wäre es von Vorteil, wenn es ein Programm gäbe, mit dem sie wirklich anonym ihre Meinung verbreiten können, ohne sich vor Zensur oder Verfolgung fürchten zu müssen. In dieser Arbeit wird ein FreeNet ähnliches Netzwerk implementiert und analysiert. FreeNet ist ein Netzwerk, bei dem genau diese Anonymität im Vordergrund steht. Es wird gezeigt, wie das generelle Design aussieht und wie es an bestimmten Stellen verbessert wird. Anhand von Simulationen ist gut zu erkennen, dass das Netzwerk ohne Churn nahezu Fehlerfrei funktioniert. Das Design weist Schwachstellen auf, da unter Churn nicht immer alle abgelegten Dateien gefunden werden, aber es wird gezeigt, dass diese Implementierung Absenderanonymität gewährleistet und das sollte das Ziel dieser Arbeit sein. Dass nicht immer alle Dateien gefunden werden liegt aber auch daran, dass Teilnehmer des Netzwerks (Knoten), die Daten speichern, offline gehen und deshalb die Daten nicht mehr verfügbar sind, bis der Knoten wieder online ist. Das Verhalten mit Churn wird in Simulationen getestet und daraufhin analysiert. Die Analysen ergeben, dass die Ergebnisse dieser Simulationen große Unterschiede zu den Ergebnissen der Simulationen ohne Churn aufweisen.

# Danksagung

An dieser Stelle möchte ich mich bei meinem Betreuer, Tobias Amft, bedanken, der mich während der Bearbeitung dieser Bachelorarbeit stets motiviert und unterstützt hat. Vor allem in den Treffen, aber auch in den e-mails, gab er mir immer gute Anregungen und Hinweise zu meinen Ideen. Außerdem möchte ich ihm für das Korrekturlesen dieser Bachelorarbeit danken.
Vielen Dank, dass sie mir ermöglicht haben diese Arbeit bei Ihnen zu schreiben.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung

Ein immer wieder aktuelles Thema ist die Meinungsfreiheit in bestimmten Ländern der Welt. Wenn dort eine Person eine andere Meinung zu einem Thema hat, soll er diese für sich behalten oder besser noch, seine Meinung ändern. Kommt es vor, dass jemand diese andere Meinung äußert, so muss er damit rechnen, dass der Staat ihn dafür belangen wird. Teilweise sogar mit harten Strafen. Deshalb trauen sich viele Menschen in diesen Ländern nicht, ihre eigene Meinung zu sagen, da selbst im "unabhängigen" Internet diese Personen anhand ihrer IP-Adresse ausfindig gemacht werden können. In dieser Arbeit werden wir ein FreeNet ähnliches Netzwerk kennenlernen, welches anonymes Teilen von Daten ermöglicht. Kann der Urheber einer Nachricht nicht ermittelt werden, so kann auch niemand für den Inhalt bestraft werden. Wir werden sehen, dass dieses Netzwerk, unterschiedlich gut, im Bereich Datenhaltung, funktioniert, je nachdem, ob gewisse Eigenschaften erfüllt sind. Das Netzwerk wird als Peer-to-Peer-Netzwerk angelegt und entzieht sich somit, als dezentrales System, auch der Kontrolle jeglicher Organisation. Das hier betrachtete Netzwerk ist nicht FreeNet sondern nur ein auf FreeNet basierendes Netzwerk.

## 1.1 Gliederung der Arbeit

In Kapitel 1 gab es eine kurze Einleitung, warum ein solches Netzwerk von Interesse ist und welche Funktionen ein Netzwerk beinhalten sollte. In Kapitel 2 werden wir Arbeiten kennen lernen, die wichtig für das Verständnis dieser Arbeit sind. Dort werden auch Peer-to-Peer-Netzwerke und FreeNet genauer erklärt. Außerdem wird auf den Simulator eingangen, den wir verwenden werden. Kapitel 3 beschreibt das generelle Design und zeigt wie zwei Probleme,welche den Join-Prozess und Churn betreffen, gelöst wurden, die während der Implementierung aufgetreten sind. Darauf folgt Kapitel 4, welches sich mit eben dieser Implementierung beschäftigt und die wichtigsten Klassen und Methoden vorstellt. Danach folgt Kapitel 5, in dem das Netzwerk getestet wird. Hier wird es zwei Hauptthemen

geben. Eines wird das Verhalten des Netzwerks in einer idealen Umgebung sein und das andere das Verhalten des Netzwerks in einer weniger idealen Umgebung unter Churn. Was Churn genau ist wird noch erklärt werden. Auf dieses Kapitel folgt ein Fazit (Kapitel 6), das diese Arbeit noch einmal zusammenfasst und das Netzwerk bewertet. Abschließend werden in Kapitel 7 Themen für zukünftige Arbeiten aufgezeigt, welche wir bei der Implementierung und Simulation erkennen werden.

# Kapitel 2

# Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten erläutert, die in dieser Arbeit von Bedeutung sind. Dazu gehört eine grobe Erklärung, was Peer-to-Peer-Netzwerke sind und welche Vorteile sie haben. Des weiteren wird erklärt, was FreeNet für ein Netzwerk ist und wie es funktioniert. Als letztes wird der Simulator erklärt, der für diese Arbeit verwendet wurde.

## 2.1 Peer-to-Peer-Netzwerke

Peer-to-Peer-Netzwerke sind dezentrale Netzwerke. Das bedeutet sie benötigen keine zentrale Instanz um funktionieren zu können. Im Gegensatz zu einer dezentralen Peer-to-Peer-Architektur, steht die zentrale Server-Client-Architektur. Hierbei ist der Server eine zentrale Instanz, auf die alle Nutzer (Clients) zugreifen können um zum Beispiel eine Website herunter zu laden. In einem dezentralen System gibt es keinen Server. Vielmehr sind alle Daten auf das gesamte Netzwerk aufgeteilt. Das bedeutet jeder Benutzer (Knoten) speichert einen Teil der Daten und niemand ist auf einen zentralen Anbieter der Daten angewiesen. Bei Peer-to-Peer Netzwerken spricht man von so genannten Overlays, da sie über die eigentliche Netzwerkarchitektur gelegt werden. Diese Overlays können strukturiert oder unstrukturiert sein. Bei einem strukturierten Overlay werden neue Knoten an bestimmten Stellen im Overlay eingefügt, welche durch das Overlay bestimmt wird. Ein bekanntes strukturiertes Overlay ist Chord [SMK+01]. In Chord werden die Knoten als Ring angeordnet, wobei jeder Knoten einen Vorgänger und einen Nachfolger hat. Die Position, die ein Knoten im Ring einnimmt wird über einen so genannten Identifier ermittelt. Der Vorgänger hat einen kleineren Identifier, der Nachfolger einen größeren. Zusätzlich wird der Knoten mit dem größten Identifier mit dem Knoten mit dem kleinsten Identifier verbunden. Weitere bekannte strukturierte Overlays sind z.B. Pastry [RD01] oder Kademlia [MM02]. Im Gegensatz dazu stehen unstrukturierte Overlays, bei denen Knoten dem Netzwerk beitreten und keinen festgelegten Platz haben, sondern sich ohne Strukturierung mit anderen Knoten verbinden. Hierbei ist das bekannteste Overlay, das Gnutella [Cli02] Overlay. Hier können Nach-

richten nur durch Fluten, also das Weitersenden von Nachrichten/Requests an alle Kontakte, erreicht werden. Dabei wird jeder Knoten erreicht, aber es werden auch sehr viele überflüssige Nachrichten erzeugt. Das in dieser Arbeit implementierte Netzwerk zählt zu den strukturierten Netzwerken und benutzt dazu, wie Chord, Identifier.

## 2.2 FreeNet

FreeNet [Cla99] ist ein Peer-to-Peer-Netzwerk, welches anonymen Datenaustausch ermöglichen soll. Dazu setzt FreeNet darauf, dass der Absender von Nachrichten nicht ermittelt werden kann. Jeder Knoten entscheidet für sich an welchen Knoten er Daten weiterleitet. Dadurch weiß ein Knoten nur von wem er die Nachricht bekommen hat, nicht aber ob dieser die Nachricht nur weitergeleitet hatte. Jeder Knoten besitzt einen Identifier, der ihn im Netzwerk eindeutig identifiziert. Dieser Identifier wird verwendet um Nachrichten an sein Ziel zu bringen. Um dies zu ermöglichen verwendet FreeNet 4 grundlegende Nachrichten.

### 2.2.1 Request-Message

Ein Request wird von einem Knoten ausgeführt, wenn dieser Daten von einem anderen Knoten erhalten möchte. Dazu wird eine Request-Nachricht erzeugt, welche dann an einen Knoten geschickt wird, der vorher mit Hilfe der Identifier bestimmt wurde. In einem Request steht ein Identifier, der aus den gesuchten Daten generiert wird. Wie dies genau funktioniert spielt in dieser Arbeit keine Rolle, kann aber im FreeNet White Paper [Cla99] nachgelesen werden. Anhand von diesem Identifier wird die Nachricht nun durch das Netzwerk geleitet, was Routing genannt wird. Dadurch wird der Request zu einem Knoten geroutet, dessen Identifier ähnlich zu dem des Requests ist.

### 2.2.2 Insert-Message

Ein Insert wird gebraucht um Daten im Netzwerk ablegen zu können. Es funktioniert wie der Request nur das bei dieser Nachricht Daten mitgesendet werden und am Zielknoten abgelegt werden. Aus diesen Daten wird der Identifier für die Nachricht erstellt.

### 2.2.3 Reply-Message

Eine Reply-Message ist die Antwort auf eine Request-Message. Sie liefert die angefragten Daten zurück an den Ausgangsknoten. Anders als beim Request wird hierbei kein Identifier für das Routing gebraucht. Die Nachricht wird über die selben Knoten zurückgeschickt, über die der Request kam. Dadurch bleibt der Knoten, der den Request losgeschickt hat immer noch anonym.

### 2.2.4 Failed-Message

Die Failed-Message dient dazu, dem Knoten, der den Request erzeugt hat, mitzuteilen, dass der Zielknoten die Daten nicht hat. Entweder wurden die Daten überschrieben, der Knoten, der sie hat ist dann offline, oder sie wurden nie abgelegt.

Mit Hilfe der Art und Weise wie diese Nachrichten benutzt werden, bleibt der Ausgangsknoten anonym, da niemand weiß ob der Knoten, von dem man eine Nachricht erhalten hat, Ausgangspunkt der Nachricht war oder ob dieser die Nachricht nur weitergeleitet hat.

## 2.3 Simulator – PeerfactSim.KOM

PeerfactSim.KOM [Gra11] ist ein Simulator, mit dem Peer-to-Peer-Netzwerke simuliert werden können. Dazu ist ein Simulator praktisch, da ein Peer-to-Peer-Netzwerk erst ab vielen Knoten richtig funktioniert. Da aber nicht immer viele Knoten in Hardware umgesetzt werden können verwenden wir einen Simulator. PeerfactSim.KOM ist von der TU Darmstadt entwickelt und danach von der Universität Paderborn und der Heinrich-Heine-Universität Düsseldorf weiterentwickelt worden. Der Simulator ist unter der Gegneral Public License als Quellcode in Java erhältlich. Die Simulationen werden in Java implementiert. Der Simulator simuliert jedoch nicht nur das reine Overlay der Peer-to-Peer-Netzwerke, sondern auch die Netzwerkschichten darunter. So werden zum Beispiel auch die Netzwerkschicht und die Transportschicht simuliert und können ausgewertet werden.

# Kapitel 3

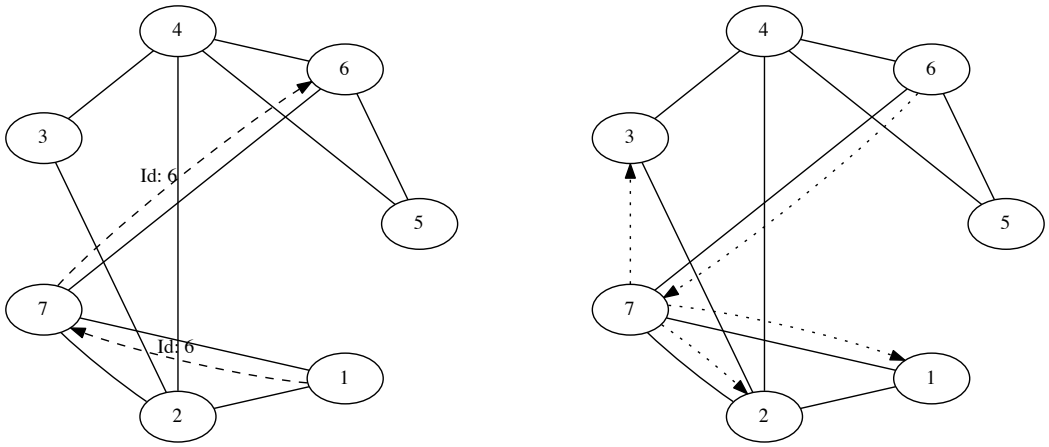
# Design

In diesem Kapitel wird das grundlegende Design von FreeNet, welches als Basis dieser Arbeit verwendet wurde, erläutert. Außerdem wird auf zwei Probleme eingegangen und gezeigt, wie diese behoben wurden. Diese wurden im FreeNet White Paper nicht betrachtet und somit musste ein Weg gefunden werden diese Operationen durchzuführen. Das erste Problem beschäftigt sich mit dem Join-Prozess. Dieser war bei der Implementierung des Overlays im Simulator zunächst fehlgeschlagen. Das zweite Problem tritt nur unter Churn auf, da nur dort Knoten das Netzwerk verlassen.

## 3.1 Generelles Design

Das in dieser Arbeit verwendete Design basiert auf dem FreeNet White Paper [Cla99]. Aus diesem Paper ist der Mechanismus Anonymität zu erzeugen von besonderem Interesse. Dieser Algorithmus basiert darauf, dass ein Knoten, der eine Nachricht empfängt, nicht sagen kann, ob der Knoten, von dem die Nachricht kommt die Anfrage erzeugt hat, oder ob dieser die Nachricht nur weitergeleitet hat. Jeder Knoten, der eine Nachricht erhält, überprüft, ob die Nachricht für ihn ist. Sollte die Nachricht nicht für ihn sein, so leitet er sie entsprechend Richtung Ziel weiter. Jeder Knoten kennt jedoch nur seine eigenen Nachbarn. Es gilt also zu klären, wie eine Nachricht in die richtige Richtung weitergeleitet werden kann. Dafür besitzt jeder Knoten einen Identifier. Dieser Identifier ist dem Knoten eindeutig zugeordnet, was bedeutet, dass dieser Identifier nur einmal im gesamten Netz vorkommt. Anhand dieses Identifiers tritt ein Knoten dem Netzwerk bei (Join) und besitzt eine eindeutige Position in diesem Netzwerk. Wenn nun eine Datei im Netzwerk abgelegt werden soll, so wird aus dieser ein Identifier bestimmt. Dieser Identifier wird mit in die Nachricht geschrieben, die die Datei am entsprechenden Knoten ablegen soll. Nun wird derjenige Nachbarknoten bestimmt, dessen Identifier am nächsten am Identifier der Nachricht liegt. An diesen Knoten wird die Nachricht nun geschickt. Dieser Knoten überprüft seine Kontakte und überprüft ob er selber das Ziel ist. Hat er einen Kontakt gefunden dessen Identifier näher an der der Nachricht liegt, so sendet er diesem Knoten die Nachricht. Diese geschieht

so lange, bis ein Knoten sich selbst als nächsten Knoten feststellt. Dies ist in Abbildung 3.1 (a) zu sehen. In diesem Beispiel ist die Knotennummer der Identifier des jeweiligen Knotens. Dort sendet Knoten 1 eine Anfrage nach einer Datei mit Identifier 6. Knoten 1 hat Verbindungen zu Knoten 2 und Knoten 7. Da Knoten 7 näher an 6 liegt als Knoten 2 wird die Nachricht an Knoten 7 geschickt. Knoten 7 hat Verbindungen zu Knoten 1 und Knoten 6. Knoten 7 stellt fest, dass 6 am nächsten zu Identifier 6 ist und schickt die Nachricht weiter zu Knoten 6. Knoten 6 weiß nun aber nur, dass Kno-




(a) Knoten 1 sendet einen Request

(b) Wer ist der Absender der Nachricht?

Abbildung 3.1: Routing von Nachrichten

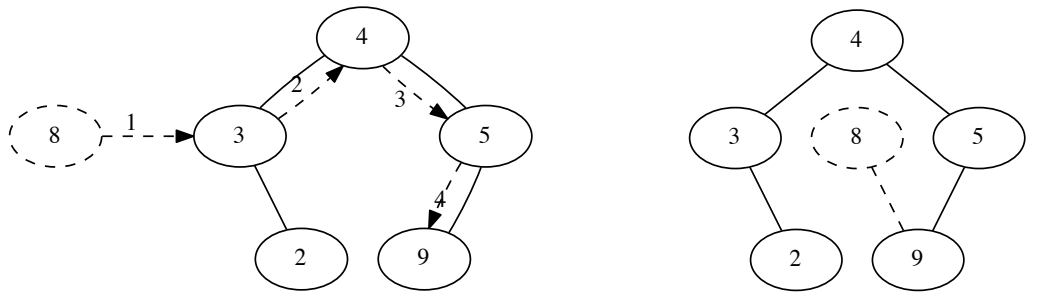
ten 7 ihm eine Nachricht geschickt hat. Wer aber die Daten genau haben will ist nicht zu ermitteln. Hierzu betrachten wir Abbildung 3.1 (b). Dort ist zu sehen, dass Knoten 6 nicht weiß, wer die Nachricht ursprünglich losgeschickt hat, denn Knoten 6 kann nicht feststellen zu welchen Knoten Knoten 7 Verbindungen hat. Deshalb kann Knoten 6 nur feststellen, dass die Nachricht nicht von Knoten 5 oder 4 stammen kann, da diese ihm die Nachricht auf direktem Weg geschickt hätten. In diesem kleinen Beispiel ist die Anonymität noch nicht besonders gut, allerdings konnten nur die direkten Nachbarn ausgeschlossen werden. Somit führt dies in einem größeren Netzwerk zu einer höheren Anonymität. Auch mehr Knoten auf dem Weg zum Zielknoten erhöhen die Anonymität, da mehr Knoten beteiligt sind, die ihre Kontakte gegenseitig nicht kennen.
Daten werden durch eine DataInsert-Nachricht abgelegt. Dazu wird aus den Daten ein Identifier gebildet und die Daten zusammen mit diesem Identifier in eine solche Nachricht gespeichert. Anhand des Identifiers gelangt die DataInsert-Nachricht zu einem Knoten, der für diesen Identifier verantwortlich ist. An diesem Knoten werden die Daten gespeichert. Will ein Knoten eben diese Daten abrufen, so erzeugt er eine DataRequest-Nachricht mit dem Identifier der Daten und einer NachichtenID. Anhand des Identifiers wird die Nachricht zu dem Knoten geroutet, an dem diese vorher abgelegt wurden. Unterwegs zu diesem Knoten hat jeder Knoten, der die Nachricht weitergeleitet hat, gespeichert von

wem er die Nachricht erhalten hat. Der Knoten, der die Daten gespeichert hat sendet die DataReply-Nachricht an den Knoten, von dem er vorher die DataRequest Nachricht erhalten hat. Die NachrichtenID der DataRequest-Nachricht und der DataReply-Nachricht stimmen überein, sodass jeder Knoten die DataReply-Nachricht als Antwort auf die DataRequest-Nachricht erkennt. Jeder Knoten sendet die Nachricht an den Knoten zurück, von dem er vorher die DataRequest-Nachricht erhalten hat. Dadurch gelangt die DataReply-Nachricht, mit den Daten, zurück zum Erzeuger der DataRequest-Nachricht. Zusätzlich speichert jeder Knoten auf dem Rückweg die Daten, sodass nicht nur der Erzeuger der DataRequest-Nachricht die Daten hat, sondern auch alle Knoten, die diese Nachricht weitergeleitet haben. Den Vorteil hiervon kann man im Beispiel sehen. Würde nach der Anfrage von Knoten 1 Knoten 2 dieselbe Nachricht mit Id: 6 losschicken, so sendet schon Knoten 7 die Antwort auf die Nachricht.

## 3.2 Erstes Problem – Join

Das FreeNet White Paper beschreibt, wie das Routing in FreeNet funktionieren soll, jedoch nicht, wie das Netzwerk aufgebaut wird. Hier ist man von einem bereits existierenden Netzwerk ausgegangen. Für die Simulation von FreeNet muss es aber eine Join Operation geben, die das Netzwerk im Sinne von FreeNet aufbaut. In diesem Teil wird Churn nicht betrachtet.

### 3.2.1 Erster Ansatz




(a) Knoten 8 mëchte dem Netzwerk beitreten

(b) Knoten 8 ist dem Netzwerk beigetreten

Abbildung 3.2: Join – Erster Ansatz für den Join-Prozess

Dabei ist der erste Ansatz, dass Knoten nach dem selben Prinzip hinzugefügt werden, wie Nachrich-

ten durch das Netzwerk geschickt werden. Was bedeutet man sucht einen Knoten, mit dem man sich verbinden kann. Dieser Knoten ist dann der nächste zu sich selbst (bzgl. des Identifiers). In Abbildung 3.2 (a) sieht man, wie dieser Join durchgeführt wird. Dabei versucht ein Knoten mit Identifier 8 dem Netzwerk beizutreten. Knoten 8 kennt Knoten 3 und somit sendet er diesem bei 1 einen Join-Request. Knoten 3 stellt fest, dass er einen Knoten kennt welcher näher an 8 ist als er selbst, der Knoten 4 ist. Dieser stellt ebenfalls fest, dass er einen Knoten kennt der ebenfalls näher an 8 ist als 4, nämlich Knoten 5. Knoten 5 bestimmt Knoten 9 als näheren. Dadurch wurde nun der Knoten gefunden, welcher am nächsten an Knoten 8 liegt. Also joint Knoten 8 Knoten 9. Das Ergebnis des Joins sieht man in Abbildung 3.2 (b).
Zur Bestimmung, welcher Knoten näher am gesuchten Identifier ist, wird von den Identifiern der Knoten, der gesuchte Identifier subtrahiert und die Beträge der Ergebnisse bestimmt. Nun wird über diese Beträge das Minimum gebildet (3.1) . Dadurch erhält man den Knoten, der betragsäßig am nächsten ist.

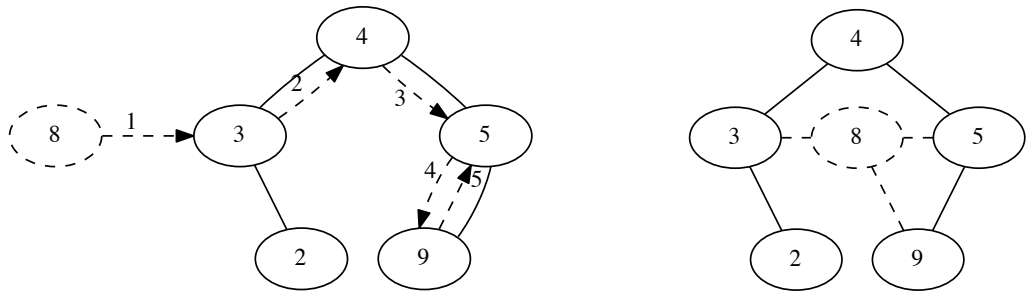
$$\min_i |Id_i - Id_{gesucht}| \tag{3.1}$$

Doch wo liegt nun das Problem?
Das Problem zeigt sich bei genauerer Betrachtung von 3.2 (b). Denn was passiert, wenn Knoten 8 nach Identifier 4 sucht?
Knoten 8 wird feststellen, dass Knoten 9 weiter von Knoten 4 weg ist. Dadurch ermittelt Knoten 8 sich selbst als Ziel der Suche. Jedoch gibt es einen Knoten mit Identifier 4 und dieser sollte auch erreicht werden. Wenn der Join schon bei diesem kleinen Beispiel falsch läuft, wird es bei einem sehr viel größeren Netzwerk zu gravierenden Fehlern führen, sodass kaum noch eine Nachricht am richtigen Ziel ankommt. Daher muss die Join-Operation verbessert werden, was im Folgenden erläutert wird.

### 3.2.2 Verbesserung

Erst einmal müssen wir feststellen wo der Fehler im Join liegt. Ist die Verbindung zwischen Knoten 8 und Knoten 9 falsch? Nein, denn nach dem Prinzip von FreeNet ist dies die richtige Verbindung, da 9 der nächste Knoten zu 8 ist.
Das Problem liegt darin, dass Knoten 8 sich nur mit Knoten 9 verbindet. Dadurch ermittelt Knoten 8 für alle Werte die kleiner als 9 sind, sich selbst als verantwortlichen Knoten. Welche Verbindungen können noch hinzugefügt werden? Wenn sich Knoten 8 während seines Join-Prozesses mit allen Knoten verbindet, mit denen er unterwegs kommuniziert, geht viel der in FreeNet geforderten Anonymität verloren. In unserem Beispiel wäre Knoten 8, nach der Join-Operation mit 4 von 5 Knoten verbunden (Knoten 3,4,5,9). Somit funktioniert nur eine Verbindung nicht, aber alle Verbindungen aufzubauen ist auch schlecht. Somit muss ein Kompromiss gefunden werden.
Zunächt einmal soll der joinende Knoten nicht nur hinzugefügt werden, sondern in das Netzwerk integriert werden. Das soll heißen, es gibt eine Stelle im Netzwerk an die dieser Knoten passt, sodass er

einen kleineren Vorgänger und einen größeren Nachfolger besitzt. In unserem Beispiel ist dies zwischen Knoten 5 und Knoten 9. Deshalb wird immer noch der nächste Knoten gesucht (Zielknoten). Dieser Knoten bestimmt dann, mit welchem Knoten sich der joinende Knoten noch verbinden soll. Dazu bestimmt dieser, ob der Identifier des joinenden Knoten kleiner oder größer als der eigene Identifier ist. Ist der Identifier des Zielknoten größer, so soll sich der joinende Knoten mit dem nächst kleineren Knoten aus den Kontakten des Zielknotens verbinden. Ist der Identifier kleiner, so soll sich der joinende Knoten mit dem nächst größeren Knoten aus den Kontakten des Zielknotens verbinden. In unserem Beispiel führt dieser Prozess zu genau dem gewünschten Ergebnis 3.3 (a). Jedoch soll der joinende Knoten eine weitere Verbindung aufbauen. Da bei FreeNet die Rede davon ist, dass man jemanden aus dem Netzwerk kennen muss um ihm beizutreten, soll sich der joinende Knoten noch mit genau diesem Verbinden. In unserem Beispiel wäre dies Knoten 3. Hier beginnt die Join-Operation von Knoten 8. Somit ergibt sich als Ergebnis 3.3 (b).




(a) Knoten 8 mëchte dem Netzwerk beitreten

(b) Knoten 8 ist dem Netzwerk beigetreten

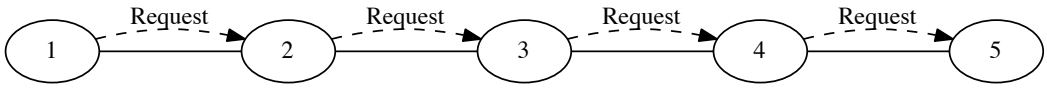
Abbildung 3.3: Join – Verbesserung

Führt Knoten 8 jetzt einen Lookup nach Identifier 4 durch, so ermittelt Knoten 8 Knoten 5 oder 3 als nächsten seiner Kontakte (Implementierungsabhängig) und leitet den Lookup an Knoten 5 oder Knoten 3 weiter. Dieser Knoten stellt fest, dass er Knoten 4 als Verbindung hat und sendet diesem den Lookup. Also kann nun von Knoten 8 Knoten 4 erreicht werden.

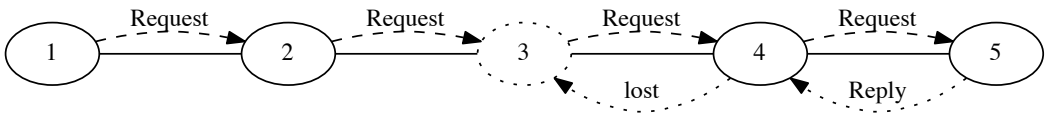
## 3.3 Zweites Problem – Verhalten unter Churn

Ein weiteres Problem tritt bei der Verwendung von Churn auf. Wie kann ein Knoten feststellen, dass einer seiner Verbindungsknoten offline ist. Erst einmal gar nicht! Dadurch entstehen 2 Probleme. Einmal kann einem Knoten, welcher offline ist eine Nachricht zugesendet werden, wodurch diese

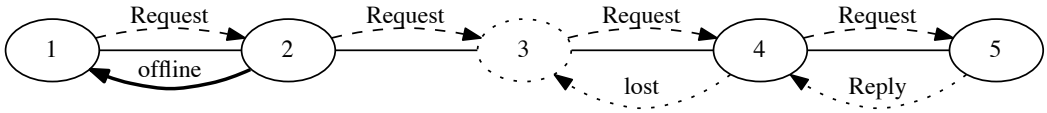
Nachricht verloren geht, da es keinen Empfänger gibt. Würde TCP verwendet gäbe es kein Problem, da TCP feststellt, dass niemand auf das SYN Packet antwortet. Jedoch verwenden wir UDP und das sendet nur die Daten los, ohne Zuverlässigkeit ob das Packet ankommt. Wir verwenden UDP da dies wesentlich weniger Nachrichten benötigt und keine Verbindungsinformationen gehalten werden müssen. Das macht Denial of Service Angriffe, wie SYN-Flooding unmöglich. Dabei werden viele SYN-Packete geschickt ohne das diese weiter benutzt werden. Der Kommunikationspartner speichert dies aber vorerst als Verbindung. Wird dies häufig durchgeführt so kann der Kommunikationspartner keine Anfragen mehr bearbeiten, da er ausgelastet ist. Das andere Problem ist etwas komplexer, deshalb werden wir dazu das Beispiel 3.4 betrachten. Hierbei schickt Knoten 1 einen Request mit




(a) Request




(b) Reply




(c) NodeOffline

Abbildung 3.4: Churn – Reply Problem

Identifier 5 los. Da das Routing jetzt funktioniert wird Knoten 5 erreicht 3.4 (a). Nachdem Knoten 3 den Request weitergeleitet hatte ging dieser offline. In 3.4 (b) wird gezeigt, was nun mit der Reply Nachricht passiert. Eigentlich ist in FreeNet gedacht, dass es auf einen Request immer eine Antwort bekommt, die entweder die Daten zurückliefet oder, dass die Daten nicht gefunden werden konnten. Jedoch erhält Knoten 1 keine Benachrichtigung, sondern wartet vergeblich immer weiter auf eine Antwort. Genauso wie Knoten 2. Das Netzwerk kann auch keinen anderen weg zurück finden, sondern

die Reply Nachricht kann nur den Weg zurück, welchen die Request-Nachricht gekommen ist.
Wie kann man diese Probleme beheben? Zunächst einmal betrachten wir so genannte PingPong-Nachrichten. Dies sind Nachrichten, die sich zwischen benachbarten Knoten zugesendet werden. Sollte auf eine Ping-Nachricht keine Pong-Nachricht zurück kommen, so wird der Knoten als offline angesehen. Die Frage ist nur, ob sich PingPong-Nachrichten eignen um dies in einem sehr großen Netzwerk durchzuführen. Dadurch würden extrem viele Nachrichten gesendet. Stattdessen werden wir einen anderen Mechanismus verwenden. Dieser Mechanismus verwendet einen Timer. Dieser Timer wird vom Ausgangsknoten gestartet, wenn eine Request-Nachricht losgeschickt wird. Sollte der Timer ablaufen, rejoined der Ausgangsknoten, wobei er aber seine bisherigen Kontakte behält. Danach wird die Request-Nachricht erneut losgeschickt. Ein Timer wird allerdings nicht nur vom Ausgangsknoten gestartet, sondern auch von jedem Knoten, der den Request weiterleitet. Dabei wird in der Request Nachricht die Dauer des Timers mitgeschickt. Somit wählt der nächste Knoten eine kleinere Zeit für den Timer, damit dieser vorher abläuft. Stellt ein Knoten fest, dass der Timer abgelaufen ist, so sendet dieser eine Nachricht an den Sender des Requests und joint dem Netzwerk neu. Der Sender des Requests wird diese Nachricht weiter zurückleiten bis der Ausgangsknoten erreicht ist, sodass dieser einen neuen Request starten kann 3.4 (c). Ein Knoten, dessen Timer abgelaufen ist und rejoint, kann weiter Nachrichten empfangen, denn er behält seine Kontakte. Jedoch kann er erst dann wieder Nachrichten senden, wenn der Join-Prozess abgeschlossen ist. In unserem Beispiel bedeutet dies, dass bei Knoten 2 der Timer abläuft und Knoten 1 eine Nachricht von Knoten 2 erhält, dass ein Knoten auf dem Weg offline ist. Deshalb schickt Knoten 1 den Request erneut los. Knoten 2 rejoint und erhält dadurch eine Verbindung zu 3. Hier funktioniert der Lookup jetzt, wenn kein Knoten zwischendurch offline geht. Auch das Problem, dass Requests verloren gehen, tritt nicht mehr auf, da der Timer ausläuft und ein neuer Request losgeschickt wird.

# Kapitel 4

# Implementierung

In diesem Kapitel wird erklärt, wie FreeNet im Simulator implementiert wurde. Dabei sind die drei Operationen, Join, Send und Forward von besonderem Interesse. Doch zunächst wird die Implementierung der grundlegenden Klassen erläutert, wie diejenigen, welche die FreeNet Knoten oder die Nachrichten im System darstellen.

## 4.1 Knoten

Die Knoten des Peer-to-Peer Netzwerks sind als *FreenetNode* Klasse implementiert. Sie enthält eine Funktion um dem Netzwerk beizutreten und startet die Operationen Join, Send und Forward. Hier werden ankommende Nachrichten verarbeitet und der Identifier gespeichert. Der Identifier wird in Form eines *FreenetContacts* gespeichert. Dies ist eine Klasse, welche den Identifier bildet und speichert. Jeder Knoten im Netzwerk besitzt eine eigene Instanz der *FreenetContact* Klasse. Dies sind auch die Informationen, welche zwischen Knoten ausgetauscht werden. Nämlich nicht die Knoten selber, sondern die Kontaktinformationen. Im folgenden sind die *FreenetContact* Instanzen als Kontakt bezeichnet.

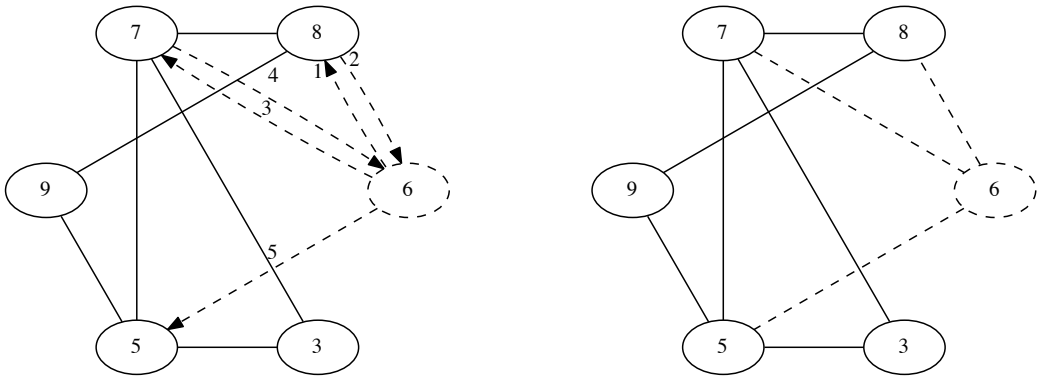
## 4.2 Nachrichten

Den vier Nachrichtentypen aus dem FreeNet Paper [Cla99] liegt eine so genannte *FreenetBaseMessage* zugrunde. Dies ist das Gerüst, auf dem die Nachrichten aufgebaut werden. Sie enthält die Informationen, welche jede der vier Nachrichtentypen besitzt. Dazu gehören Absender, Empfänger und ein Nachrichten Identifier (msgId), der wichtig für den Rückweg ist, da anhand dieser Informationen der

nächste Knoten des Rückwegs ermittelt werden kann. Außerdem eine TTL welche im nächsten Kapitel noch genauer untersucht wird und ein Timeout, welches unter Churn eine Rolle spielen wird. Jeder der vier Nachrichtentypen erhält dazu noch notwendige Erweiterungen. Dazu gehören zum Beispiel die Daten in der Reply- oder Insert-Nachricht oder der gesuchte Identifier bei einem Request.

## 4.3 Join

Die im Simulator implementierte Join-Operation ist die aus Kapitel 3.1.2 beschriebene verbesserte Version. Ein Knoten A bekommt in der Simulation von einem so genannten BootstrapManager einen Knoten B mit dem er seinen Join-Prozess starten kann. In Wirklichkeit gibt es diesen BootstrapManger nicht, jedoch muss, bei der verwendeten Version von FreeNet, ein Knoten bekannt sein um dem Netzwerk beitreten zu können. Es muss ein Knoten bekannt sein, da hier FreeNet als Darknet implementiert wurde [CSTV10]. Mit einem Darknet ist gemeint, dass man dem Netzwerk nur dann beitreten kann, wenn ein Knoten aus dem Netzwerk bereits bekannt ist. Der Gedanke dahinter ist, dass man sich nur mit einem Freund verbindet und dadurch Zugang zum Netzwerk erhält. Dadurch soll gewährleistet werden, dass jedem Knoten im Netzwerk vertraut werden kann. Nachdem Knoten A den Kontakt von Knoten B erhalten hat, sendet Knoten A eine *JoinRequestMessage*-Nachricht. Diese Nachricht teilt Knoten B mit, dass Knoten A dem Netzwerk beitreten möchte. In dieser Nachricht steht Absender der Nachricht und Empfänger. Außerdem der Kontakt von Knoten A und ein boolscher Wert, welcher angibt, ob dies die erste Join-Nachricht von Knoten A ist. Dieser Wert wird benötigt, da sich Knoten A mit dem Kontakt, den er vom BootstrapManager erhalten hat verbinden soll. Dies muss aber Knoten B mitgeteilt werden. Daher wird dieser Wert auf *true* gesetzt. Knoten B kann nun mit Hilfe des Kontakts von A den Identifier auslesen und mit diesem so verfahren, wie mit allen anderen Nachrichten. Er prüft, ob er für diesen Identifier verantwortlich ist. Ist Knoten B nicht verantwortlich für diesen Identifer, so sendet Knoten B an Knoten A eine *JoinNextReplyMessage*-Nachricht. Diese Nachricht enthält einen Kontakt zu einem Knoten C, welcher ein Kontakt von Knoten B ist. Knoten B hatte festgestellt das der Identifier von Knoten C näher am Identifier von Knoten C liegt als von sich selbst, daher sendet Knoten B an Knoten A diesen Kontakt. Knoten A sendet danach eine *JoinRequestMessage*-Nachricht in der der boolsche Wert für die erste Nachricht auf false steht. Dieser Prozess wird solange fortgeführt bis ein Knoten X feststellt, dass er für den Identifier von Knoten A verantwortlich ist. Dann sendet dieser Knoten X eine *JoinMeReplyMessage*-Nachricht. In dieser Nachricht steht der Kontakt eines weiteren Knoten Y. Dadurch weiß Knoten A, dass er sich mit Knoten X verbinden soll. Allerdings wurde in Abschnitt 3.1.2 die verbesserte Version der Join-Operation vorgestellt, wobei sich jeder Knoten der dem Netzwerk beitritt mit 3 Knoten verbindet. Bisher ist Knoten A mit Knoten B und Knoten X verbunden. Daher sendet Knoten A eine *JoinYouMessage*-Nachricht an Knoten Y. Dadurch weiß Knoten Y, dass Knoten A sich mit ihm verbindet. Wie Knoten X Knoten Y auswählt wurde in Abschnitt 3.1.2 erklärt.




(a) Join (b) Ergebnis

Abbildung 4.1: Join-Operation

Zur Verdeutlichung ein Beispiel in Abbildung 4.1 (a). Knoten 6 möchte dem Netzwerk beitreten und erhält vom BootstrapManager Knoten 8 als Kontakt. Also sendet Knoten 6 eine *JoinRequestMessage* Nachricht an Knoten 8 (1). Da der boolsche Wert in dieser Nachricht true ist, verbinden sich Knoten 6 und Knoten 8. Knoten 8 stellt fest, dass Knoten 7 näher am Identifier von Knoten 6 liegt und teilt dies, mit einer *JoinNextReplyMessage*-Nachricht, Knoten 6 mit (2). Daher sendet Knoten 6 eine *JoinRequestMessage*-Nachricht an Knoten 7 (3). Dieser stellt fest, dass er für den Identifier von Knoten 6 verantwortlich ist und ermittelt Knoten 5 als weiteren Kontakt für Knoten 6. Dies sendet Knoten 7 in einer *JoinMeReplyMessage*-Nachricht an Knoten 6 (4). Nun sendet Knoten 6 noch eine *JoinYouMessage*-Nachricht an Knoten 5 (5) um sich mit diesem zu verbinden. Das Ergebnis der Join-Operation ist in Abbildung 4.1 (b) zu sehen. Außerdem geschieht noch etwas. Knoten 7 sendet mit der JoinMeReply Nachricht alle Daten an Knoten 6, für die Knoten 7 verantwortlich war nun aber Knoten 6 verantwortlich ist. Das selbe macht Knoten 7, wenn dieser die *JoinYouMessage*-Nachricht erhält. Somit werden die Daten auch bei Knoten 6 gefunden.

## 4.4 Send und Forward

Warum es für Send und Forward eigene Operationen geben muss, wird im nächsten Kapitel erklärt. Diese Operationen werden gebraucht, um auf das Offline-gehen eines Knotens zu reagieren. Die Send Operation wird bei einem Request ausgeführt. Die Send Operation startet einen Timer, welcher abläuft wenn in einer gewissen Zeit keine Antwort auf den Request erhalten wird. Läuft der Timer ab,

so rejoint der Knoten. Das heißt er führt die Join-Operation ein weiteres mal aus, behält aber seine Kontakte bis auf den Kontakt, von dem er keine Antwort erhalten hat. Während der Knoten den Join-Prozess ausführt, kann dieser Nachrichten empfangen aber keine senden. Erst nachdem der Join-Prozess komplett durchlaufen ist kann der Knoten wieder Nachrichten senden.
Die Forward-Operation unterscheidet sich kaum von der Send-Operation. Die Forward-Operation wird gestartet, wenn ein Knoten einen Request erhält, und dieser Request weitergeleitet werden soll. Hierbei übernimmt die Forward-Operation den Wert für den Timeout vom Vorgängerknoten und dekrementiert ihn um eine Sekunde. Dadurch führt dies in der "Kette"von Knoten, welche die Nachricht weitergeleitet haben dazu, dass der letzte Knoten zuerst merkt, wenn keine Antwort zurück kommt. Dieser Knoten sendet dann eine NodeOffline Nachricht an den Ausgangsknoten. Dadurch erfährt auch jeder andere Knoten auf dem weg dorthin davon und kann die Forward-Operation beenden. Diese Nachricht ist wichtig, da auch bei einem Timeout der Forward-Operation dieser Knoten erneut einen Join durchführt. Würde nun jeder Knoten auf dem Weg auf den Timeout neu joinen, so könnte dies dazu führen, dass große Teile des Netzwerks keine Nachrichten senden können. Daher sendet der erste Knoten die NodeOffline-Nachricht und verhindert so, dass die Forward-Operation einen Timeout auslöst. Dadurch joint nur ein Knoten noch einmal und der Rest des Netzwerks bleibt bestehen.
Welcher Wert für den Timeout der Send-Operation sinnvoll ist muss noch im nächsten Kapitel ermittelt werden.

In diesem Kapitel haben wir gesehen, wie das Netzwerk im Simulator implementiert wurde und wie mit zwei Problemen umgegangen wurde um diese zu lösen. Im nächsten Kapitel werden wir mit dieser Implementierung verschiedene Simulationen durchführen und auswerten.

# Kapitel 5

# Simulation und Auswertung

In diesem Kapitel werden Simulationen durchgeführt und die Ergebnisse ausgewertet. Die Simulationen werden mithilfe des oben beschriebenen Simulators PeerfactSim.KOM durchgeführt. Dabei wird das Verhalten mit und ohne Churn von Bedeutung sein. Es werden Metriken aufgestellt, welche die Routing Funktion bewerten und die Erreichbarkeit von Dateien evaluieren. Dies sind Fragen, die uns in diesem Kapitel beschäftigen werden. Außerdem müssen die im vorherigen Kapitel erwähnten Werte, TTL und Timeout, noch genauer bestimmt werden. Abschließend soll noch ausgewertet werden, wie anonym diese FreeNet ähnliche Implementierung wirklich ist. Wir werden sehen, unter welchen Bedingungen der Absender einer Nachricht am anonymsten bleibt und unter welchen er relativ gut erraten oder bestimmt werden kann.

## 5.1 TTL

Die TTL (time-to-live) einer Nachricht gibt an nach wie vielen Hops die Nachricht verworfen werden soll. In diesem Abschnitt soll ein geeigneter Wert für die TTL gefunden werden, der nicht zu niedrig ist, sodass Nachrichten ihr Ziel gar nicht erreichen und auch nicht zu hoch, sodass Nachrichten zu lange im Netzwerk bleiben (Routing-Schleifen, etc...), ohne irgendeinen Nutzen. Dadurch könnte das Netzwerk überlastet werden, da zu viele Nachrichten verarbeitet werden, die besser verworfen worden wären. In diesem Abschnitt wird allerdings auch noch ein Ansatz aufgegriffen, der die TTL über eine Wahrscheinlichkeit implementiert. Wir werden sehen, dass dieser Ansatz zur Absenderanonymität beitragen wird.

### 5.1.1 Netzwerkdurchmesser

Um die TTL vernünftig einstellen zu können, muss zunächst der Netzwerkdurchmesser festgestellt werden. Der Netzwerkdurchmesser ist die größte mögliche Entfernung zwischen zwei Knoten, bei der auf schnellstem Weg geroutet wird. Hierbei wird die Distanz in Hops (Sprüngen) angegeben. Ein Hop ist der Schritt von einem Knoten zum nächsten. Das bedeutet, der Netzwerkdurchmesser ist die größte Distanz die eine Nachricht zurücklegen muss, um das Ziel zu erreichen, wenn die Nachricht auf kürzestem Weg zum Ziel kommt. Um den Netzwerkdurchmesser zu berechnen, verwenden wir eine Java Bibliothek mit dem Namen GraphStream [GP12]. Mit dieser Bibliothek ist es möglich einen Graphen zu erzeugen und mit einem Packet mit dem Namen algo den Netzwerkdurchmesser zu berechnen.
Es wird der Netzwerkdurchmesser bei 100, 1000, 2000, ..., 10000 Knoten berechnet. Da an einigen Stellen, der Simulation, Zufallszahlen gebraucht werden, verwendet PeerfactSim.KOM einen Mechanismus, der dafür sorgt, dass bei gleicher Parameterwahl auch immer das selbe Ergebnis heraus kommt. Damit man aber auch unterschiedliche Ergebnisse erhalten kann gibt es so genannte Seeds. Wählt man einen anderen Seed, so erhält man auch andere Zufallswerte. Hier verwenden wir 10 Seeds und bilden danach das arithmetische Mittel über die Ergebnisse mit gleicher Knotenzahl. Das arithmetische Mittel sorgt dafür, dass, der Wert besser auf alle möglichen Simulationen hochgerechnet werden kann, da mehr Werte das Ergebnis stützen. Wir werden im folgenden häufig von einer Simulation sprechen, jedoch werden dies meistens 10 Seeds sein, über die das arithmetische Mittel gebildet wird. Die Parameter der Simulationen sind in Tabelle 5.1 kompakt aufgeführt.

| | |
|---|---|
| Churn | aus |
| Inserts | aus |
| Requests | aus |
| Knoten | 100, 1000, 2000, ... , 10000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.1: Netzwerkdurchmesser – Parameter der Simulationen

Das Ergebnis der Simulationen tragen wir in einem Graphen auf. Hier beschreibt die x-Achse die Anzahl der Knoten und die y-Achse die gemittelten Netzwerkdurchmesser (Abbildung 5.1). In diesem Graphen ist zu sehen, dass bei 100 Knoten nur maximal 5 bis 6 Hops gebraucht werden. Danach steigt die Hopanzahl auf 7 bei 1000 Knoten. Diese Zahl wird in folgenden Abschnitten noch interessant sein, denn wir wollen ein Netzwerk von 1000 Knoten erzeugen. Nach 7 Hops bei 1000 Knoten steigt die Kurve langsamer an bis bei 5000 Knoten 9 Hops und bei 10000 Knoten 10 Hops benötigt werden. Für ein normales Peer-to-Peer Netzwerk wäre die Knotenanzahl zu klein. Jedoch ist anzunehmen, dass die Darknet Variante von FreeNet eine Nutzeranzahl in dieser Größenordnung besitzt.

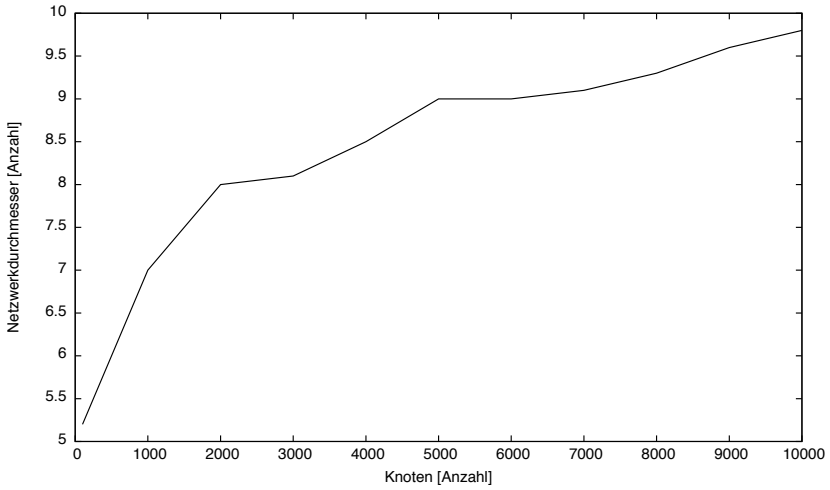
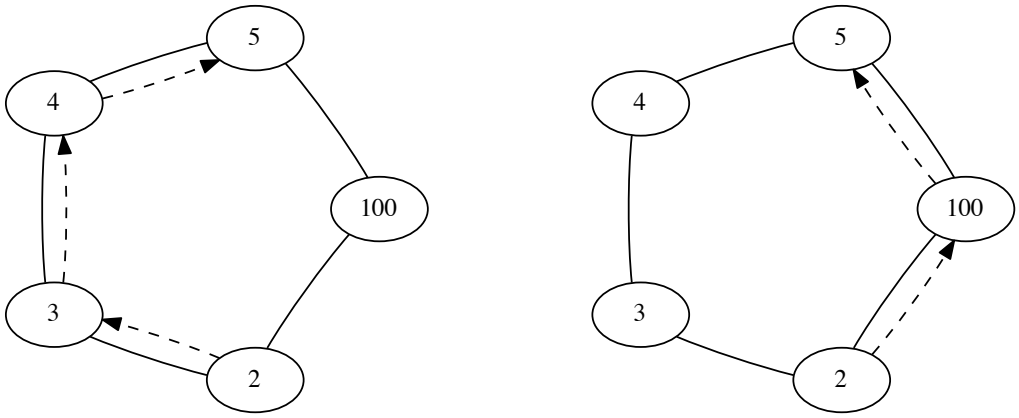



Abbildung 5.1: Netzwerkdurchmesser

Allerdings sollten größere Netzwerke keine viel größeren Werte ergeben, da der Graph anscheinend logarithmisch anwächst.

### 5.1.2 Bestimmung der TTL

Wir wollen ein Netzwerk von 1000 Knoten simulieren, daher brauchen wir eine TTL von mindestens 7. Jedoch haben wir 7 als Mittelwert aus mehreren Simulationen gebildet, daher verwenden wir eine TTL von 8. Doch hier ergibt sich ein Problem. Jedem Knoten muss diese TTL bekannt sein, wenn dieser Nachrichten verschicken will. Daher könnte ein Knoten der eine Nachricht empfängt, berechnen wie viele Knoten vor ihm die Nachricht hatten. Speziell der erste Knoten, der die Nachricht weiterleitet, wird wissen wer die Nachricht abgeschickt hat. aber genau dies sollte FreeNet ja verhindern. Niemand sollte wissen, wer der Absender der Nachricht war.
Um dies wieder anonymer zu gestallten, verwendet jeder Knoten eine TTL die größer als 8 ist. Hierbei muss darauf geachtet werden, dass die TTL nicht zu groß wird. Was zum Beispiel dadurch geregelt werden kann, dass eine TTL von 8 am wahrscheinlichsten ist und größere Werte unwahrscheinlicher werden. Dadurch kann es zwar sein, dass ein paar Nachrichten recht lange unterwegs sind, jedoch kann kein Knoten mehr sagen wie viele Knoten vor ihm die Nachricht schon hatten. Doch wieder kann der erste Knoten, der die Nachricht weiterleitet, genau genug ermitteln wer der Absender war. Denn je höher die TTL, die der Absender wählt ist, umso wahrscheinlicher wird es, dass der Knoten von dem die Nachricht kam der Absender war. Somit kann er nicht genau sagen, dass der Knoten vorher der Absender war, aber er kann dies relativ gut raten.




(a) Routing nach FreeNet (b) Optimales Routing – kürzester Weg

Abbildung 5.2: TTL – Routingproblem

Als nächstes Simulieren wir das Netzwerk mit einer TTL von 8 und 1000 Knoten. Eine TTL von 8 müsste reichen um jedes Ziel zu erreichen. In der Simulation werden nur Insert Nachrichten gezählt, da diese auf jeden Fall bis an das Ziel kommen müssen (Bei Request muss für eine erfolgreiche Suche nicht unbedingt das Ziel erreicht werden siehe Kapitel 3.1). Jeder Knoten des Netzwerks sendet einen Insert und es wird gezählt wie viele davon das Ziel erreichen. Die Parameter dieser Simulation können in Tabelle 5.2 nachgelesen werden.

| Churn | aus |
|---|---|
| Inserts | an |
| Requests | aus |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.2: TTL – Parameter der Simulation mit einer TTL von 8

| Gesendete Nachrichten: | 999 |
|---|---|
| Angekommene Nachrichten: | 676 |
| Verlorene Nachrichten: | 323 |

Tabelle 5.3: TTL – Ergebnis der Simulation mit einer TTL von 8

Das Ergebnis (Tabelle 5.3) ist anders als wir erwartet hatten. Erst einmal sollten 1000 Inserts losgeschickt werden und hier sind es nur 999. Das liegt aber daran, dass ein Knoten die Daten bei sich

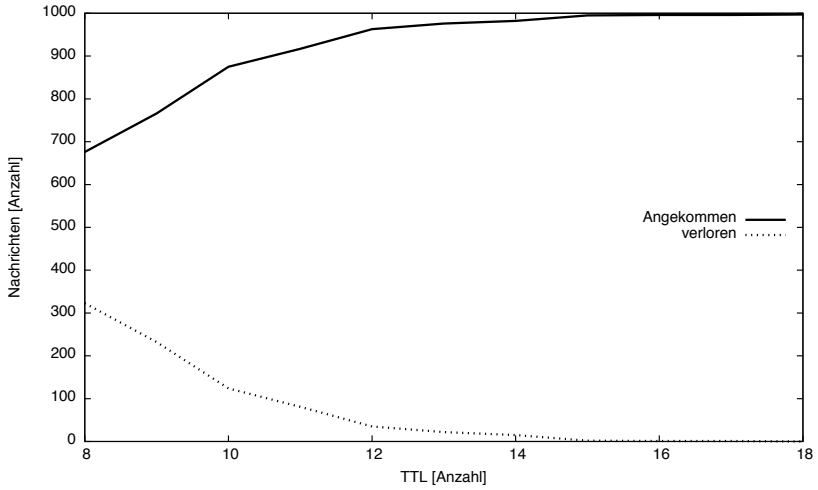



Abbildung 5.3: TTL – Ergebnis der Simulationen

selbst gefunden hat und somit keine Nachricht losschickt. Zweitens sollte jetzt jede Nachricht ankommen, aber anscheinend gehen immer noch Nachrichten verloren. Weiter oben stand jedoch etwas von einem kürzesten Weg und wir verwenden nicht den kürzesten Weg. Dazu ein Beispiel (Abbildung 5.2), in dem Knoten 2 einen Insert mit Identifier 5 schickt. Hierbei sieht man bei Abbildung 5.2 (a), wie der Weg unter FreeNet aussieht. Bei (b) sieht man wie optimal geroutet würde. Daher müssen wir nun eine TTL bestimmen, ab der alle Nachrichten ankommen. Dazu führen wir weitere Simulationen durch wobei die Parameter genauso gewählt wurden wie bei der TTL von 8. Jedoch inkrementieren wir diesen Wert nun mit jeder Simulation, bis alle Nachrichten ankommen.

| Churn | aus |
|---|---|
| Inserts | an |
| Requests | aus |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |
| TTL | 8 - 18 |

Tabelle 5.4: TTL – Parameter der Simulationen

In Abbildung 5.3 ist zu sehen, dass erst bei einer TTL von 18 keine Nachrichten mehr verloren gehen. Dazu wählt man eine Sicherheit von etwa 2, damit auch bei anderen Seeds keine Nachrichten verloren gehen. Somit erhält man eine TTL von 20 bei einem Netzwerk von 1000 Knoten. Damit hat sich das Ergebnis aus der Berechnung des Netzwerkdurchmessers verdreifacht. Dazu kommt, dass ein Knoten recht gut tippen kann, welcher Knoten die Anfrage gesendet hat. Letzteres kann man mit dem im nächsten Abschnitt erklärten Verfahren beheben.

### 5.1.3 Wahrscheinlichkeit

Ein anderer Ansatz wird in der Bachelorarbeit “Untersuchung von Anonymisierungsverfahren in DHT basierten Overlays “ erwähnt [Yil15]. Dort wird keine feste TTL verwendet, sondern mit Wahrscheinlichkeiten gearbeitet. Dabei sendet ein Knoten eine Nachricht mit einer bestimmten Wahrscheinlichkeit weiter. Dabei können wir aber trotzdem unsereTTL gebrauchen, da wir diesen Wert als Erwartungswert verwenden können. Wenn wir also unser 1000 Knoten Netzwerk mit Hilfe von Wahrscheinlichkeiten anstatt mit einer festen TTL verwenden möchten, so empfiehlt sich ein Erwartungswert, sodass der zwanzigste Knoten die Nachricht verwirft. In der nächsten Simulation liegt die Wahrscheinlichkeit, dass eine Nachricht verworfen wird, bei 2%. Damit liegt der Erwartungswert bei 20.

| Churn | aus |
|---|---|
| Inserts | an |
| Requests | aus |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |
| TTL | 2% |

Tabelle 5.5: TTL Wahrscheinlichkeit – Parameter der Simulation

| Gesendete Nachrichten: | 998 |
|---|---|
| Angekommene Nachrichten: | 938 |
| Verlorene Nachrichten: | 60 |

Tabelle 5.6: TTL Wahrscheinlichkeit– Ergebnis der Simulation

Wie in Tabelle 5.6 zu sehen ist kommen nicht alle Nachrichten an. Da etwa 6% der Nachrichten nicht ankommen, könnte man sich den Mechanismus überlegen, der die Nachricht nach einer gewissen Zeit erneut losschickt. Bei der TTL hätte dies keinen Sinn gemacht, jedoch hier kann die Nachricht jedes mal unterschiedlich weit kommen, da dies vom Zufall abhängig ist. Dadurch könnte sich die Zahl der Nachrichten, die nicht ankommen weiter verringern. Angenommen man würde die Nachricht drei mal losschicken, so würde im Schnitt weniger als eine Nachricht verloren gehen, wenn immer 6% der Nachrichten nicht ankommen ($1000 * 0,06^3 = 0.216$). Doch dies soll hier nicht betrachtet werden und wäre etwas für spätere Arbeiten.

In allen folgenden Simulationen werden wir eine statische TTL verwenden, da dadurch alle Nachrichten ankommen und so besser festgestellt werden kann, wie gut FreeNet funktioniert, ohne dass Nachrichten durch die TTL verloren gehen. Jedoch gibt es bei einem Request, also den Operationen

Send und Forward, noch eine weitere Größe, die Nachrichten verwerfen kann. Diese Größe ist der Wert für die Dauer bis ein Timeout auftritt. Da ein Rejoin jedoch nur bei Churn auftritt, wird dies im Kapitel zum Verhalten mit Churn (Kapitel 5.3) näher untersucht.

## 5.2 Verhalten ohne Churn

In diesem Abschnitt sollen Eigenschaften des Netzwerks untersucht werden, wie wie viel Hops werden für einen Lookup durchschnittlich gebraucht. Werden alle Daten gefunden oder kommt es zu Datenverlust. Außerdem besitzt FreeNet eine Eigenschaft, die dazu führt, dass Daten von Knoten vergessen werden. Sonst würde dies auch dazu führen, dass irgendwann jeder Knoten alle Daten hat. Jedoch kann jeder Knoten nur begrenzt viele Daten aufnehmen (In dieser Implementierung 5 Datenpakete). Wie lange ist eine Datei also im Netzwerk verfügbar und wo liegt sie dann? Des weiteren soll ermittelt werden, wie viele Bytes in den Nachrichten hin und her geschickt werden.

### 5.2.1 Geschwindigkeit

Der erste Punkt, der uns interessiert, ist wie schnell wir ein Ziel erreichen. Dabei ist es schwierig dies in Zeit anzugeben, da dies von den Verbindungen der einzelnen Knoten untereinander abhängt. Liegt der eine in Europa der nächste in Amerika und der übernächste in Asien, so sind die Zeiten der Verbindungen untereinander relativ groß. Wir würden aber gerne ein allgemeines Model betrachten. Deshalb zählen wir die Anzahl der Hops, die eine Nachricht vom Ursprung zum Ziel durchläuft. Bei der TTL wollten wir herausfinden, wie viele Hops eine Nachricht maximal benötigt. Hier jedoch wollen wir eine durchschnittliche Anzahl. Wir können hier, für die Geschwindigkeit, die Anzahl der Hops verwenden, da die Anzahl der Hops eine von der physischen Verbindung losgelöste Größe ist und sich somit für ein allgemeines Model eignet.

| Churn | aus |
|---|---|
| Inserts | 450 |
| Requests | 100 |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.7: Verhalten ohne Churn – Parameter der Simulation zur Geschwindigkeit

Wir werden 1000 Knoten simulieren von denen 450 Knoten Daten im Netzwerk ablegen (Parameter in Tabelle 5.7). Danach werden 100 Knoten nach jeweils einer zufällig ausgewählten Datei suchen. Wie

bei allen Simulationen in diesem Abschnitt simulieren wir ohne Churn. Dabei werden nur Requests und Inserts gewertet, da dies die einzigen beiden Nachrichtentypen, die nach dem Identifier geroutet werden.

| | |
|---|---|
| Gesendete Nachrichten: | 550 |
| Anzahl der Hops: | 3196 |
| durchschnittliche Hopzahl: | 5.8109 |

Tabelle 5.8: Verhalten ohne Churn – Ergebnis der Simulation zur Geschwindigkeit

Das Ergebnis (Tabelle 5.8) zeigt, dass durchschnittlich nur etwas weniger als 6 Hops benötigt werden um an das Ziel zu kommen. In Anbetracht dessen, was wir über die TTL erfahren haben, ist dies schon außergewöhnlich, denn man brauchte eine TTL von fast 20 damit alle Nachrichten ankommen. Aber durchschnittlich nur 5 Hops erscheint relativ wenig. Wir müssen uns aber die Frage stellen ob weniger Hops wirklich besser sind, da je weniger Hops es im Durchschnitt sind, desto unanonymer wird das Netzwerk. Wenn es viele Hops sind kann die Nachricht schlecht zurückverfolgt werden. Wenn es wenig Hops sind wird dies einfacher, da man, um den Absender ermitteln zu können, einen Teil der Kette von Absender zu Ziel kontrollieren muss. Dadurch kann man den Weg der Datei nachvollziehen.

### 5.2.2 Inserts

Erreichen die Daten, die durch eine DataInsertMessage-Nachricht verschickt wurden auch den richtigen Knoten oder werden sie an falscher Stelle abgelegt? Um dies zu ermitteln starten wir eine Simulation mit den Parametern aus Tabelle 5.9 Während der Simulation messen wir, wie viele Inserts

| | |
|---|---|
| Churn | aus |
| Inserts | 450 |
| Requests | aus |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.9: Verhalten ohne Churn – Parameter der Simulation zu Inserts

losgeschickt wurden und wie viele am richtigen Knoten ankommen. Das Ergebnis ist eindeutig (Tabelle 5.10). Alle Inserts haben den richtigen Knoten erreicht und dort die Daten abgelegt. Dadurch können wir uns sicher sein, dass das hinzufügen von Daten in das Netzwerk, ohne Churn, richtig funktioniert. Jedoch unter der Voraussetzung, dass keine Daten durch die TTL verworfen werden.

| Gesendete Inserts: | 450 |
|---|---|
| Erfolgreiche Inserts: | 450 |

Tabelle 5.10: Verhalten ohne Churn – Ergebnis der Simulation zu Inserts

### 5.2.3 Requests

In diesem Versuch wollen wir herausfinden, ob für jeden Request nach einer Datei, die vorher im Netzwerk abgelegt wurde, auch die Daten zurückgesendet werden. Dazu simulieren wir erneut ein Netzwerk von 1000 Knoten. Die restlichen Parameter sind äquivalent zu denen, der letzten Simulation, bis auf die Requests, die nun dazu kommen. Es werden 100 Knoten einen Request durchführen und danach wird gemessen, wie viele Antworten, die entsprechenden Daten beinhalten.

| Churn | aus |
|---|---|
| Inserts | 450 |
| Requests | 100 |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.11: Verhalten ohne Churn – Parameter der Simulation zu Requests

Auch hier fällt das Ergebnis (Tabelle: 5.12) eindeutig aus. Auf alle Requests, erhält der entsprechende Knoten die richtigen Daten. Das bedeutet, dass ohne Churn Daten im Netzwerk abgelegt werden können und diese auch wiedergefunden werden.

| Gesendete Requests: | 100 |
|---|---|
| Erfolgreiche Requests: | 100 |
| Fehlgeschlagene Requests | 0 |

Tabelle 5.12: Verhalten ohne Churn – Ergebnis der Simulation zu Requests

### 5.2.4 Verfügbarkeit von Dateien

Wir wollen herausfinden, wie sich eine Datei im Netzwerk verbreitet, da jeder Knoten, der eine *DataReplyMessage*-Nachricht empfängt, die darin enthaltene Datei speichert. Was passiert, wenn eine Datei besonders “populär“ ist und viel gesucht wird? Was passiert mit weniger “populären“ Dateien? Diese Fragen sollen in diesem Abschnitt geklärt werden.

#### Wo sind die Dateien gespeichert?

Diese Frage klingt banal, da wir weiter oben gesehen haben wie Dateien im Netzwerk abgelegt werden und auch an welcher Stelle. Jedoch kann sich die Datei auch auf andere Knoten verschieben, wie durch das neu Hinzukommen von Knoten, welche von deren Nachbarknoten Dateien bekommen (siehe Kapitel 4.4), oder durch das Speichern der Dateien aus den DataReplyMessage-Nachrichten entlang des Rückwegs. Dazu simulieren wir ein Netzwerk von 1000 Knoten und ermitteln die Anzahl der Knoten, die eine entsprechende Datei speichern. Dazu simulieren wir, dass 500 Knoten nach der selben Datei suchen, die vorher im Netzwerk abgelegt wurde. Dieses Szenario kann sogar realistisch sein, da es Dateien gibt, welche häufiger angefordert werden als andere. Dies trifft zum Beispiel auf besonders häufig angefragt Dateien zu. Doch wie viele Knoten besitzen diese Datei zu unterschiedlichen Zeitpunkten der Simulation. Die Parameter der Simulation sind inTabelle 5.21 aufgeführt.

| Churn | aus |
|---|---|
| Inserts | 500 ab 31 [min] bis 60 [min] |
| Requests | 500 ab 90 [min] bis 140 [min] |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 150 |

Tabelle 5.13: Verhalten ohne Churn – Parameter der Simulation zum Speicherort der Dateien

Das Ergebnis (Abbildung 5.4) zeigt, dass nur 500 Knoten einen Request nach dem Identifier, der Datei, senden, aber etwa 700 Knoten die Datei nach der Simulation besitzen, obwohl diese vorher nur auf einem Knoten gespeichert war. Das bedeutet, 20% des Netzwerks bekommen die Datei, ohne danach gesucht zu haben. Dadurch erreichen wir, dass Dateien, die häufig gesucht werden, schneller gefunden werden, da nicht der, für die gesuchte Datei, verantwortliche Knoten, die Antwort sendet, sondern ein Knoten, der den Request vorher bekommt und die Datei schon besitzt. Durch diesen Mechanismus werden häufig gesuchte Dateien verbreitet und selten gesuchte Dateien, von den häufig gesuchten, überschrieben, da jeder Knoten nur begrenzt viele Dateien speichern kann.

#### Verweildauer von Dateien im Netzwerk

Wie lange Dateien im Netzwerk bleiben, hängt davon ab wie gefragt diese Dateien sind. Wie wir gerade gesehen haben verteilt sich eine “populäre“ Datei schnell im Netzwerk. Selten gesuchte Dateien werden jedoch von dien “populären“ überschrieben. Es lässt sich somit keine Zeit angeben, wie lange eine Datei im Netzwerk bleibt. Bisher hatten wir in unserer FreeNet ähnlichen Implementierung keine Möglichkeit Dateien zu löschen. Diese Möglichkeit haben wir immer noch nicht. Jedoch

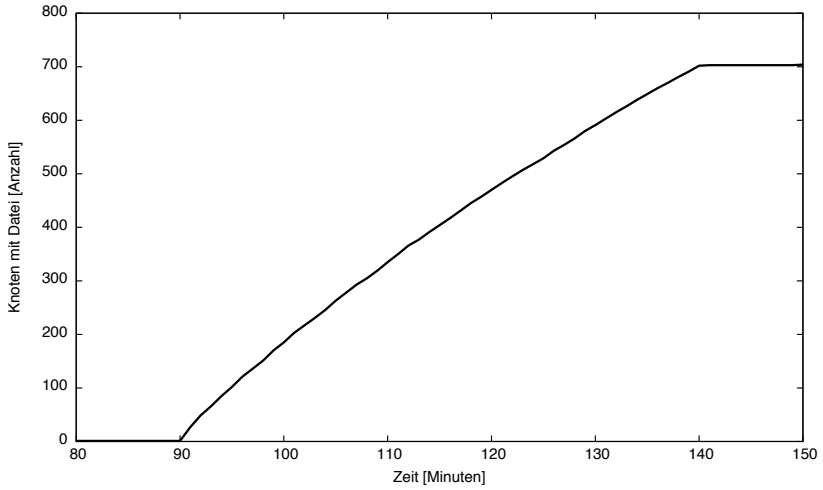



Abbildung 5.4: Verhalten ohne Churn – Ergebnis der Simulation zum Speicherort der Dateien

werden manche Daten mit der Zeit überschrieben. Das führt zu einem anderen Vorteil, denn niemand kann einfach Dateien aus dem Netzwerk entfernen, damit nicht mehr darauf zugegriffen werden kann. Dieser Mechanismus schützt das Netzwerk vor Zensur.

### 5.2.5 Gesendete Bytes

Als letzte Simulation ohne Churn, werden wir herausfinden wie viele Bytes mit den Nachrichten gesendet werden. Wir wollen wissen wie stark das Netzwerk ausgelastet wird und ob es Zeiten in der Simulation gibt zu denen mehr Bytes gesendet werden als zu anderen.
Die Parameter dieser Simulation sind in Tabelle 5.14 aufgeführt. Hierbei wird ein Netzwerk von 1000 Knoten simuliert und die Dateigrößen mit dem Zufallsgenerator des Simulators bestimmt. Allerdings mit einem maximalen Wert von 40kB um die Dateien nach oben hin zu begrenzen. Dieser Wert ist willkürlich festgelegt und kann durch einen beliebigen anderen ersetzt werden. Sie sollte nur so Groß sein, dass sie sich im Graphen später gut von den anderen Nachrichten abheben.

| Churn | aus |
|---|---|
| Inserts | 500 |
| Requests | 500 |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.14: Verhalten ohne Churn – Parameter der Simulation zu gesendeten Bytes pro Minute

In Abbildung 5.5 ist das Ergebnis dieser Simulation zu sehen. Hierbei lassen sich die unterschiedlichen Phasen der Simulation gut erkennen. Am Anfang treten die Knoten dem Netzwerk bei, was sehr wenige Bytes benötigt. Dafür werden hier sehr viele Nachrichten gebraucht, die jedoch kaum Daten enthalten. Danach erfolgen, bei Minute 30, die Inserts und Dateien werden im Netzwerk abgelegt. Hier erzeugt dies die meisten Bytes in den Nachrichten. Wir erreichen einen Wert von fast 2.5MB pro Minute, was das Maximum bei dieser Simulation ist. Als letzte Phase werden die Requests losgeschickt. Hierbei werden ebenfalls 500 Nachrichten losgeschickt doch der Wert ist wesentlich geringer, als bei den Inserts. Dazu haben wir im letzten Abschnitt gesehen, wie gut sich Dateien im Netzwerk verbreiten. Das führt dazu, dass weniger Nachrichten geschickt werden müssen und somit auch weniger Bytes aus großen Dateien. Hier liegt die Kurve der Nachrichten bei den Requests höher als bei den Inserts, jedoch werden hier Request und Reply Nachrichten gezählt, was dazu führt, dass wir nur halb so viele Reply Nachrichten, mit den Daten haben, wie in der Kurve zu sehen ist. Zwischen Minute 60 und Minute 80 werden eigentlich keine Daten gesendet, doch um den Graphen anschaulicher zu machen wurden die Daten Bezier interpoliert um eine glatte Kurve zu erhalten.
Das Ablegen der Daten im Netzwerk erfordert es mehr Bytes zu senden, als das Anfragen der Daten, da die Datei definitiv bis zum verantwortlichen Knoten gelangen muss und nicht schon früher die Datei abgelegt werden kann. Aber die Menge an Bytes welche in Nachrichten geschickt wird hängt vor allem von den Größen der Dateien ab.

Wir haben gesehen, dass das Netzwerk ohne Churn gut funktioniert. Alle Dateien werden an der richtigen Stelle abgelegt und auch per Request wiedergefunden. Außerdem wissen wir nun, dass etwa 20% der Knoten im Netzwerk Dateien erhalten, für die sie nicht verantwortlich sind und nach denen sie auch keinen Request gesendet haben. Des weiteren scheint ein Insert, im Durchschnitt, mehr Bytes zu brauchen, als ein Request, obwohl ein Request noch eine Reply oder RequestFailed Nachricht auslöst, was aus 500 gesendeten Requests auch 500 Antworten macht. Dadurch sind es doppelt so viele Nachrichten und trotzdem weniger Daten.

## 5.3 Verhalten mit Churn

In diesem Abschnitt sollen die selben Simulationen wie im letzten Kapitel durchgeführt werden, nur dieses Mal mit Churn. Wir werden sehen wie sich Churn auf die guten Ergebnisse des letzten Abschnitts auswirkt.

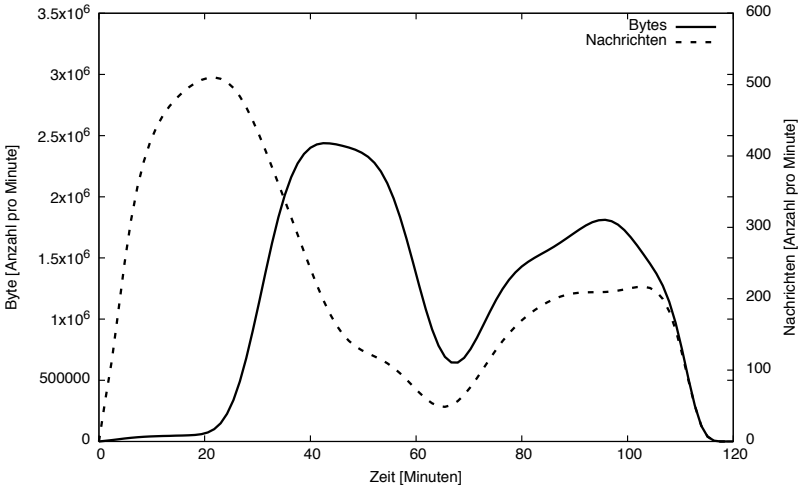



Abbildung 5.5: Verhalten ohne Churn – Ergebnis der Simulation zu gesendeten Bytes pro Minute

### 5.3.1 Erste Simulation mit Churn

Als erstes wollen wir herausfinden, wie viele Nachrichten mit Churn noch ankommen. Dazu betrachten wir eine Simulation mit 1000 Knoten. Von diesen 1000 Knoten senden 100 eine Insert Nachricht. Diese 100 Knoten gehen die komplette Simulation nicht offline. Von den anderen Knoten werden immer mehr aus dem Netzwerk verschwinden, bis nur noch die 100 Knoten übrig sind. Die Simulation wird mit den Parametern aus Tabelle 5.15 ausgeführt.

| Churn | an ab Minute 30 |
|---|---|
| Inserts | 100 |
| Requests | aus |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.15: Verhalten mit Churn – Parameter der Simulation zu Churn

Abbildung 5.6 zeigt, dass die Anzahl der ankommenden Nachrichten schnell auf 60 - 80% abfällt. Das bedeutet, dass 20-40% verloren gehen, und das bereits nachdem erst 30% der Knoten offline sind. Danach jedoch fällt die Anzahl der ankommenden Nachrichten nicht mehr so stark ab wie vorher. Ob 30% der Knoten oder 90% offline sind macht kaum etwas aus. Hier bewegt sich der Nachrichtenverlust bei 40 - 50%. Das Problem der Insert Nachrichten ist, dass Knoten nicht feststellen können, ob ihre Kontakte offline sind. Das dann immer noch 50% der Nachrichten ankommen ist somit ein gutes Ergebnis. Da es in der Realität nicht die abgetrennten Phasen Join, Insert, Request gibt, sondern alles gleichzeitig abläuft hilft dies hierbei, da durch die Send und Forward Operationen Knoten

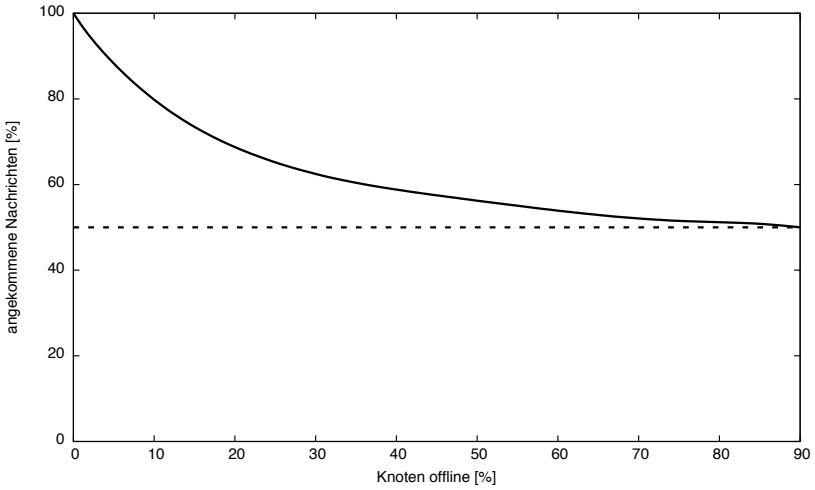



Abbildung 5.6: Verhalten mit Churn – Ergebnis der Simulation zu Churn

erkannt werden, die offline sind und somit das Netzwerk wieder neu aufbauen. Damit können diese Verbindungen auch für Insert Nachrichten genutzt werden und Kontakte, die offline sind, werden aus den Kontakten gelöscht, sodass weniger Nachrichten verloren gehen. Es gehen weniger Nachrichten verloren, da weniger Nachrichten an Knoten gesendet werden, die das Netzwerk bereits verlassen haben. Dies effizienter zu machen wäre ein Thema für eine zukünftige Arbeit und wird hier nicht weiter betrachtet.

In den folgenden Simulationen werden wir davon ausgehen, dass 30% der Knoten offline gehen. Dies werden allerdings nie die Knoten sein, die einen Request senden, denn ein Knoten, der einen Request sendet, möchte Daten erhalten und wird nicht das Netzwerk verlassen. Abbildung 5.6 hat gezeigt, dass 30% ein Wert ist, ab dem der Wert der verloreneren Nachrichten annähernd konstant bleibt.

### 5.3.2 Geschwindigkeit

Die Simulation in 5.2.1 hat gezeigt, dass durchschnittlich etwa 6 Hops gebraucht werden um ein Ziel zu erreichen. Nun ist die Frage ob diese Anzahl bei Churn gleich bleibt oder sich verändert. Die Parameter aus Tabelle 5.16 werden bei dieser Simulation verwendet.

Da das Netzwerk durch Churn kleiner wird, scheint auch die durchschnittliche Hopzahl kleiner zu werden (Tabelle 5.17), denn die ist um etwa einen Hop kleiner als ohne Churn. Das bedeutet mit Churn wird ein Ziel, wenn es erreicht wird, schneller erreicht. Doch dies ist nicht unbedingt erwünscht, denn durch eine kleinere Hopzahl wird auch die Anonymität geringer, da weniger Knoten die Nachrichten weiterleiten müssen.

| Churn | an |
|---|---|
| Inserts | 450 |
| Requests | 100 |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.16: Verhalten mit Churn – Parameter der Simulation zur Geschwindigkeit

| Gesendete Nachrichten: | 548 |
|---|---|
| Anzahl der Hops: | 2631 |
| durchschnittliche Hopzahl: | 4.8010 |

Tabelle 5.17: Verhalten mit Churn – Ergebnis der Simulation zur Geschwindigkeit

Inserts mit Churn werden hier nicht genauer betrachtet, da wir sie zum Teil am Anfang des Abschnitts betrachtet haben und zum anderen, für Inserts, keine Operationen gegen Churn implementiert sind.

### 5.3.3 Requests

In dieser Simulation soll ermittelt werden, auf wie viele Requests eine Antwort zurück kommt. Zu dem Zweck simulieren wir, wie meistens, ein Netzwerk mit 1000 Knoten. Von diesen 1000 Knoten werden 100 einen Request schicken. Hier muss jedoch beachtet werden, dass im Gegensatz zu den Insert Nachrichten, hier die Send und Forward Operationen, Churn entgegenwirken können.

| Churn | an ab 60 [min] |
|---|---|
| Inserts | 450 ab 31 [min] bis 60 [min] |
| Requests | 100 ab 90 [min] bis 110 [min] |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.18: Verhalten mit Churn – Parameter der Simulation zu Requests

Auch hier fällt das Ergebnis (Tabelle 5.19) schlechter aus, als bei der Simulation ohne Churn. Ohne Churn haben wir auf jeden Request, die angeforderten Dateien erhalten. Hier erhalten wir nur etwa Dreiviertel der Dateien. Jedoch sind auch 30% der Knoten offline, welche Dateien speichern können, die wir suchen. Außerdem sollten wir noch die Operationen Send und Forward besser einstellen. Hierbei lässt sich der Timeout, der Operationen, verbessern, denn möglicherweise werden einige Requests zu früh durch einen Timeout abgebrochen. Genau dies wollen wir im folgenden anpassen.

| | |
|---|---|
| Gesendete Requests: | 100 |
| Erfolgreiche Requests: | 74 |
| Fehlgeschlagene Requests | 26 |

Tabelle 5.19: Verhalten mit Churn – Ergebnis der Simulation zu Requests

### 5.3.4 Timeout der Operationen Send und Forward

Hat eine der Operation, Send und Forward, einen Timeout, so sendet der zugehörige Knoten eine Nachricht an den Ausgangsknoten, dass er keine Antwort mehr erwarten kann. Dadurch passiert etwas ähnliches wie ein Verwerfen der Nachricht und sie kommt nicht an. Doch die Frage ist wo kann außer bei Churn noch ein Timeout auftreten? Ein weiterer Timeout kann auftreten, wenn ein Knoten einen rejoin durchführt. Dann kann dieser Nachrichten empfangen, aber keine mehr senden bis er mit dem Rejoin fertig ist. Dabei bleibt die Nachricht einige Zeit bei diesem Knoten und wird nicht weitergeleitet. Wir werden eine Simulation durchführen um festzustellen, welche Zeit sich als Timeout eignet.

| | |
|---|---|
| Churn | an ab 60 [min] |
| Inserts | 450 ab 31 [min] bis 60 [min] |
| Requests | 100 ab 90 [min] bis 110 [min] |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.20: Verhalten mit Churn – Parameter der Simulation zu Timeouts der Operationen Send und Forward

In dieser Simulation verwenden wir die Parameter (Tabelle 5.20), die wir schon häufiger verwendet haben. Hierbei verändern wir nur den Wert für den Timeout der Send und Forward Operationen. Der Timeout der Forward-Operation hängt vom Timeout der Send Operation ab und muss deshalb nicht eingestellt werden.

Die Kurve aus Abbildung 5.7 zeigt, dass bei einem Timeout von 0 Sekunden auch keine Nachrichten ankommen. Dies war auch nicht anders zu erwarten, da der Timeout sofort ausgelöst wird. Darauf steigt die Kurve schnell an, was bedeutet, dass der Timeout nur wenig erhöht werden muss um bessere Ergebnisse zu erhalten. Jedoch scheinen auch die Möglichkeiten, den Timeout einstellen zu können beschränkt zu sein, da ein Maximum von 77 erreicht wird. Egal wie hoch der Timeout gewählt wird, an manche Dateien wird man nicht kommen, da 30% der Knoten mit den Daten offline gegangen sind. Somit reicht ein Timeout von 15 Sekunden. Dieser Wert erreicht auch das Maximum ist aber von der

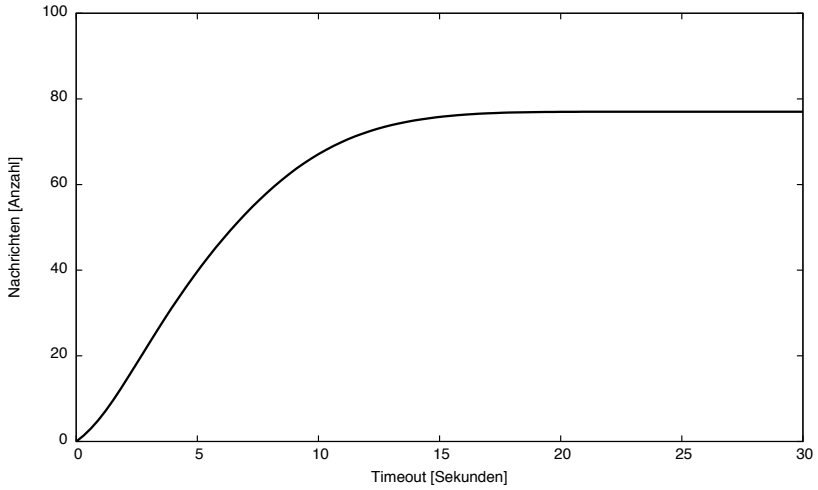



Abbildung 5.7: Verhalten mit Churn – Ergebnis der Simulation zu Timeouts der Operationen Send und Forward

Dauer der geringste Wert. In der Kurve ist auch zu sehen, dass in der Simulation zu Requests (Kapitel 5.3.3) mit einem Timeout von 12 Sekunden gearbeitet wurde, da hier der Wert der erfolgreichen Requests von 75 heraus kam und dieser Wert in der Kurve bei 12 Sekunden erreicht wird. Dieser Wert war damit eine recht gute Annahme, da der maximale Wert von 77 nur knapp unterschritten wurde.

### 5.3.5 Verfügbarkeit von Dateien

Bei Churn sind manche Knoten offline. Wird sich dies negativ auf die Verfügbakeit von Dateien auswirken? Diese Frage ist wichtig, da die Verfügbarkeit von Dateien ein wichtiger Punkt ist. Wenn Dateien nicht gefunden werden, so macht das Netzwerk keinen Sinn, da keine Informationen ausgetauscht werden können.

#### Wo sind die Dateien gespeichert?

In diesem Abschnitt betrachten wir, wie schon bei den Simulationen ohne Churn, die Verfügbarkeit einer häufig gesuchten Datei. Bei den Simulationen ohne Churn, kam heraus, dass 20% des Netzwerks die Datei erhalten haben, ohne aktiv danach gesucht zu haben. Die Parameter dieser Simulation haben sich nur der Churn-Parameter geändert.

In Abbildung 5.8 sind 3 Kurven zu erkennen. Eine Kurve beschreibt die Anzahl der Knoten, die online sind. Die zweite Kurve zeigt wie viele, von den Knoten die online sind, die Datei gespeichert

| Churn | an |
|---|---|
| Inserts | 500 |
| Requests | 500 ab 60 [min] bis 140 [min] |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 150 |

Tabelle 5.21: Verhalten ohne Churn – Parameter der Simulation zum Speicherort der Dateien

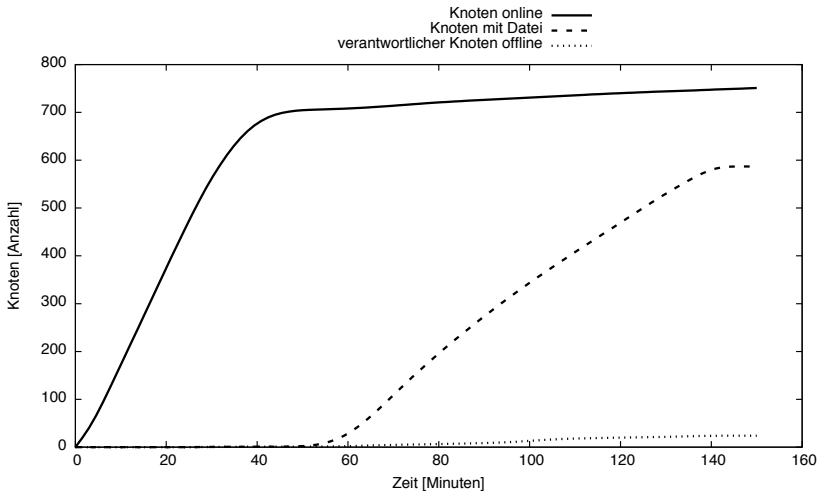



Abbildung 5.8: Verhalten mit Churn – Ergebnis der Simulation zum Speicherort der Dateien

haben. Dies sind immerhin noch fast 600 Knoten. Das bedeutet mit 30% Churn erhalten 10% der Knoten zusätzlich die Datei, ohne sie angefordert zu haben. Von den Knoten, die online sind, sind 600 Knoten etwa 80%. Damit ist der Wert sogar noch höher als, bei den Simulationen ohne Churn. Die dritte Kurve zeigt einen Sonderfall. Hier ist der Knoten offline, welcher die gesuchte Datei hat. Bevor er offline gegangen ist, hat ihn jedoch mindestens ein Request erreicht. Dadurch wurde die Datei an ein paar wenige Knoten weitergegeben. Nun haben viel weniger Knoten die Datei erhalten, sie ist jedoch nicht unerreichbar. Der Knoten ging etwas zu früh offline, denn wäre die Datei vorher mehrfach angefragt worden, so hätten danach mehr Knoten sie noch erhalten können. Somit bringt die Eigenschaft dieser Implementierung, dass “populäre“ Dateien sich gut im Netzwerk verbreitet unter Churn noch einen weiteren Vorteil. Dieser Vorteil ist, dass diese Datei selbst dann noch gut gefunden werden kann, wenn der Knoten, der für die Datei verantwortlich ist offline ist. Bei weniger häufig bis selten gesuchten Dateien funktioniert dies nicht und somit werden diese mit der Zeit aus dem Netzwerk verschwinden.

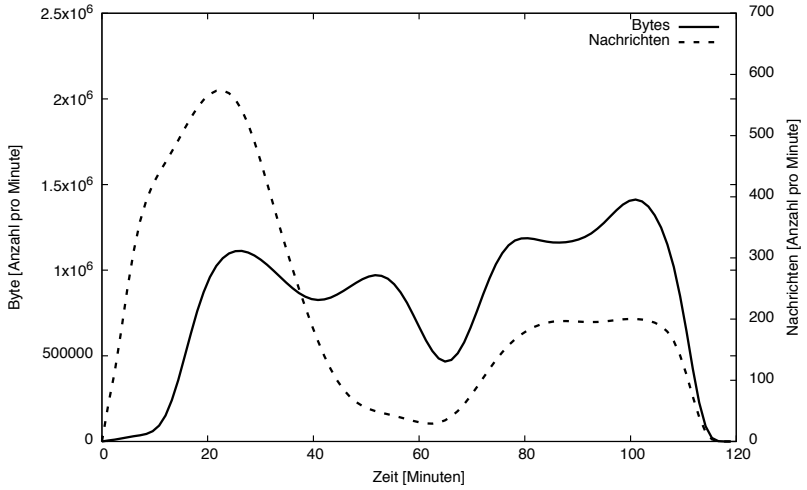



Abbildung 5.9: Verhalten mit Churn – Parameter der Simulation zu gesendeten Bytes pro Minute

### 5.3.6 Gesendete Bytes

Als letzte Simulation werden wir herausfinden, ob mit Churn mehr Nachrichten und mehr Bytes gesendet werden. Dazu werden wir die selbe Simulation ausführen, wie im Abschnitt ohne Churn, nur dieses mal mit Churn. Es werden wieder 1000 Knoten simuliert, von denen 500 einen Insert senden und 500 einen Request. 300 der Knoten, die einen Insert gesendet haben verwenden Churn und gehen während der Simulation offline.

| Churn | an |
|---|---|
| Inserts | 500 |
| Requests | 500 |
| Knoten | 1000 |
| Seeds | 10 |
| Dauer [min] | 120 |

Tabelle 5.22: Verhalten mit Churn – Parameter der Simulation zu gesendeten Bytes pro Minute

Das Ergebnis dieser Simulation (Abbildung 5.9) zeigt ein ähnliches Bild, wie bei der Simulation ohne Churn (Kapitel 5.2.5). Die Wege werden kürzer, deshalb sollten weniger Nachrichten verschickt werden, jedoch wirken dem die Operationen Send und Forward entgegen, da diese weitere Nachrichten schicken. Der Bytewert pro Minute für die Inserts ist nicht ganz so hoch, dafür dauert diese Phase länger.

In diesem Abschnitt haben wir gesehen, wie gut oder nicht gut diese Implementierung mit Churn

funktioniert. Einige Methoden müssen noch verbessert werden, damit Nachrichten auch bei Churn besser ankommen. Andere Methoden sind dagegen fast resistent gegen Churn und bieten einige Vorteile. Im nächsten Abschnitt werden wir diskutieren, wie anonym diese Implementierung ist und wie die Anonymität verbessert werden könnte.

## 5.4 Anonymität

In diesem Abschnitt wird gezeigt, dass die Implementierung in einem gewissen Maß Anonymität liefert. Die Anonymität kann jedoch noch weiter verbessert werden.

Bei den Simulationen ohne Churn haben wir gesehen, dass im Schnitt etwa 6 Hops benötigt werden um mit einer Nachricht ein Ziel zu erreichen. Da die Knoten gegenseitig nicht ihre Kontakte kennen, kann ein Kontakt eines Knotens mit jedem anderen Knoten des Netzwerks verbunden sein, was es bei zwei Hops schon unmöglich macht den Absender zu ermitteln. Dazu kommt, dass man nicht weiß, wie viele Knoten eine Nachricht schon durchlaufen hat, wenn sie einen Knoten erreicht. Selbst, wenn jemand einen großen Teil des Netzwerks unter Kontrolle hat, kann ein fremder Knoten nicht als Absender festgestellt werden. Deshalb funktioniert in dieser Implementierung die Anonymität aus dem FreeNet Paper. Jedoch müssen wir aufpassen, dass wir keine feste TTL wählen, sondern eine Variable oder den Ansatz mit Wahrscheinlichkeiten, da sonst die Anzahl der Hops bis zum jeweiligen Knoten bekannt ist.
Ein weiterer Punkt ist das Mitlesen von Nachrichten. Wir verwenden momentan keinerlei Verschlüsselung. Jeder, der eine Nachricht abfängt, kann diese Nachricht in Klartext mitlesen. Dadurch könnte möglicherweise der Absender von außerhalb des eigentlichen Netzwerks ermittelt werden. Doch dies ist ein Thema für zukünftige Arbeiten.

Abschließend ist in diesem Kapitel zusammenfassend zu sagen, dass diese Implementierung die Anonymität von FreeNet in gewissem Maß widerspiegelt. Außerdem kann man unter günstigen Bedingungen alle abgelegten Informantionen erhalten.

# Kapitel 6

# Fazit

In dieser Arbeit haben wir eine FreeNet ähnliche und auf dem FreeNet White Paper basierende Implementierung analysiert. Dabei haben wir festgestellt, dass diese Implementierung in Netzwerken, in denen immer alle Knoten online sind, sehr gut funktioniert. In Netzwerken, bei denen Knoten häufig offline sind wird diese Implementierung schlechter funktionieren, jedoch nicht vollkommen zusammenbrechen. Dabei scheint es, dass die im Kapitel zum Design beschriebenen Probleme gut durch die vorgestellten Lösungen behoben wurden. Wir konnten sehen, dass Daten, welche häufig abgefragt werden, schneller gefunden werden können, da sich diese Daten im Netzwerk ausbreiten. So bleiben die Informationen im Netzwerk aktuell und alte oder wenig angeforderte Daten verschwinden mit der Zeit aus dem Netzwerk. Dies ist genau das Prinzip der Nachrichten, denn dort zählen nur aktuelle Informationen oder Informationen, die viele Personen interessiert. Werden ein paar Verbesserungen an der Implementierung vorgenommen, sodass das Netzwerk unter Churn besser funktioniert, so stellt diese Implementierung ein anonymes Netzwerk zur Verfügung, über das (für den Absender) heikle Daten geteilt werden können. Damit unterliegt das Netzwerk der Kontrolle von Niemandem und niemand kann für das, was er denkt oder sagt, belangt werden. Ein Nachteil hierbei ist jedoch, das illegales Filesharing hierdurch auch nicht auffindbar wird. Damit können Urheberrechtsverletzungen nicht mehr verhindert oder geahndet werden. Doch zu bewerten was hier mehr wiegt, ist nicht meine Aufgabe, da sie ein rechtliches und kein technisches Problem ist.

# Kapitel 7

# Zukünftige Arbeiten

Während dieser Arbeit sind Themen aufgefallen, welche in zukünftigen Arbeiten betrachtet werden können. Dazu zählt ein anderer Join-Vorgang, der den hier beschriebenen Join-Vorgang verbessert oder einen kleine-Welt Graphen erzeugt und so Routing in diesem Netzwerk effizienter macht. Außerdem haben wir gesehen, dass beim Auftreten von Churn die Insert Nachrichten ihr Ziel nicht immer erreichen. Dies sollte auch überarbeitet werden, da es auch keine Rückmeldung gibt, wenn ein Insert erfolgreich war. Das letzte Thema beschäftigt sich mit der Sicherheit des Netzwerks. Es sollte überlegt werden, ob Verschlüsselung verwendet werden sollte und wenn ja, wie sollte dies umgesetzt werden. Des weiteren sollte ausgeschlossen werden können, dass sich ein Knoten mit allen anderen Knoten verbindet, da er damit nachvollziehen kann wer Nachrichten schickt.

## 7.1 Alternativer Join-Vorgang

Ein anderer Join-Vorgang sollte gefunden werden um Routing im Netzwerk effizienter zu machen. Dabei könnten 2 Ansätze verfolgt werden. Einmal ein Ansatz, der über eine Wahrscheinlichkeit funktioniert und ein anderer Ansatz, der ein kleine-Welt Netzwerk erzeugt.

### 7.1.1 Wahrscheinlichkeit

Momentan verbindet sich ein joinender Knoten mit dem Knoten, den er vom BootstrapManager bekommt (In Wirklichkeit ist dies der Freund, der benötigt wird um dem Darknet beizutreten) und den beiden anderen Knoten, welche über den Identifier ermittelt wurden. Doch wieso nicht auch mit einer bestimmten Wahrscheinlichkeit mit einem Knoten verbinden, der während des Join-Vorgangs genutzt wird.

### 7.1.2 kleine-Welt Netzwerk

Ein anderer Ansatz ist ein komplett verschiedener. Hier wird ein kleine-Welt Netzwerk [EK10] erzeugt. Ein Kleine-Welt Netzwerk, ist ein Netzwerk in dem jeder Knoten nur wenige Hops benötigt um eine Nachricht an einen anderen zu senden. Dies würde das Routing stark beschleunigen, doch muss dabei analysiert werden, ob die Anonymität dann noch gewährleistet werden kann.

## 7.2 Insert check

Wir haben gesehen, dass viele Insert Nachrichten, unter Churn, verloren gehen. Um dies zu beheben kann man zwei Ansätze auswerten. Einmal eine Nachricht an den Sender zurück senden oder vor dem Weitersenden der Insert Nachricht überprüfen, ob der andere Knoten online ist.

### 7.2.1 InsertReply

Dieser Ansatz ist äquivalent zu dem der Request Operationen Send und Forward. Hier wird auf das Ausfallen eines Knotens reagiert, da keine Antwortnachricht zurück kommt. Das selbe Prinzip ließe sich bei Inserts realisieren, indem eine Antwortnachricht auf einen Insert zurück gesendet wird. Dadurch sollten mehr Insert Nachrichten ankommen.

### 7.2.2 Ping-Pong vor Senden der Daten

Die Insert Nachrichten gehen verloren, wenn diese an einen Knoten geschickt werden, der offline ist. Doch dies lässt sich leicht beheben, indem vor dem Senden der Insert Nachricht ein Ping-Pong mit diesem Knoten ausgeführt wird. Ein Ping-Pong bedeutet der sendende Knoten sendet dem Empfangenden eine Ping-Nachricht und der empfangende Knoten sendet dem Sendenden eine Pong Nachricht. Erhält der sendende Knoten keine Pong-Nachricht, so muss er davon ausgehen, dass der empfangende Knoten offline ist und sucht einen anderen Knoten.

## 7.3 Sicherheit

Die Sicherheit von Daten ist in der letzten Zeit immer wichtiger geworden. In diesem Netzwerk möchte man zwar Daten teilen, aber möglicherweise nur mit anderen Teilnehmern aus dem Netzwerk. Des-

halb sollten die Nachrichten verschlüsselt werden. Außerdem sollte in diesem Netzwerk sichergestellt sein, dass niemand die Möglichkeit hat das Netzwerk so zu hacken, dass die Absenderanonymität gefährdet ist.

### 7.3.1 Verschlüsselung

Bei Verschlüsselungsverfahren verwendet man zwei unterschiedliche. Einmal die symmetrische und einmal die asymmetrische Verschlüsselung. Es wäre festzustellen welches Verfahren sich hier besser eignet.

#### Symmetrisch

Bei der symmetrischen Verschlüsselung ist allen Kommunikationspartnern, die die Nachricht ver- oder entschlüsseln dürfen, der gleiche Schlüssel bekannt. Dabei muss erforscht werden, ob alle Knoten diesen Schlüssel besitzen oder nur bestimmte Knoten, von denen man den Schlüssel erfahren kann. Des weiteren müssen diese Schlüssel laufend erneuert werden, damit dieser Schlüssel auch wirklich nur im Netzwerk bekannt ist. Denn sollte ein Knoten gehackt werden, könnte dieser Schlüssel bekannt werden und die Nachrichten können wieder von außerhalb gelesen werden. Abgesehen von den anderen Problemen die auftreten, wenn ein Knoten gehackt wird.

#### Asymmetrisch

Asymmetrische Verschlüsselung [DH76] basiert auf einem mathematischen Problem, das nicht effizient gelöst werden kann. Dabei gibt es zum verschlüsseln einen anderen Schlüssel, als zum entschlüsseln. Dieses Verschlüsselungsverfahren wird auch bei der e-mail Verschlüsselung PGP verwendet. In unserem Fall gäbe es, für jeden Knoten, eine Schlüssel, der nur ihm bekannt ist. Dazu kommt ein weiterer Schlüssel, welcher öffentlich ist und von allen Knoten des Netzwerks erhalten werden kann. Wie Knoten an diesen öffentlichen Schlüssel kommen müsste noch überlegt werden und wie dann die Verschlüsselung der Daten funktioniert auch, da auch jeder Knoten auf dem Rückweg die Daten speichern können soll.

### 7.3.2 Absenderanonymität

Es müssen Schwachstellen im Netzwerk erkannt werden, die es ermöglichen den Absender zu identifizieren. Sollte sich zum Beispiel ein Knoten mit allen anderen Knoten des Netzwerks verbinden, so weiß er, dass der Absenderknoten der Knoten ist, von dem er die Nachricht empfangen hat. Diese Möglichkeiten müssen ausgeschlossen werden, damit das Netzwerk seinen Sinn erfüllt.

# Literaturverzeichnis


[Cla99] CLARKE, I.: FreeNet White Paper. In: *Division of Informatics, University of Edinburgh* (1999).

[Cli02] CLIP2 - THE GNUTELLA DEVELOPER FORUM: *Gnutella*. http://rfc-gnutella.sourceforge.net/developer/stable/index.html, 2002.

[CSTV10] CLARKE, I.; SANDBERG, O.; TOSELAND, M.; VERENDEL, V.: Private Communication Through a Network of Trusted Connections: The Dark Freenet. In: *PET* (2010).

[DH76] DIFFIE, W.; HELLMAN, M. E.: New Directions in Cryptography. In: *IEEE Transactions on Information Theory* 22 (1976), Nr. 6, S. 644–654.

[EK10] EASLEY, D.; KLEINBERG, J.: The Small-World Phenomenon. In: *Networks, Crowds, and Markets: Reasoning about a Highly Connected World* (2010), S. 611–644.

[GP12] GRAPHSTREAM-PROJECT: *GraphStream*. http://graphstream-project.org, 2012.

[Gra11] GRAFFI, K.: PeerfactSim.KOM: A P2P System Simulator – Experiences and Lessons Learned. In: *IEEE P2P '11: Proc. of the Int. Conf. on Peer-to-Peer Computing*, 2011.

[MM02] MAYMOUNKOV, P.; MAZIÈRES, D.: Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In: *IPTPS '02: Proc. of the Int. Workshop on Peer-To-Peer Systems* Bd. 2429, Springer, 2002 (LNCS).

[RD01] ROWSTRON, A. I. T.; DRUSCHEL, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: *IFIP/ACM Middleware '01: Proc. of the Int. Conf. on Distributed Systems Platforms* Bd. 2218, Springer, 2001 (LNCS).

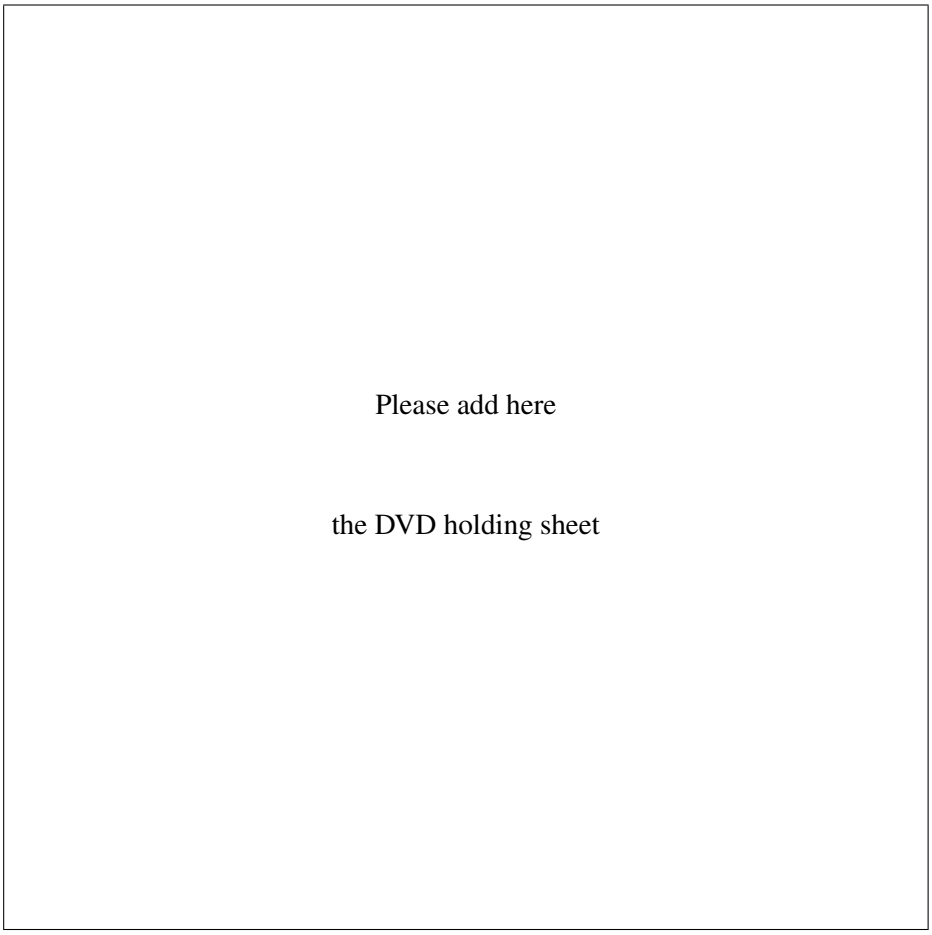
[SMK+01] STOICA, I.; MORRIS, R.; KARGER, D.; KAASHOEK, M. F.; BALAKRISHNAN, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: *SIGCOMM '01: Proc. of the Int. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM, 2001. ISBN 1–58113–411–8.

[Yil15] YILDRIM, E.: Untersuchung von Anonymisierungsverfahren in DHT basierten Overlays. In: *Bachelorarbeit* (2015), Juli.

# 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, 28.August 2015 Timm Kenfenheuer




**This DVD contains:**

- A *pdf* Version of this bachelor thesis
- All LaTeXand grafic files that have been used, as well as the corresponding scripts
- **[adapt]** The source code of the software that was created during the bachelor thesis
- **[adapt]** The measurment data that was created during the evaluation
- The referenced websites and papers