
# Implementierung und Evaluierung von Dunbar-basierten Techniken Sozialer Netzwerke

Bachelorarbeit

vorgelegt von

Marjan Basic

geboren in
Duisburg

eingereicht bei der

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

September 2015

Betreuer:
Tobias Amft M.Sc

# Abstract

„Soziales Netzwerk“ ist ein Begriff, den fast jeder kennt. Diese Netzwerke haben in unserem heutigen Leben einen hohen Stellenwert eingenommen. Einen Großteil unserer Zeit verbringen wir damit, unsere Freundschaften online zu pflegen und uns zu informieren, was unsere Freunde zur Zeit treiben. Private Gespräche werden Tag für Tag über diesen Weg der Kommunikation geführt, wodurch eine ziemlich großer Menge an Informationen täglich durch Soziale Netzwerke strömt.

Ein großes Problem dabei ist die Privatsphäre. Private Gespräche werden nicht lokal, sondern auf den Servern der Anbieter gespeichert. Somit sind unsere Nachrichten, Fotos, Videos oder sogar wichtige Dokumente, die in einer Nachricht verschickt werden, in fremden Händen. Man hat keine Kontrolle mehr über seine Daten und weiß nicht genau was mit ihnen gemacht wird.

Peer-to-Peer Netzwerke bieten eine gute Möglichkeit ein Soziales Netzwerk dezentral zu betreiben. Sie sind robust und skalieren bei großer Teilnehmerzahl ziemlich gut. Im Fokus dieser Arbeit stehen Dunbar-basierte Techniken, die in DOSNs (engl. Distributed Online Social Networks) verwendet werden können. Eine dieser Techniken ist ein Datenhaltungsservice, welcher mit befreundeten Knoten realisiert wird. In dieser Arbeit wird ein Ansatz aus [1] verwendet und ein Peer-to-Peer Overlay um einen Datenhaltungsservice erweitert. Dieser Service verwendet soziale Beziehungen als Speicherpunkte und bietet somit eine Möglichkeit, Daten (Profilinformationen, Fotos, Pinnwandeinträge, Nachrichten etc.) auf befreundeten Knoten zu replizieren, um trotz Abwesenheit vom Netzwerk, die Erreichbarkeit der Daten so gut wie möglich zu gewährleisten. Damit soll ein möglichst großer Anteil der Daten im Netzwerk dauerhaft erreichbar sein, trotz der Wahrscheinlichkeit, dass Knoten aus dem Netzwerk austreten oder ungewollt das Netzwerk verlassen.

Diese Arbeit beschäftigt sich mit der Implementierung von Dunbar-basierten Techniken in einem Overlay. Es wird zum einen die Idee des Datenhaltungsservices beschrieben und ein Entwurf zur Implementierung in einem P2P Simulator erstellt. Außerdem wird ein bekanntes Overlay als Basis verwendet und um den Datenhaltungsservice erweitert. Dieses modifizierte Overlay wird am Ende der Arbeit simuliert und ausgewertet. Das Ergebnis der Evaluation zeigt, dass der implementierte Datenhaltungsservice funktioniert. Es zeigt sich eine gute Erreichbarkeit der Datenobjekte und es werden akzeptable Ergebnisse erzielt (sogar unter Churn).

# Danksagung

Ich danke meiner Familie und meinen Freunden für ihre Unterstützung. Besonderen Dank gilt meiner Mutter für das leckere Essen, Helena Hagemeier, Victor Roth, Erzen Hyko und meinem Betreuer Tobias Amft für die gute Betreuung und die hilfreichen Tipps.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung

Soziale Netzwerke sind heutzutage kaum mehr wegzudenken. Sie begleiten uns Tag ein und Tag aus. Der Bekanntheitsgrad dieser Netzwerke hat in den letzten Jahren stark zugenommen. Viele von ihnen besitzen jedoch eine zentrale Infrastruktur. Hierbei können erhebliche Schäden verursacht und große Mengen an Daten abgegriffen werden, falls diese zentralen Systeme angegriffen werden. Geheimdienste haben durch diese Netzwerke zum Beispiel die Möglichkeiten, Nachrichtenkommunikationen zu verfolgen und viele wichtige Informationen (z.B. versendete Dokumente in privaten Nachrichten) abzugreifen. Nach dem NSA Skandal 2013 ist der Ruf nach mehr Privatsphäre groß und der Wunsch mehr Kontrolle über die eigenen Daten zu haben immer verbreiteter.

Das Betreiben eines DOSN (engl. Distributed Online Social Network) [2], welches eine dezentrale Infrastruktur verwendet, mindert die Probleme von zentralisierten OSNs (engl. Online Social Networks). Hierbei ist jeder Teilnehmer, der sich in einem Peer-to-Peer Netzwerk (im Folgenden: P2P Netzwerk) befindet, daran beteiligt, dieses System aufrecht zu erhalten. Damit die Daten der Knoten im Netzwerk erhalten bleiben, muss ein Datenhaltungsservice verwendet werden. Auf diese Weise können Freunde Profilinformationen, Bilder, Kommentare und Nachrichten einsehen und erhalten, obwohl der Besitzerknoten nicht mehr im Netzwerk ist.

In dieser Arbeit wird eine Idee für einen Datenhaltungsservice verwendet und ein Entwurf für eine Implementierung erstellt. Nach der Implementierung in einem Simulator wird diese ausgewertet.

## 1.1 Ziele der Arbeit

Die folgende Arbeit beschäftigt sich mit Dunbar-basierten Techniken Sozialer Netzwerke. Das Ziel ist diese Techniken in ein vorhandenes Overlay einzubauen bzw. dieses zu erweitern und in einem P2P Simulator zu evaluieren. Somit soll eine Grundlage für Distributed Online Social Networks, im

Folgenden: DOSN, geschaffen werden. Im Fokus der Arbeit stehen die Datenhaltung und die Frage, wie sich ein entsprechender Service in einen Simulator und in ein vorhandenes Overlay einbauen lässt. Ein hier verwendeter Ansatz basiert auf der Idee eines Datenhaltungsservices aus [1]. Diese Idee wird aufgenommen und ein Protokoll entworfen, welches dann in einem Simulator implementiert wird. Die nächsten Kapitel zeigen, wie hier vorgegangen wird und welche Entscheidungen getroffen werden, um dies zu realisieren.

## 1.2 Überblick der Arbeit

In Kapitel 2 werden einige Grundlagen und verwandte Arbeiten betrachtet. Des Weiteren wird in Kapitel 3 das Design und die Idee des Datenhaltungsservices besprochen und ein Entwurf für die Implementierung dargelegt. Darauf folgt in Kapitel 4 die Implementierung dieses Services in einem Simulator. Kapitel 5 zeigt die Evaluation des Overlays. Am Ende folgt in Kapitel 6 eine Zusammenfassung dieser Arbeit und ein Ausblick auf weitere mögliche Arbeiten.

# Kapitel 2

# Verwandte Arbeiten und Grundlagen

In diesem Kapitel wird ein kleiner Überblick zur Hashtabelle und Verteilter Hashtabelle in 2.1 gegeben. Des Weiteren steht in Abschnitt 2.2 Chord und dessen Funktionsweise im Fokus, da es in dieser Arbeit verwendet und um einen Datenhaltungsservice erweitert wurde. Am Ende des Kapitels in 2.3 wird noch auf die verwandte Arbeit DiDuSoNet [1] eingegangen.

## 2.1 Hashtabelle, Verteilte Hashtabelle

### Hashtabelle

Eine Hashtabelle ist eine Tabelle, in der Schlüssel-Werte-Paare abgelegt werden. Hier bildet ein Schlüssel (engl. key) auf einen Wert (engl. Value) ab. Um dann an einen bestimmten Wert in dieser Tabelle zu kommen, wird eine Berechnung, mittels einer Hashfunktion (z.B. SHA-1 [3]), durchgeführt. Mit dieser Funktion wird der Schlüssel gehasht und somit gelangt man direkt zur gesuchten Position und an den unter diesem Hashwert abgelegten Wert. Die Berechnung erfolgt in konstanter Zeit.

### DHT (Distributed Hash Table), Verteilte Hashtabelle

Eine Verteilte Hashtabelle (engl. *Distributed Hash Table*, im Folgenden: DHT) ist eine Tabelle, die aufgeteilt dezentral gespeichert wird. Hier sind Knoten für bestimmte Schlüsselintervalle/Schlüsselbereiche zuständig, die über alle Knoten verteilt werden. Diese übernehmen die Verwaltung der Einträge und geben die Werte zurück, die hinter dem Hash bei ihnen abgelegt sind, falls der gesuchte Schlüssel von ihnen verwaltet wird. Chord [4] implementiert z.B. eine solche DHT. Darüber hinaus gibt es noch andere DHT Implementierungen wie z.B. CAN [5], Pastry [6].

## 2.2 Chord und Funktionsweise

Chord [4] ist ein Protokoll, welches in P2P-Netzwerken als Overlay fungiert. Es wird in dieser Arbeit als Basis verwendet und erweitert. Dieses Protokoll nutzt eine Ring-Topologie, wobei hier ChordIDs nach ihrer Größe sortiert angeordnet werden. Diese ChordIDs können Werte von 0 bis $2^{m-1}$ annehmen. Das m in Chord sollte relativ groß gewählt werden, damit es keine Kollisionen gibt. Mit der Verwendung von SHA-1 [3] als Hashfunktion ist m z.B. auf 160 gesetzt, da diese Hashfunktionen einen 160 Bit langen Hashwert liefert. Ein Knoten in Chord besitzt also eine ChordID und ist für alle Schlüsselwerte bis zu seinem Vorgänger (*PredecessorID + 1*) verantwortlich.

### Lookup (Schlüsselsuche)

Eine Suche nach einem bestimmten Schlüssel erfolgt, indem der Konten seinen Nachfolgerknoten fragt, ob dieser für den gesuchten Schlüssel zuständig ist. Falls dem nicht so ist, wird diese Anfrage weitergeleitet zum nächsten Nachfolger (engl. *Successor*), der dann prüft, ob die Zuständigkeit bei ihm liegt. Die Anfrage wird dann entlang des Rings weitergegeben, bis der zuständige Knoten gefunden ist. Dies ist der erste Ansatz, der in [4] beschrieben wird. Um diese Suche zu beschleunigen wird eine sog. *Fingertabelle, Finger Table* eingebaut. Diese besitzt Fingerkontakte j, die für die Schlüssel (ChordID + $2^{j-1}$) mod $2^m$ zuständig sind. Das j steht hier für die in der Fingertabelle eingetragenen Fingerkontakte und nimmt Werte von $1 <= j <= m$ an, wobei m für die Anzahl der Kontakte in dieser Tabelle steht. Bevor man also eine Anfrage an den Nachfolger verschickt, schaut man in diese Fingertabelle und sucht sich den Knoten mit der größten ChordID aus, die noch vor dem gesuchten Schlüssel liegt. An diesen wird die Anfrage weitergeschickt. Der Zielknoten ist dann der Knoten, dessen Nachfolger für den gesuchten Schlüssel zuständig ist.

### Verwaltung des Chord-Rings

Damit der Chord-Ring aufrecht erhalten wird, obwohl Knoten das Netzwerk verlassen und wieder betreten, werden Methoden benötigt, die diese Mechanismen umsetzen. Dafür sorgen die Methoden *stabilize(), check_predecessor()* und *fix_fingers()*. Die *fix_fingers()* Methode aktualisiert in regelmäßigen Abständen die Fingertabelle und repariert Einträge, die nicht mehr aktuell sind. Die *stabilize()* Methode hingegen überprüft in periodischen Abständen, ob der Nachfolger von uns noch korrekt ist und ob sich in der Zwischenzeit nicht ein neuer Knoten vor uns im Chord-Ring eingereiht hat. Hier wird der Vorgänger unseres Nachfolgers überprüft und falls dieser sich geändert hat, als neuer Nachfolger bei uns gesetzt. Der neue Nachfolger wird von unserem ehemaligen Nachfolger dann benachrichtigt und setzt uns als Vorgänger. Weiter überprüft die Methode *check_predecessor*, ob unser Vorgänger noch zu erreichen ist. Falls dies nicht der Fall ist, können wir davon ausgehen, dass sich dieser nicht mehr im Netzwerk befindet und setzen den Zeiger zum Vorgänger auf *NULL*.

## 2.3 Datenhaltungsservice

### Keine Ausfallsicherheit bei Chord

Die Implementierung von Chord alleine garantiert keine Ausfallsicherheit der Daten. Geht ein Knoten offline, sind seine gespeicherten Daten für andere Knoten nicht mehr erreichbar. Eine mögliche Problemlösung dafür ist die Implementierung eines Datenhaltungsservices. Beim Datenhaltungsservice werden die Freunde eines Knotens verwendet, um die Daten des Besitzerknotens zu replizieren. Die Freunde, die ausgewählt werden die Daten eines Knotens zu speichern, nennt man *Point of Storage*, im Folgenden: *PoS*. Eine Idee dafür wird von Tobias Amft in [1] beschrieben. Sie wird in dieser Arbeit verwendet, um Chord zu erweitern und somit eine Daten-Replikation der Besitzerknoten zu erreichen. Die Verwendung von Freunden als Speicherorte basiert auf der Idee aus [1].

### DiDuSoNet

In der wissenschaftlichen Arbeit zu DiDuSoNet [1] wurde herausgefunden, dass durch eine Auswahl von zwei PoSs eine Erreichbarkeit der Profile bei ca. 95% liegt. Für die Implementierung im Simulator PeerfactSIM.KOM [7] hat man das Overlay Protokoll Pastry [6] verwendet. Hier wird hingegen Chord als Basis verwendet, um evtl. Unterschiede zu identifizieren. Die Arbeit [1] beruht auf Robin Dunbars Ansatz [8], [9], [10], dass Menschen sich ca. 150 menschliche Beziehungen untereinander merken und verwalten können. Diese Zahl wird auch Dunbar-Zahl genannt.

# Kapitel 3

# Design und Idee

In diesem Kapitel wird die Idee des Datenhaltungsservices in 3.1 und in 3.2 behandelt. Darüber hinaus wird ein Entwurf des Protokolls in Abschnitt 3.3 beschrieben. Dieses gibt eine Übersicht über die zu implementierenden Operationen im Simulator.

## 3.1 Elemente des Datenhaltungsservices

Im Folgenden Abschnitt werden die wichtigen Elemente diese Datenhaltungsservices beschrieben. Es muss ein Vorgehen geben, wie Freunde Im Netzwerk ausfindig gemacht werden und wie eine Datenabfrage abläuft. Außerdem muss geklärt werden, wie die Speicherpunkte ausgewählt werden und wie Freunde wissen, woher sie Daten ihrer anderen Freunde beziehen.

### Suchvorgang der Freunde mit einer bekannten UserID

Es wird vorausgesetzt, dass jeder Knoten im Netzwerk die *UserIDs* seiner Freunde kennt. Durch den Hash der *UserID* kommen wir an die PoS-Liste, in der auch die *OverlayID* des Besitzerknotes enthalten ist. Dadurch erhalten wir mit einem weiteren *Lookup* in der DHT die Kontaktadresse (IP-Adresse), um unseren Freund direkt kontaktieren zu können.

### Anfragen von Datenobjekten eines Freundes

Bei Datenanfragen kontaktieren Knoten die PoSs, die für einen Besitzerknoten die Datenobjekte verwalten. Die Daten werden dann mit einer Punkt-zu-Punkt-Verbindung übertragen. An dieser Stelle lassen sich noch Sicherheitsmechanismen einbauen, so dass wirklich gewährleistet ist, dass es sich hier um einen Freund des Besitzerknotens, und nicht um einen böswilligen Knoten, handelt. Hier könnte man Authentifizierungsmethoden, wie z.B. Public Key Authentifizierung verwenden, um zu ermitteln, ob es wirklich ein Freund ist.

### Auswahl der Point of Storages (PoSs)

Die Auswahl der Speicherorte wird durch die sozialen Beziehungen beeinflusst, welche der Besitzerknoten pflegt. Als Speicherorte für die Daten kommen somit nur Knoten infrage, die der Besitzerknoten persönlich kennt, also mit denen er befreundet ist. Dies hat zum Vorteil, dass Daten dort gespeichert werden, wo ein gewisses Vertrauensverhältnis herrscht. Andere Ansätze wie z.B. OceanStore [11] speichern die Daten auf Knoten, mit denen man gar kein Beziehungsverhältnis hat. Hier ist es sehr unsicher, was im Endeffekt mit den Daten geschieht.

### PoS-Liste

Eine DHT (Chord) wird hier verwendet, um die PoS-Liste zu propagieren. Somit weiß jeder Knoten, wo er die Daten seiner Freunde finden kann. Jeder Knoten hasht dafür seine eigene UserID, die seinen Freunden bekannt ist, und legt hinter diesem Hash die PoS-Liste in der DHT ab. Somit kann jeder befreundete Knoten, der die UserID kennt, auf die PoS-Liste bzw. auf die darin enthaltene OverlayID zugreifen. Dadurch kann z.B. ein direkter Kontakt mit Nachrichtenkommunikation gestartet oder es können Daten angefordert werden (wie z.B. Profildaten, Fotos etc.).

## 3.2 Idee des Datenhaltungsservices

Die für diese Arbeit verwendete Idee des Datenhaltungsservices kommt von Tobias Amft und wurde in [1] veröffentlicht. Hierbei macht man sich die Freundschaftsbeziehungen zu seinen Mitmenschen, die man kennt, zu nutzen. Ziel ist es mit dem Datenhaltungsservice immer zwei Point of Storages (PoSs) zu haben, auf denen Besitzerknoten ihre replizierten Daten ablegen können. Es wird versucht zwei dezentrale Punkte zu verwenden, um seine Daten (Profilinformationen, Bilder, Kommentare etc.) dort zu speichern. Damit besteht für befreundete Knoten die Möglichkeit an diese Informationen zu gelangen, unabhängig davon, ob man zu diesem Zeitpunkt online ist.

Die Abbildungen im Folgenden zeigen, welche Situationen hier auftreten können.

Es gibt hier verschiedene Fälle, die auftreten können. Fall 1 ist der Fall, in dem ein PoS gewollt das Netzwerk verlässt und dem anderen PoS eine Benachrichtigung schickt (s. Abbildung 3.1). Fall 2 (s. Abbildung 3.2) tritt auf, wenn der Knoten das Netzwerk unfreiwillig, also ohne Benachrichtigung an den anderen PoS, unerwartet verlässt. Fall 3 (s. Abbildung 3.3) findet statt, wenn zufällig beide PoSs das Netzwerk ungewollt verlassen und dadurch keinen Knoten durch eine Benachrichtigung informieren können.

Auf den unteren vier Abbildungen 3.4, 3.5 wird dargestellt, wie man die Konsistenz der Daten erhält.


(a) n wählt A als PoS.

(b) A teilt n mit, dass er das Netzwerk verlässt. B wird als neuer PoS gewählt.

Abbildung 3.1: Auswahl eines 2. PoS und gewolltes Verlassen des Netzwerks. [1]


(a) n verlässt das Netzwerk ungewollt.

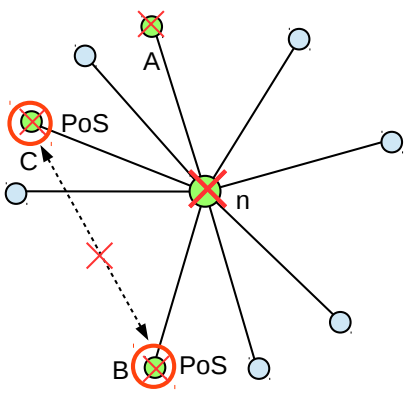
(b) B wählt neuen PoS für n

Abbildung 3.2: Fall 2, der hier dargestellt wird. [1]

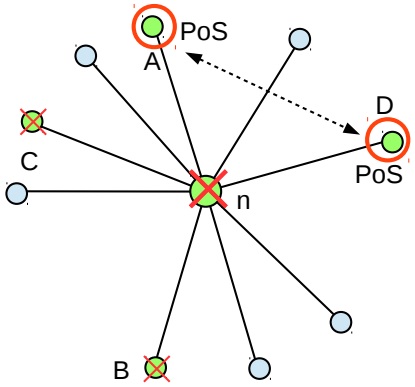
Die erste Abbildung auf 3.4 zeigt, wie Knoten *n*, der gerade Daten verändert hat, ein Update der Daten durchführt, indem er Knoten *A* die aktualisierten Daten zuschickt, da dieser zu diesem Zeitpunkt ein PoS ist.

Falls Fall 3 auftreten sollte, also der Fall, bei dem beide PoSs willkürlich ausfallen (s. Abbildung 3.4), sind nun beide PoSs nicht mehr erreichbar. Ein Knoten, der vorher für *n* schon einmal PoS war, tritt dem Netzwerk wieder bei und merkt, dass beide PoSs offline sind, indem er in die aktuelle PoS-Liste schaut und überprüft, ob die eingetragenen PoSs aktiv sind. Er setzt sich wieder als ersten PoS und wählt danach aus *n*'s Freundesliste einen neuen zweiten PoS aus (s. Abbildung 3.5).

Wenn jetzt Knoten *n* dem Netzwerk wieder beitritt, merkt dieser, dass die Daten auf den PoSs veraltet



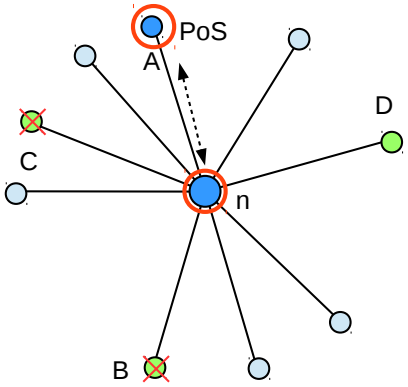

(a) B und C fallen ungewollt gleichzeitig aus.



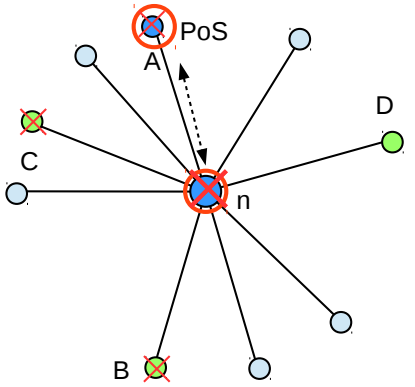

(b) Ein ehemaliger PoS betritt erneut das Netzwerk setzt sich als PoS und wählt einen 2. aus.

Abbildung 3.3: Fall 3, in dem beide PoSs ungewollt ausfallen. [1]

sind und veranlasst eine Datenaktualisierung, womit die Datenobjekte bei den PoSs aktualisiert werden. Dieser beschriebene Vorgang soll dafür sorgen, dass Daten auf den PoSs konsistent bleiben.

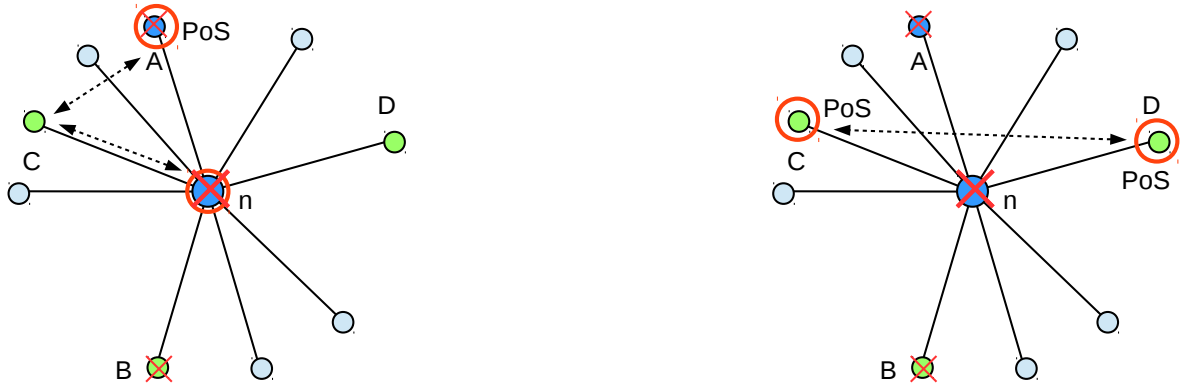



(a) n hat seine Datenobjekte verändert und sendet aktualisierte Daten an PoS A.




(b) Fall 3 tritt auf, beide PoSs fallen aus.

Abbildung 3.4: Datenaktualisierungsprozess und Ausfall beider PoSs. [1]




(a) Ehemaliger PoS C ist wieder im Netzwerk und merkt das beide PoSs offline sind.

(b) C setzt sich als ersten PoS und wählt D als zweiten PoS.

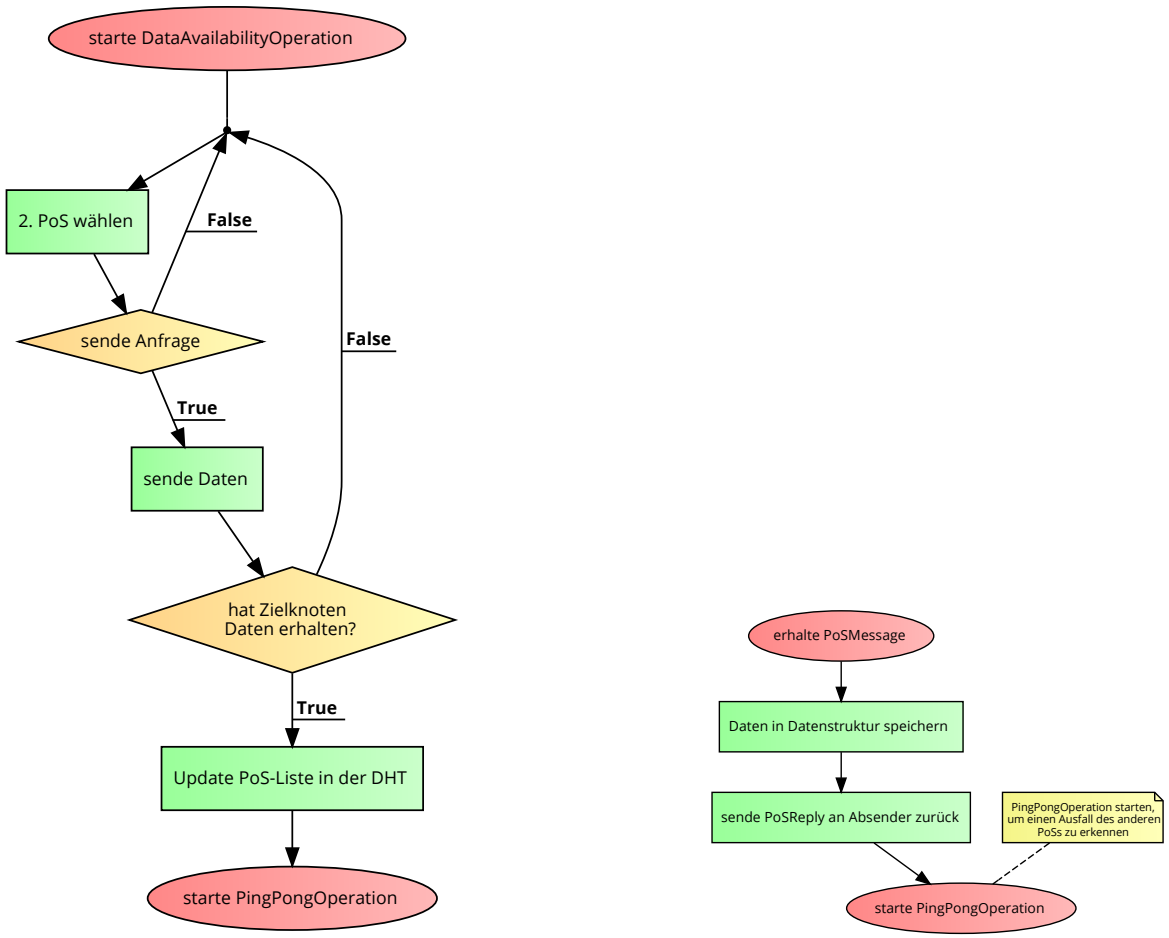
Abbildung 3.5: Ehemaliger PoS tritt dem Netzwerk wieder bei und prüft die PoSs. [1]

## 3.3 Entwurf des Protokolls

### 3.3.1 Datenhaltung

Wie auf Abbildung 3.6 wird nun der Entwurf des Protokolls erklärt. Um einen Überblick zu erhalten, werden einige Flowcharts verwendet, um dies zu illustrieren. Nach dem Start der Datenhaltungsoperation wählt diese einen zweiten PoS aus der Freundesliste aus und schickt, nach erfolgreicher Kontaktaufnahme diesem gewählten PoS relevante Datenobjekte zu (Fotos, Pinnwandeinträge, Kommentare etc.). Zudem soll die aktuelle Freundesliste des Besitzerknotens versendet werden. Wenn der Zielknoten die Daten bzw. die Anfrage als zweiten PoS erhalten hat, wird die PoS-Liste mit dem neuen PoS in der DHT aktualisiert. Danach wird eine PingPong-Operation zwischen diesen beiden PoSs gestartet, um mögliche Ausfälle zu erkennen. Diese Operation wird in Abbildung 3.7 skizziert. Falls eine Kontaktaufnahme zum auserwählten PoS nicht möglich ist, wird erneut zufällig ein neuer Freund aus der Freundesliste gewählt.

Der Zielknoten, der als PoS ausgewählt wurde, erhält eine Nachricht mit den Datenobjekten. Diese Datenobjekte werden in einer Datenstruktur auf dem Zielknoten gespeichert. Zudem muss eine Antwort an den Absender zurückgesendet werden, damit dieser ein Update der PoS-Liste veranlassen kann. Danach wird auch vom gewählten PoS eine PingPong-Operation gestartet, um Ausfälle zu erkennen.




(a) Vorgehen Datehaltungsoperation

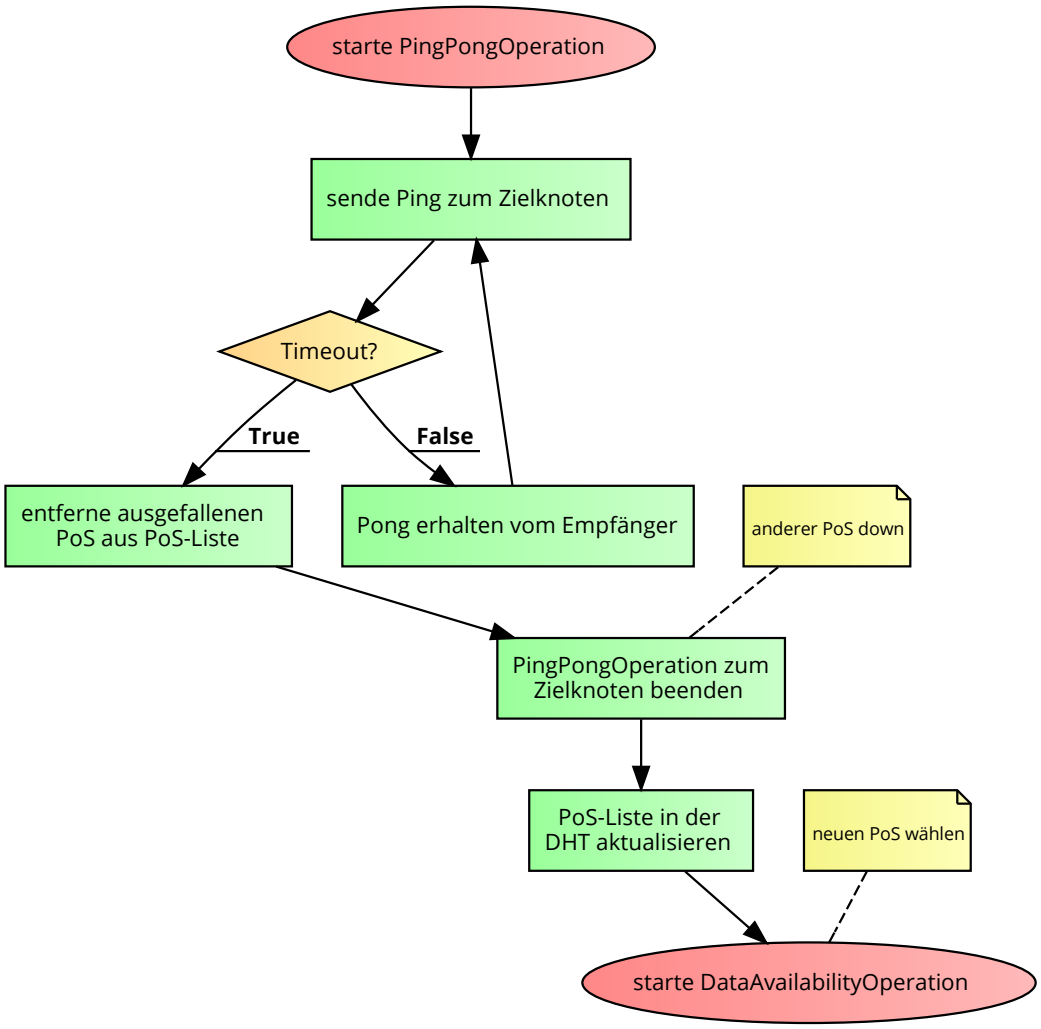
(b) Behandlung einer Anfrage als PoS

Abbildung 3.6: Datenhaltungsoperation und Vorgehen bei Erhalt einer Anfrage als PoS

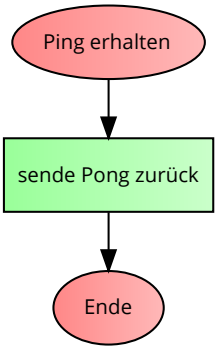
### 3.3.2 Überprüfung der Erreichbarkeit des anderen PoSs

Die PingPongOperation (s. Abbildung 3.7) soll von PoSs ausgeführt werden. Es wird zuerst eine Ping-Nachricht zum anderen PoS gesendet. Falls eine Pong-Nachricht erhalten wird, wird dieser Vorgang wiederholt, indem eine Ping-Nachricht nach einem bestimmten Zeitintervall erneut versendet wird. Falls ein Timeout auftritt, da keine Pong-Nachricht zurückgekommen ist, wird angenommen, dass der andere PoS sich nicht mehr im Netzwerk befindet. Hiernach wird der ausgefallene PoS aus der PoS-Liste ausgetragen. Des Weiteren erfolgt ein Update der PoS-Liste und die DataAvailabilityOperation wird angestoßen. Diese sorgt dafür, dass ein neuer zweiter PoS aus der Freundesliste zufällig ausgewählt wird.

Trifft die Ping-Nachricht beim Zielknoten ein, sendet dieser eine Pong-Nachricht zurück, um zu signalisieren, dass er sich noch im Netzwerk befindet und aktiv ist (s. Abbbildung 3.7).




(a) Vorgehen um Ausfälle von PoSs zu erkennen



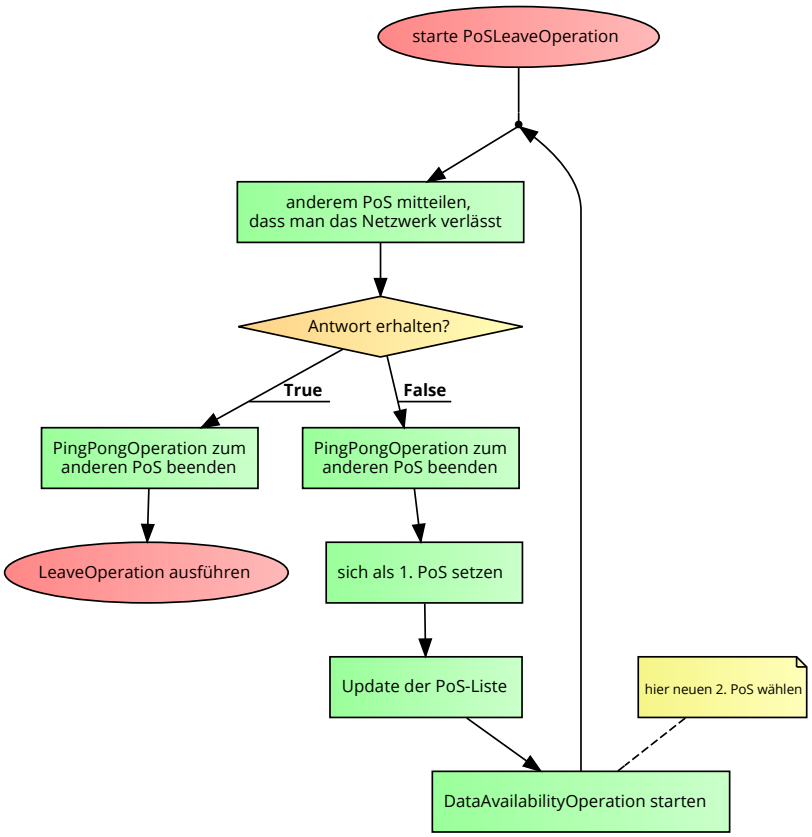

(b) Behandlung einer Ping Nachricht

Abbildung 3.7: PingPongOperation und Vorgehen bei Erhalt einer Ping Nachricht

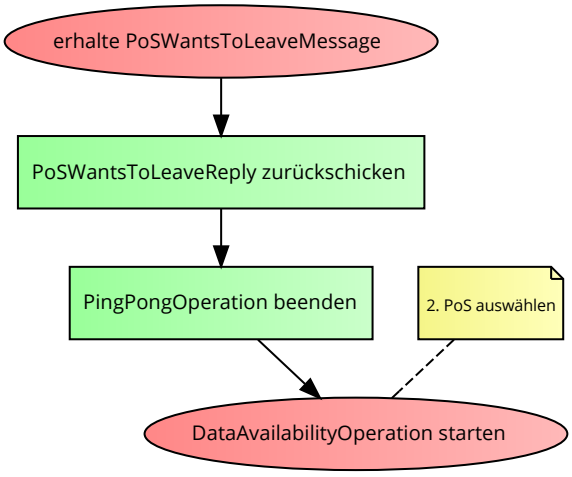
### 3.3.3 Gewolltes Verlassen des Netzwerks

Will ein Knoten das Netzwerk gewollt verlassen, benachrichtigt er den anderen PoS. Dieser erhält eine Nachricht (PoSWantsToLeaveMessage) und schickt dem Knoten, der das Netzwerk verlassen möchte eine Antwort zurück (s. Abbildung 3.8). Zudem wird die PingPongOperation beendet und danach direkt die DataAvailabilityOperation gestartet, die dann einen zweiten PoS auswählt. Der „Leaver“ erhält diese Antwort und beendet ebenfalls die PingPongOperation (s. Abbildung 3.8). Danach folgt das Verlassen des Netzwerks mit der LeaveOperation.

Falls der „Leaver“ keine Antwort erhält, könnte man jetzt darauf warten, dass die PingPongOperation den Ausfall bemerkt und einen neuen PoS auswählt. Alternativ kann man hier aber auch eine Prozedur starten, die die PingPongOperation beendet, sich danach als ersten PoS setzt, die PoS-Liste in der DHT aktualisiert, dann die DataAvailabilityOperation startet und dem neuen PoS mitteilt das Netzwerk verlassen zu wollen.




(a) Vorgehen beim Verlassen des Netzwerk mit Information an die PoSs



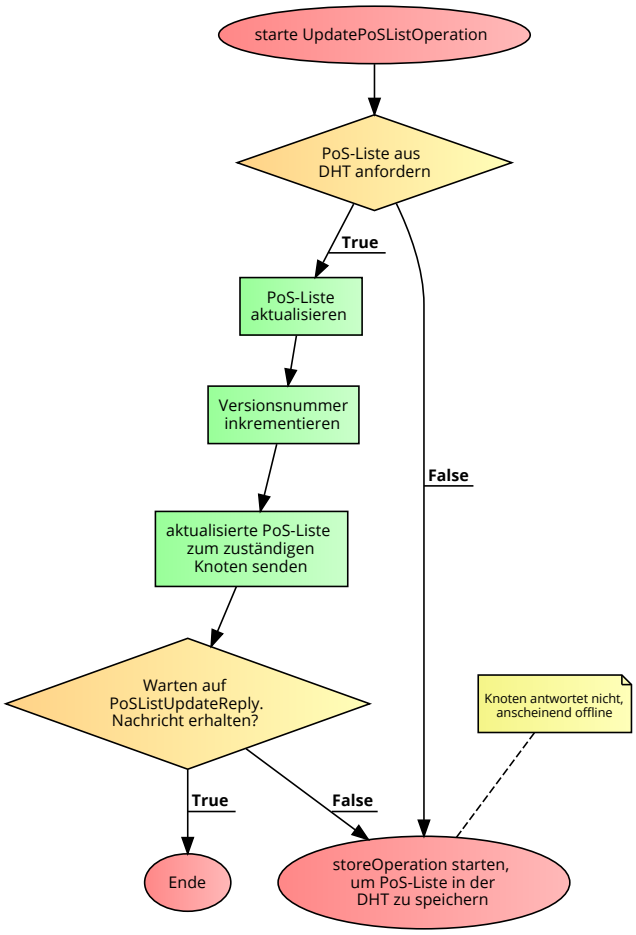

(b) Behandlung einer PoSWantsToLeaveMessage

Abbildung 3.8: Flowchart zur Operation, falls ein Knoten gewollt das Netzwerk verlassen möchte und Vorgehen beim Eintreffen einer solchen Nachricht

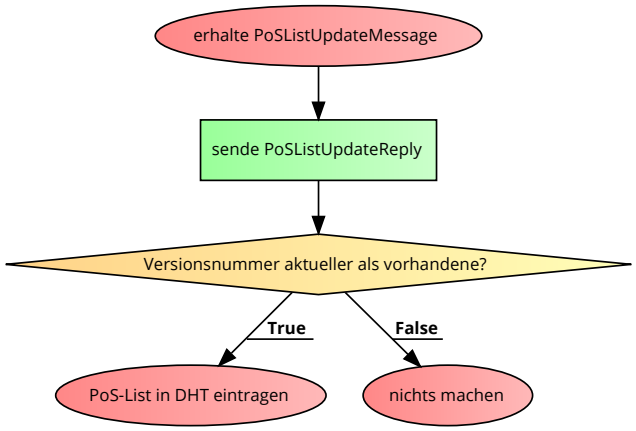
### 3.3.4 Aktualisierung der PoS-Liste in der DHT

Der Vorgang der Aktualisierung (s. Abbbildung 3.9) der PoS-Liste in der DHT startet damit, dass zuerst die PoS-Liste aus der DHT angefordert wird. Bei Erhalt folgt eine Aktualisierung der PoS-Liste, zudem muss die Versionsnummer inkrementiert werden. Danach wird die Liste dem zuständigen Knoten zugesendet und auf eine Antwort (PoSUpdateReply) gewartet. Kommt die Antwort, wissen wir, dass die Aktualisierung erfolgreich war. Kommt sie nicht, scheint der zuständige Knoten, der diesen DHT-Eintrag verwaltet, nicht erreichbar zu sein. Hier starten wir einfach eine storeOperation, um die PoS-Liste hinter der UserID in der DHT zu speichern.

Bei Erhalt der Update-Nachricht (3.9) sendet der Zielknoten eine Antwort (PoSUpdateReply) zurück, um mitzuteilen, dass der Vorgang erfolgreich war. Außerdem wird die Versionsnummer mit dem aktuellen DHT-Eintrag verglichen und, falls die empfangene PoS-Liste aktueller ist, in die DHT eingetragen. Andernfalls soll nichts gemacht werden.




(a) Vorgang zu Aktualisierung der PoS-Liste in der DHT




(b) Vorgehen bei Erhalt einer PoS-List Update-Nachricht

Abbildung 3.9: Entwurf für die UpdatePoSListOperation und Vorgehen beim Eintreffen einer Update-Nachricht

### 3.3.5 Aktualisierung der Datenobjekte auf den PoSs

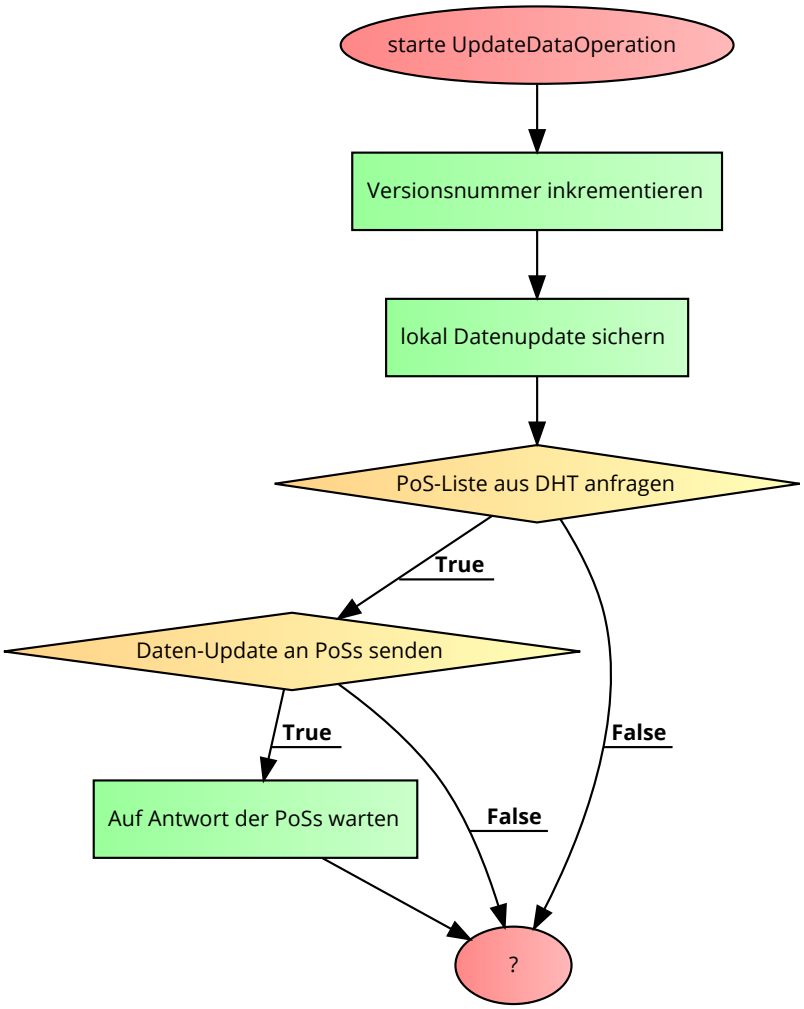
Verändert ein Besitzerknoten seine Daten, müssen diese an die PoSs weitergegeben werden, damit die Datenkonsistenz erhalten bleibt. Hierfür wird ein Vorgehen benötigt, das dafür sorgt, dass die PoSs diese aktualisierten Datenobjekte erhalten. In Abbildung 3.10 ist zu sehen, wie ein entsprechendes Protokoll funktionieren könnte. Nach dem Ändern der Datenobjekte muss die Version angepasst und die Daten lokal gesichert werden. Um an die aktuellen PoSs zu kommen wird die PoS-Liste aus der DHT angefragt und bei Erfolg den PoSs zugeschickt.

Bei Erhalt des Daten-Updates (s. Abbildung 3.10) wird die Versionsnummer mit den erhaltenen Datenobjekten und den lokal vorhandenen Datenobjekten verglichen und, falls die erhaltenen aktueller sind, lokal gespeichert und eine Antwort zurückgesendet. Falls veraltete Datenobjekte gesendet wurden, soll nichts geschehen.

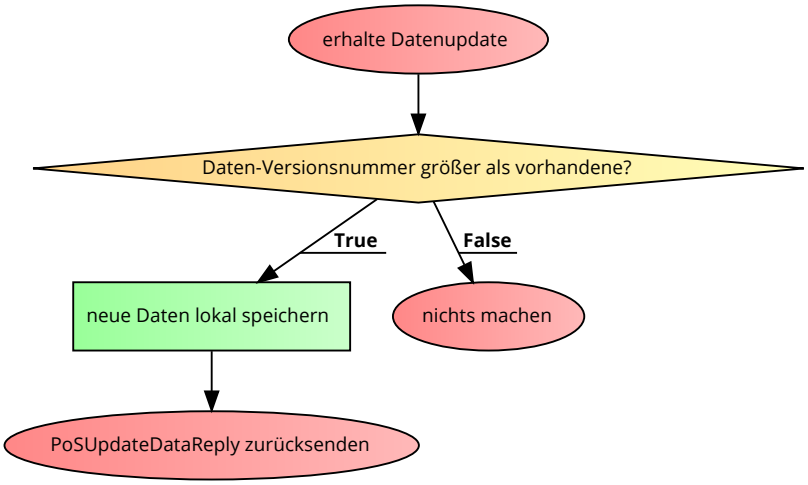
### 3.3.6 Erneutes Betreten des Netzwerks

Der Fall, in dem ein Knoten das Netzwerk erneut betritt, wird in den beiden Abbildungen (s. Abbildung 3.11) dargestellt. Kommt ein Knoten wieder ins Netzwerk, prüft dieser erst mal, ob er Daten besitzt. Falls nicht, müssen keine weiteren Aktionen erfolgen. Falls doch, so wird die Operation checkOldDataFromOwnerNodes (s. Abbildung 3.12) ausgeführt. Danach wird noch überprüft, ob alle Daten, die die PoSs besitzen, aktuell sind. Dies geschieht so, dass eine Operation den PoSs eine Nachricht, mit der aktuell vorhandenen Version der Daten, sendet und diese eine entsprechende Antwort zurücksenden. Falls die PoSs alte Daten verwalten, werden diese dann aktualisiert.

Mit der Operation checkOldDataFromOwnerNodes werden alle PoS-Listen der Knoten angefragt, von denen man alte Daten besitzt (s. Abbildung 3.12). Falls diese Anfrage erfolgreich war wird überprüft, welche Größe die PoS-Liste hat. Hier gibt es drei Fälle, die auftreten können: Die PoS-Liste ist leer, also Größe 0, besitzt Größe 1, oder Größe 2. Wenn Kontakte in der PoS-Liste eingetragen sind, wird danach überprüft, ob diese Liste veraltet ist. Es kann vorkommen, dass man selbst in der Liste steht, da man vor kurzem vom Netzwerk getrennt wurde. Hier kann es sein, dass der andere PoS es noch nicht gemerkt hat oder aber auch ausgefallen ist. Falls er auch ausgefallen ist, setzt man sich als ersten PoS, führt ein Update der PoS-Liste in der DHT durch und startet die DataAvailabiltyOperation, um einen zweiten PoS zu wählen. Ist die PoS-Liste nicht veraltet und der PoS ist auch online kann es evtl. kurz dauern bis ein zweiter PoS von diesem gewählt wird. Danach werden die Datenversionen, welche die PoSs besitzen, überprüft und falls veraltet, die aktuellen Datenobjekte gesendet.




(a) Entwurf einer Operation um Daten auf PoSs zu aktualisieren




(b) Vorgehen beim Erhalt einer Aktualisierungsnachricht

Abbildung 3.10: Operation um Daten auf PoSs zu aktualisieren und Vorgehen beim Erhalt einer Update-Nachricht

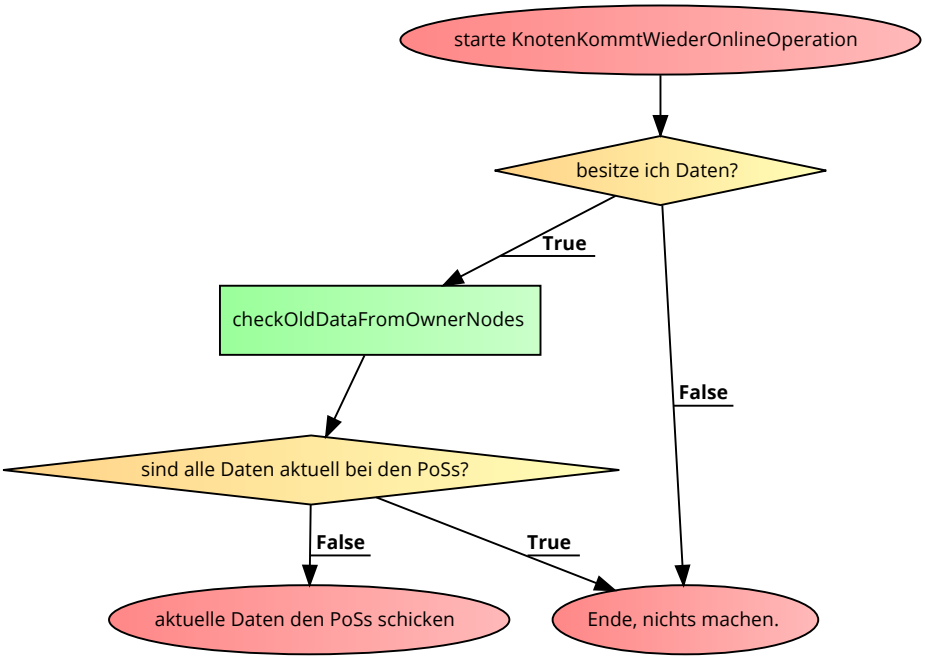



Abbildung 3.11: Entwurf für eine Operation, falls ein Knoten wieder dem Netzwerk betritt

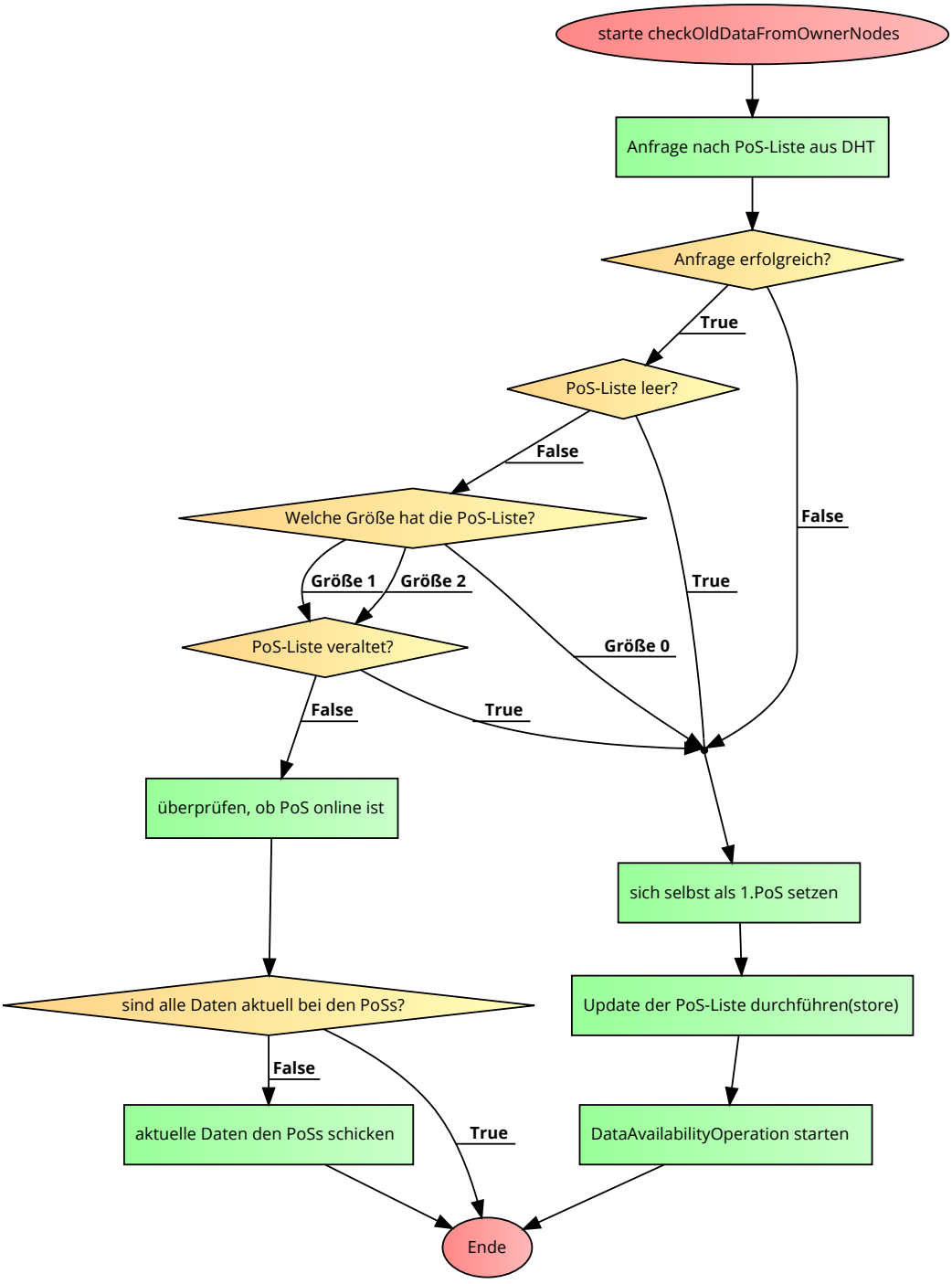



Abbildung 3.12: Vorgehen, falls man Daten von befreundeten Knoten besitzt

# Kapitel 4

# Implementierung des Datenhaltungsservices

Die Idee des Datenhaltungsservices wurde in Kapitel 3 beschrieben und das zu implementierende Protokoll mittels Flowcharts skizziert. Im Folgenden wird die konkrete Umsetzung im Simulator besprochen. In Abschnitt 4.1 wird ein wenig auf den verwendeten Simulator *PeerfactSIM.KOM* [7] eingegangen. Des Weiteren wird in 4.2 beschrieben, wie die Implementierung von *Chord* [4] aus dem Simulator erweitert und angepasst wird, um den Dunbar-basierten Datenhaltungsservice zu realisieren.

## 4.1 P2P Simulator PeerfactSIM.KOM

Als Simulator in dieser Arbeit wurde *PeerfactSIM.KOM* [7] verwendet. Es handelt sich dabei um einen eventbasierten Simulator, der in der Programmiersprache Java entwickelt wurde und an dem noch heute gearbeitet wird. Er wurde 2005 von der TU Darmstadt entwickelt und seit 2011 von der Universität Paderborn und der Universität Düsseldorf erweitert. Mit dem Simulator hat man die Möglichkeit *P2P* Netzwerke zu simulieren und auszuwerten. Es besteht auch die Möglichkeit eigene Overlays zu entwickeln oder vorhandene zu erweitern und diese im Simulator zu testen. Außerdem lassen sich diese mit Visualisierungen darstellen. Der Simulator besitzt eine Vielzahl von schon implementierten P2P Overlays wie z.B. CAN [5], Chord [4] und Pastry [6].

## 4.2 Implementierung

Es folgt nun der Teil der Implementierung. Das Design und die Idee aus Kapitel 3 werden nun verwendet und in *PeerfactSIM.KOM* [7] implementiert. Einen Überblick über die Pakete des DunbChord

Overlays folgen in Abschnitt 4.2.1. Weiter werden die im Rahmen dieser Arbeit modifizierten und erweiterten Module beschrieben.

### 4.2.1 Überblick und Erweiterungen der Pakete von DunbChord

Die Klassen der Chord-Erweiterung für das Dunbar-basierte Overlay mit dem Datenhaltungsservice sind im Paket *org.peerfact.impl.overlay.dht.chord.dunbchord* enthalten. Die Unterteilung der Pakete sehen folgendermaßen aus:

- *org.peerfact.impl.overlay.dht.chord.dunbchord.components* - hier befinden sich einige Komponenten, wie z.B. die Klasse ChordNode, ChordContact, DunbarChordAnalyzer, FriendList, NodeGraphReader, DataObjects etc.

- *org.peerfact.impl.overlay.dht.chord.dunbchord.messages* - in diesem Paket liegen die Nachrichten, die für die Kommunikation zwischen den Knoten versendet werden, wie z.B. CheckDataMessage, CheckOnlineMessage, PingMessage, PoSMessage etc.

- *org.peerfact.impl.overlay.dht.chord.dunbchord.operations* - im dritten Paket sind die wichtigen Operationen, die dafür sorgen, dass das Overlay richtig funktioniert wie z.B. DataAvailabilityOperation, GetOperation, UpdateDataOperation, UpdatePosListOperation etc.

Weitere verwendete Klassen und Module sind in *org.peerfact.impl.overlay.dht.chord.base* enthalten, da jede Chord-Implementierung auf diese Basismodule in *PeerfactSIM.KOM* zugreift und sie verwendet. Außerdem wurden die Klassen CheckPredecessorOperation, UpdateDirectSuccessorOperation, JoinOperation, CheckSuccessorOperation zur besseren Übersicht in das Paket *org.peerfact.impl.overlay.dht.chord.dunbchord.operations* verschoben.

**Erweiterungen der Klasse ChordNode.java**

Diese Klasse repräsentiert einen Knoten im *DunbChord-Overlay* und beinhaltet wichtige Funktionen. Die Originalklasse wurde um viele Flags, Parameter, Datenstrukturen zur Verwaltung des Datenhaltungsservices, Testmethoden, Speicherung und Verwaltung der Freundesliste, Speicherung vieler Informationen wie z.B. für welche UserIDs man zur Zeit ein PoS ist, Speicherung eigener Datenobjekte und Datenobjekte von Freunden, Methoden zum Starten der Operationen für die Datenhaltung, erweitert. Viele Methoden können so auch über die Actiondatei angesprochen werden, wie z.B. *LatinAmerica1 37m ChordNode:storeMyPOSList*. Hiermit speichert dann ein Knoten in der Minute 37 seine *PoS-Liste* in der DHT und veröffentlicht damit Informationen über den Aufenthalt der Datenreplikationen.

**ChordMessageHandler.java**

Der *ChordMessageHandler* ist die zentrale Stelle, wo fast alle ankommenden Nachrichten bearbeitet werden. Diese Klasse wird erweitert, um mit den von uns verwendeten Nachrichten für den Datenhaltungsservice umgehen zu können. Bei einer ankommenden Nachricht wird vom Simulator *messageArrived(TransMsgEvent receivingEvent)* aufgerufen, womit die Klasse aus dem *TransMsgEvent* ihre übertragene Nachricht erhält. Durch if-Abfragen wird dann geschaut, welcher Typ Nachricht angekommen ist und entsprechende Handlungen danach ausgeführt.

### 4.2.2 Einbau der realen Freundschaftsbeziehungen in die Simulation

Der Einbau der extrahierten Facebook-Beziehungen findet in der Klasse *ChordNodeFactory.java* statt. Man findet diese im Package *org.peerfact.impl.overlay.dht.chord.dunbchord.components*. Diese Klasse erstellt die Komponenten (Hosts) für die Simulation. Außerdem wird hier an jeden erstellten Knoten eine generierte *UserID* vergeben, wobei die Vergabe bei 0 anfängt. Die entnommenen *FacebookIDs* werden dann mit den im Simulator verwendeten *UserIDs* gemappt und können nun für die Simulation verwendet werden. Die Extraktion der 1000 Knoten aus der Datei mit den Facebook-Beziehungen [12], die hier für die Simulation verwendet werden, übernimmt die implementierte Klasse *NodeGraphReader.java*. Der ganze Graph an sich wird in der wissenschaftlichen Arbeit [13] näher beschrieben. Jeder Knoten bekommt darüber hinaus die entsprechende Freundesliste zugewiesen, die durch *NodeGraphReader* exrahiert wurden. Mit der implementierten Methode *getNodeGraphMap()* wird eine *HashMap* zurückgegeben, wobei hier *UserIDs* auf eine Freundesliste (eine *LinkedList* mit *UserIDs*) abbilden. Den daraus resultierenden, ungerichteten Graphen sieht man hier in der Abbildung 4.1. Diese 1000 Knoten werden dann im weiteren Teil verwendet, um den Datenhaltungsservice zu testen.

Mittels *NodeGraphReader*, werden die ersten 1000 Knoten und deren Freundschaftsbeziehungen aus der Datei mit den realen Facebook-Beziehungen [12] extrahiert. In der Datei sind Knoten befreundet, falls in einer Zeile steht: *NodeID1 NodeID2 TieStrength*. Die *TieStrength* wurde für unser Vorhaben nicht verwendet, da wir nur eine realitätsnahe Modellierung von Freundschaftsbeziehungen für unsere Simulation benötigen. Bei den verwendeten Facebook-Beziehungen handelt es sich um einen gerichteten Graphen, der dann für die Simulation in einen ungerichteten umgewandelt wurde.

### 4.2.3 Datenreplizierung, Wahl des 2. PoS, Aufrechterhaltung der PoSs

Es wurden zwei Operationen implementiert, die die Datenreplizierung, die Wahl des zweiten PoSs und die Aufrechterhaltung der PoSs verwalten. Zum einen die DataAvailabilityOperation, die sich um

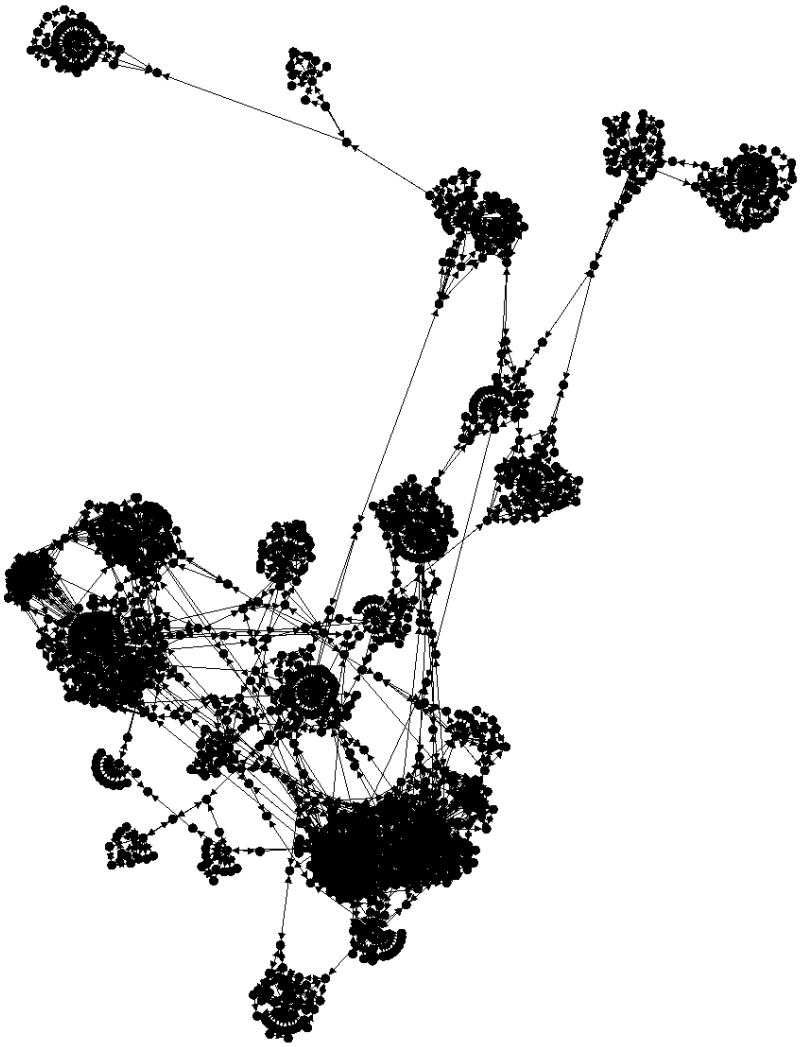


Abbildung 4.1: Graph mit 1000 Knoten und Freundesbeziehungen untereinander

einen zweiten PoS kümmert und Daten zum neuen PoS schickt und die PingPongOperation, die sich um die Aufrechterhaltung der PoSs kümmert und Ausfälle erkennt. Diese werden im Folgenden näher erläutert.

**DataAvailabilityOperation.java**

Diese Operation wird von einem *Besitzerknoten* gestartet, der seine Daten replizieren und im Netzwerk halten möchte. Im ersten Schritt wird die aktuelle *PoS-Liste* angefragt. Hierfür wird ein *Value-Lookup* ausgeführt, welches dann die *PoS-Liste* als *ChordPosListObject* zurückliefert. Dieses Objekt implementiert das *DHTObject*-Interface, das sich dann in der DHT ablegen lässt. Die *PoS-Liste* ist, wie beschrieben in 3, unter dem Hash der *UserID* abgelegt.

Im nächsten Schritt wird mit der Methode *choose2ndPos()* ein zweiter *PoS* aus der Freundesliste des *Besitzerknotens* gewählt. Die zufällige Auswahl unter den in der Freundesliste gespeicherten *UserIDs* erfolgt mit der im Simulator genutzten PseudoRandom-Funktion *Simulator.getRandom()*. Da hier nur die bekannten *UserIDs* zur Verfügung stehen (aus der Freundesliste) muss nach der Wahl des zweiten PoSs wiederum mit der *retrievePoSList* Methode die *PoS-Liste* des Freundes aus der DHT geholt werden. Dadurch erhalten wir dann die *ChordID*. Nach dieser *ChordID* starten wir nun einen *nodeLookup*, wodurch wir an den *ChordContact* kommen und den Freundesknoten direkt kontaktieren können. In diesem *ChordContact* befinden sich sowohl die *ChordID* und die die IP-Adresse unseres Freundes, als auch der Port für die direkte Kommunikation.

Für jede dieser Operationen ist in der *DataAvailabilityOperation.java* ein *Callback* eingebaut. Dafür muss das *OperationCallback*-Interface implementiert werden. Damit ist es dann möglich auf Antworten anderer Operationen zu "warten", wodurch es gelingt asynchrone Operationsaufrufe in der Simulation zu erreichen.

Der nächste Schritt ist das Senden der aktuellen Daten (z.B. Fotos oder Pinnwandeinträge) des *Besitzerknotens*, sowie der Freundesliste mit allen *UserIDs*, an den befreundeten zweiten PoS. Hierfür wird die Methode *sendDataToPoS()* verwendet. Diese Informationen werden dann, mittels des UDP Protokolls, vom Underlay als *PoSMessage* über das simulierte Netzwerk versendet. Als erfolgreiche Antwort bekommt man ein *PoSReply*, womit wir dann wissen, dass unsere *PoSMessage* am Zielknoten angekommen ist. Unser zweiter PoS ist nun ausgewählt.

Danach starten wir die *UpdatePoSListOperation*, was die *PoS-Liste* in der DHT updatet. Nach einem *Callback* der *UpdatePoSListOperation* wird eine *PingPongOperation* zwischen den beiden aktuellen PoSs gestartet, um Ausfälle zu erkennen und bei Bedarf eine neue Wahl eines PoSs zu veranlassen.

**PingPongOperation.java**

Die verwendete *PingPongOperation* wird vom Paket *org.peerfact.impl.overlay.dht.edu.star* verwendet und wurde zum größten Teil angepasst und erweitert. Hier wird eine *PingMessage* zum Zielknoten gesendet. Der Zielknoten muss mit einer *PongMessage* antworten. Daraufhin wird die Operation zu einem bestimmten Intervall neugestartet. Beide Knoten starten jeweils eine eigene *PingPongOperation*, um Ausfälle zu erkennen.

Falls ein Ausfall eines Knotens auftritt, wird die Methode *reactOnOfflineNode()* ausgeführt. Hierbei wird zuallererst die aktuelle *PoS-Liste* aus der DHT beantragt, für den diese *PingPongOperation* ausgeführt wird. Als nächstes wird der ausgefallene Knoten aus der *PoS-Liste* entfernt und eine Aktualisierung der *PoS-Liste* mit der *UpdatePoSListOperation* durchgeführt. Nach der Aktualisierung der *PoS-Liste* in der DHT folgt die Replikation der Daten mittels der *DataAvailabilityOperation*, welche

einen neuen zweiten PoS auswählt und diesem dann Daten, Freundesliste etc. zusendet.

Jeder Knoten speichert sich die Freundeslisten der Besitzerknoten, für die er zuständig ist, in einer *HashMap*.

### 4.2.4 Anfordern von Daten und Aktualisierung der Daten bei den PoSs

Zur Datenanforderung wurde die GetOperation.java Klasse erstellt. Diese fordert Daten befreundeter Knoten an. Außerdem wird hier die UpdateDataOperation beschrieben, die zur Datenaktualisierung dient. Die in der Simulation verwendeten Datenobjekte werden ebenfalls beschrieben.

#### Anforderung von Daten eines Knotens

Falls Knoten Daten von einem ihrer Kontakte aus der Freundesliste anfordern möchten, muss die hierfür implementierte *GetOperation*-Klasse ausgeführt werden. Zunächst erfolgt die Auswahl der *UserID*, deren Daten angefordert werden sollen. Danach muss die entsprechende *PoS-Liste* aus der DHT angefordert werden. Dies wird mit der Methode *retrievePoSList()* durchgeführt. Falls diese Operation erfolgreich war, bekommen wir eine *PoS-Liste*, wodurch wir eine Auswahl von zwei Kontakten *ChordContacts* haben. Die Wahl wird hier mit der Pseudorandom-Funktion ausgeführt. Als nächstes wird dem Zielknoten eine *GetDataMessage* gesendet. Falls der Zielknoten sich zu diesem Zeitpunkt im Netzwerk befindet, bekommen wir eine *GetDataReply* samt den gewünschten Daten zugesendet. Falls keine Antwort zurückkommt, wird der andere PoS in der Liste kontaktiert, um uns die gewünschten Daten zu schicken.

#### UpdateDataOperation.java

Diese Operation wird genutzt, um ein Datenupdate auf den PoSs durchzuführen, falls der *Besitzerknoten* seine Daten aktualisiert oder verändert. Die Versionierung wird hier also verwaltet. Der *Besitzerknoten* holt sich zuerst die *PoS-Liste* aus der DHT und sendet beiden PoSs die neuen Daten mit einer *DataUpdateMessage* zu. Als Antwort kommt dann ein *DataUpdateReply*.

#### Verwendete Daten in der Simulation

Für diese Simulation werden zufällig jeweils 1 - 7 Datenelemente pro Knoten generiert, die eine Größe von 1 - 1024 Byte besitzen und in eine *HashMap* abgelegt werden. Anfragen nach diesen Datenobjekten werden vom Knoten beantwortet und eine Hashtabelle mit den Datenelementen zum anfragenden Zielknoten gesendet. In Tests mit größeren Dateien (1 - 5 MB) hatten sich Probleme ergeben. Viele dieser Nachrichten wurden gedroppt. Leider hat die Zeit nicht mehr gereicht, um dieses Problem anzugehen.

### 4.2.5 Aktualisierung der PoS-Listen

Die Methoden und Mechanismen für die Aktualisierung der PoS-Listen werden nun in den folgenden Abschnitten beschrieben.

**UpdatePosListOperation.java**

Bei einer Änderung der PoSs für einen *Besitzerknoten* muss die *PoS-Liste* in der DHT aktualisiert werden. Dies übernimmt die implementierte Klasse *UpdatePosListOperation*. Als erstes wird für die bestimmte *UserID* eines *Besitzerknotens* die *PoS-Liste* aus der DHT mit dem Methodenaufruf *retrievePoSList(ownerUserID, callbackRetrieveChordPosList)* aus der Klasse *ChordNode* angefordert. Nach einem *Callback* wird dann die Versionsnummer inkrementiert und die als Parameter übergebene neue *PoS-Liste* gesetzt. Anschließend wird der *ChordKey* aus der *UserID* ermittelt, wonach wir dann einen *nodeLookup* starten, um die Kontaktdaten des zuständigen Knotens, die im *ChordContact* enthalten sind, zu bekommen. Nach einem weiteren *Callback* der *LookupOperation* kontaktieren wir den zuständigen Knoten mit einer *PoSListUpdateMessage*. In dieser Nachricht sind enthalten: Sender, Empfänger, PoS-Liste und die *UserID* des *Besitzerknotens*. Nach Erhalt des *PoSListUpdateReply* wird die Operation mit *operationFinished(true)* beendet.

**Erhöhung der Ausfallsicherheit der *PoS-Listen***

Wenn ein Knoten das Netzwerk verlässt, kann es passieren, dass für eine bestimmte *UserID* die *PoS-Liste* in der DHT nicht mehr verfügbar ist. Die Chord-Implementierung des Simulators sorgt nicht für eine Weitergabe von DHT-Einträgen an den *Successor* bei Verlassen des Netzwerks. Eine Überlegung ist hier die *PoS-Liste* hinter 2-3 weitere *ChordIDs* in der DHT abzulegen. Man könnte die gehashte UserID nochmals hashen, dann wiederum diesen Hash erneut hashen, so dass man am Ende die *PoS-Liste* auf 2-3 ChordIDs verteilt hat. Möglich ist auch das Nutzen von drei Hashfunktionen oder die Verwendung der drei nächsten Nachfolger als Replica-Knoten. Dies würde zu mehr Traffic im Overlay führen, jedoch würde man dadurch die Erreichbarkeit erhöhen. Man müsste dann beobachten, wie sich das Overlay mit dieser Änderung verhält.

### 4.2.6 Beabsichtigtes Verlassen des Netzwerks eines PoS

Nun kommen wir zum beabsichtigtem Verlassen des Netzwerks. Hierfür wurde die PoSLeaveOperation implementiert.

**PoSLeaveOperation.java**

Verlässt ein Knoten gewollt das Netzwerk, wird die Operation *PoSLeaveOperation* ausgeführt. Der Knoten kontaktiert alle Knoten, die ebenfalls PoSs für einen bestimmten *Besitzerknoten* sind und schickt ihnen eine *PoSWantsToLeaveMessage*, um zu signalisieren, dass er jetzt das Netzwerk verlässt. Zusätzlich ist in der Nachricht vermerkt, für welche *UserID* eines Besitzerknotens man sich abmeldet. Um die PoSs zu kontaktieren wird für jede *UserID* eines Besitzerknotens, mit der Methode aus der Klasse *ChordNode.java retrievePoSList(ownerUserID, callbackChordPosList)*, die aktuelle *PoS-Liste* angefordert. Bei erfolgreichem Ausführen dieser Operation, trägt sich der Knoten, der das Netzwerk verlässt, aus der *PoS-Liste* aus und führt eine Aktualisierung der *PoS-Liste* mit der *UpdatePoSListOperation* durch. Nach erfolgreicher Durchführung kommt dann zuletzt die Benachrichtung an den anderen PoS. Dieser Knoten startet die *DataAvailabilityOperation*, um einen neuen PoS aus der Freundschaftsliste des *Besitzerknotens* zu wählen. Als Antwort erhält man einen *PoSWantsToLeaveReply*, wonach die *PingPongOperation* zum anderen Knoten beendet und dieser aus der internen Liste ausgetragen wird. Erst wenn alle PoSs kontaktiert wurden, wird die *LeaveOperation* durchgeführt. Falls ein *Messagetimeout* auftritt, sorgt die *PingPongOperation* dafür, dass ein neuer PoS ausgewählt wird.

### 4.2.7 DHT-Eintragsverwaltung

PoS-Listen werden in der DHT abgelegt. Es kommt aber vor, dass Knoten aus dem Netzwerk fallen und dadurch die PoS-Listen nicht mehr erreichbar sind. Aus diesem Grund wurde versucht dieses Problem mit einer DHT-Eintragsverwaltung anzugehen. Dies wird im folgedem Abschnitt beschrieben.

**Netzwerk beitreten**

Die Standard Chord-Implementierung des Simulators sorgt nicht für Replikation und Erhalt der DHT-Einträge. Deswegen wurde eine DHT-Eintragsverwaltung implementiert, welche dafür sorgt, dass die Knoten sich in der *Join-Phase* die DHT-Einträge, entsprechend ihrer Zugehörigkeit, untereinander zuschicken. Die Klasse *ChordMessageHandler.java* in der Chord-Implementierung wurde um die Methode *handleDHTEntrys* erweitert und wird, falls eine *HandshakeMsg* eintrifft, ausgeführt.

**Netzwerk verlassen**

Zusätzlich wird der Fall, falls ein Knoten das Netzwerk verlässt behandelt. Verlassen Knoten gewollt das Netzwerk, teilen sie den PoSs ihr Vorhaben mit. Weiterhin werden die DHT-Einträge, wie z.B. die *PoS-Liste*, an den Nachfolger versendet. Diese Funktion wurde in der Klasse *ChordNode.java* hinzugefügt. Dafür wurde die Methode *handleDHTEntrysWhenLeaving* implementiert und die Methode in *leaveoperationFinished* angepasst. Der Knoten, der das Netzwerk verlässt, schickt dem *Successor* hiermit in einer *dhtEntrysReply* diese Einträge und verlässt danach das Netzwerk. Der *Successor* trägt daraufhin diese bei sich in die DHT ein. So wird versucht die *PoS-Listen* weitestgehend in der DHT bzw. im Netzwerk zu halten, falls ein Knoten das Netzwerk verlässt.

### 4.2.8 Erneutes Betreten des Netzwerks und Mechanismen

Kommen Knoten wieder ins Netzwerk, sollen diese einige Überprüfungen starten. Die implementierten Operationen hierfür werden in den folgenden Abschnitten beschrieben.

**NodeOnlineAgainOperation.java**

Kommt ein Knoten nach einem *Churn-Event* wieder online, wird die Operation *NodeOnlineAgainOperation* direkt ausgeführt. Hier wurde in *ChordNode.java* die Methode *joinOperationFinished(successor, predecessor, succFingerTable)* um die Überprüfung erweitert, ob der Knoten, der wieder das Netzwerk betritt, denn schon einmal früher online war. An dieser Stelle wird *NodeOnlineAgainOperation* angestoßen, womit eine Überprüfung beginnt, ob man Datenobjekte anderer *Besitzerknoten* bzw. eigene Datenobjekte besitzt. Trifft dies zu, so wird mit dem Methodenaufruf *retrievePoSList(ownerUserID, callbackChordPosList)* der Klasse *ChordNode* eine *ValueLookupOperation* nach allen *PoS-Listen* der *UserIDs* gestartet, für die der aktuelle Knoten alte Daten besitzt. Als Ergebnis der *ValueLookupOperation* werden dann jeweils die *PoS-Listen* als *DHTObject* zurückgeliefert. Hier kann jetzt die *PoS-Liste* mehrere Fälle haben:

- *PoS-Liste* ist leer: Hier trägt der Knoten seinen *ChordContact* in die *PoS-Liste* ein, startet mit dem Methodenaufruf *beFirstPos(ownerUserID)* eine Aktualisierung der *PoS-Liste* und nach einem erfolgreichen *Callback* die Wahl eines zweiten PoSs.

- *PoS-Liste* hat die Größe 1: Es findet zuerst eine Überprüfung statt, ob dieser eingetragene PoS man selbst ist. Ist dies der Fall, wird wieder die Methode *beFirstPos(ownerUserID)* angewendet, womit sich dieser PoS als ersten PoS setzt und einen zweiten PoS wählt. Trifft das nicht zu, wird der Knoten mittels *CheckOnlineOperation* nach dem Onlinestatus überprüft und falls als Ergebnis feststeht, dass dieser sich nicht mehr im Netz befindet, wird erneut *beFirst-*

*Pos(ownerUserID)* verwendet, um sich als ersten PoS zu setzen und einen zweiten neuen PoS aus der Freundschaftsliste des jeweiligen *Besitzerknotens* auszuwählen.

- *PoS-Liste* hat die Größe 2: Erneut startet zuerst eine Überprüfung, ob man einer dieser eingetragenen PoSs ist und es wird, mit der *CheckOnlineOperation*, der Onlinestatus der Knoten überprüft. Ist man noch eingetragen und der andere Knoten online, kann es evtl. noch eine kurze Zeit dauern bis die *PingPongOperation* den Ausfall bemerkt und einen neuen PoS wählt. Hier muss man eigentlich nichts machen. Falls der PoS jedoch offline und die Liste somit veraltet ist, wird *beFirstPos(ownerUserID)* wieder angestoßen, um sich selbst als ersten PoS zu setzen und eine Wahl des zweiten PoSs durchzuführen. Sind zwei andere PoSs eingetragen, wird der Onlinestatus dieser Knoten mit *CheckOnlineOperation* überprüft und bei erfolgreichem *Callback* die *CheckDataVersionOperation* ausgeführt, womit die Version der Datenobjekte auf den PoSs überprüft wird und, falls notwendig, eine Übertragung der aktuelleren Datenobjekte veranlasst.

**CheckOnlineOperation.java**

Diese Operation überprüft, ob die in der *PoS-Liste* enthaltenen Knoten online sind. Als Parameter wird hier die *PoS-Liste* übergeben. Danach wird jedem enthaltenen *ChordContact* eine *CheckOnlineMessage* gesendet. Die abgeschickten Nachrichten und deren eindeutige KommunikationsID werden in einer *HashMap* zwischengespeichert. Damit kann ein Nachrichten-Timeout einem Knoten zugeordnet werden, falls *messageTimeoutOccured(int commId)* vom Simulator ausgelöst wird, da der Zielknoten zur Zeit nicht antwortet. Hier werden dann maximal drei Nachrichten zum Zielknoten geschickt, bevor dieser Versuch beendet wird. Kommt ein *CheckOnlineReply* eines Knotens zurück, wissen wir, dass dieser noch online ist. Die Liste lässt sich dann nach einem *Callback* mittels *getResult()* einsehen.

**CheckDataVersionOperation.java**

Die Versionen der Daten, die bei den PoSs gespeichert sind, wird mit dieser Operation überprüft. Der Kostruktor der Operation benötigt einmal den Knoten, der diese Operation ausführt, ein *OperationCallback*, die *PoS-Liste* und die *UserID* des *Besitzerknotens*, für den diese Überprüfung gestartet wird. Danach wird den PoSs eine *CheckDataMessage* gesendet. Weiterer Inhalt dieser Nachricht sind die *UserID* des *Besitzerknotens* und die Version der Daten, die man selbst besitzt. Die implementierte Methode *receive* erwartet einmal als Nachricht eine *CheckDataUpToDateReply*, womit mitgeteilt wird, dass der Zielknoten aktuelle Daten besitzt. Außerdem erwartet sie eine *CheckDataNotUpToDateReply*, womit wir als Antwort eine *DataUpdateMessage*, samt aktueller Datenobjekte, zurückschicken. Zum Schluss erwartet die Methode ein *DataUpdateReply*, wodurch signalisiert wird, dass die aktuellen Daten angekommen sind.

### 4.2.9 Testen der Implementierung

Um die Implementierung zu testen und zu konfigurieren bzw. um Daten für die Evaluation zu sammeln wurden diese drei Klassen implementiert, um diese Aufgaben zu erledigen.

**PeriodicRandomOperation.java**

Um das DunbChord-Overlay auszutesten, wurde eine Operation implementiert, die zufällig eine bestimmte Operation ausführt. Diese Operation wird in einem Intervall von 60 Sekunden von jedem Knoten im Netzwerk ausgeführt, wobei hier zu einer bestimmten Wahrscheinlichkeit eine GetOperation, PoSLeaveOperation, JoinOperation oder eine Methode aufgerufen wird, die Daten verändert und diese auf den Knoten aktualisiert. Zusätzlich wird eine UpdateDataOperation ausführt, die dann ein Update an die aktuellen PoSs verschickt um die Datenkonsistenz zu wahren. Hier gibt es zudem die Möglichkeit verschiedene Wahrscheinlichkeitswerte einzustellen.

**DunbarChordAnalyzer.java**

Die Implementierung von Chord wurde außerdem um den Analyzer *DunbarChordAnalyzer* erweitert, der die für uns wichtigen Daten zur Evaluierung sammelt und in eine Datei schreibt. Auf Grundlage dieser Datei werden die Messdaten später mittels *gnuplot* [] dargestellt.

**DunbarChordConfiguration**

Mit dieser Klasse kann man einige Parameter für die Simulation festlegen, wie zum Beispiel in welchen Abständen von den PoSs die *PingPingOperationen* gestartet werden sollen.

# Kapitel 5

# Evaluation der Implementierung

In diesem Kapitel wird die Evaluation, des *DunbChord* Overlay, welches um ein Datenhaltungsservice erweitert wurde, anhand einiger Parameter durchgeführt. Es wird einmal die Simulation ohne und einmal mit Churn betrachtet. Die hier verwendeten Freundesbeziehungen sind ein Teil realer Facebookbeziehungen, die hier genutzt werden, um den Datenhaltungsservice zu testen. In Abschnitt 5.1 werden die Einstellungen des Simulators beschrieben. Nachfolgend in 5.2 werden die Szenarien und Parameter dargelegt. Am Ende werden in Abschnitt 5.3 die Ergebnisse der Simulation dargestellt und besprochen.

## 5.1 Einstellungen des Simulators

Einen kleinen Überblick über die wichtigsten Einstellungen im Simulator gibt Tabelle 5.1. Wie in Tabelle 5.1 zu sehen, wird PeerfactSIM.KOM [7] als Simulator verwendet. Die Simulationen werden ohne Paketverlust durchgeführt. Außerdem werden Jitter aus Messungen des PingER Projekts [14] und GNP [15] verwendet, um Distanzen und Verzögerungen zwischen zwei Knoten zu bestimmen.

| Verwendeter Simulator | PeerfactSim.KOM |
|---|---|
| Netzwerkeinstellungen | kein Paketverlust, GNP, Jitter (PingER Projekt) |
| Churn | KadChurnModel |
| Knotenanzahl | 1000 mit Freundschaftsbeziehungen untereinander |
| Overlay | DunbChord(modifiziertes Chord) |

Tabelle 5.1: Überblick Einstellungen des Simulators

## 5.2 Einstellungen der Szenarien und Parameter

Die Simulationen in *PeerfactSIM.KOM* laufen mit 10 zufällig gewählten *Seeds*. Es werden 1000 Knoten, die aus der Datei mit den Facebookbeziehungen [12] für diese Simulation extrahiert wurden, verwendet. Die Laufzeit der Szenarien 1, 2 und 3 wird auf 120 Minuten gesetzt. Der Start der für den Datenhaltungsservice relevanten Operationen, wird bis auf Minute 44 - 46 verlegt, da die *Join-Phase*, wie in 5.2.1 beschrieben, sich über Minute 30 erstreckt. In Abschnitt 5.2.2 wird nur die pure Erreichbarkeit der Daten ohne Einflüsse (kein Churn) betrachtet. Weiter in Abschnitt 5.2.3 wird das Szenario 2 besprochen. Der Fokus liegt hier auf der Erreichbarkeit der Daten, wenn Knoten ihren PoSs mitteilen, dass sie das Netzwerk verlassen. Und in Abschnitt 5.2.4 das Szenario 3, wobei hier das Churn aktiviert wird.

### 5.2.1 Join-Phase Chord Implementierung

Die *Join-Phase* wurde auf 1s - 30m, also von Sekunde 1 bis Minute 30 gesetzt. Ab Minute 44 generieren alle Knoten ihre Datenobjekte. Danach wird die Speicherung der PoS-Listen im Netzwerk ab Minute 46 gestartet. In Tests hat sich gezeigt, dass Simulationen, die nur die *Join-Phase* betreffen, teilweise bis Minute 42 oder länger benötigen, bis der ganze Chord-Ring sich aufgebaut hat. Aus diesem Grund wurde der Start der für die Datenhaltung relevante Operationen ab Minute 44 - 46 gesetzt, damit auch alle Besitzerknoten ihre PoS-Liste in der DHT ablegen können. Hierdurch soll das Ergebnis der Simulationen bezüglich des Datenhaltungsservices von *DunbChord* so wenig wie möglich verfälscht werden. Dadurch, dass der Chord-Ring nicht vollständig aufgebaut wird, bzw. da die *Join-Phase* mit manchen *Seeds* nicht vollständig beendet wird, können nicht alle Knoten ihre PoS-Listen veröffentlichen. Somit fallen die Erreichbarkeiten der Daten bei diesen *Seeds* unter ein normales Level.

In der Tabelle 5.2 sieht man eine Übersicht aller Szenarien. Die einzelnen Beschreibungen dieser folgen in den kommenden Abschnitten.

| Szenario | 1 | 2 | 3 |
|---|---|---|---|
| Aktionen | Join | Join | Join |
| | Datengenerierung | Datengenerierung | Datengenerierung |
| | PoS-Listen in DHT speichern | PoS-Listen in DHT speichern | PoS-Listen in DHT speichern |
| | Start Datenhaltung | Start Datenhaltung | Start Datenhaltung |
| | periodische GetOperation | periodische GetOperation | periodische GetOperation |
| | | periodische Zufallsoperationen | Start Churn |

Tabelle 5.2: Übersicht der Szenarien 1 - 3

### 5.2.2 Szenario 1: Überprüfung der Datenerreichbarkeit

In Szenario 1 wird überprüft, ob die *GetOperation* ihren Dienst erfüllt und Datenobjekte eines bekannten Freundes anfordert. Es soll gezeigt werden, dass Datenobjekte von den PoSs bezogen werden und auch ihr Ziel, den anfragenden Freund-Knoten, erreichen. Wie zu sehen in Tabelle 5.3 treten von Sekunde 1 bis Minute 30 alle 1000 Knoten dem Netzwerk bei. Gegen Minute 44 generieren dann alle Knoten im Netzwerk Datenobjekte, die dann später den PoSs geschickt werden können. Ab Minute 46 startet die Speicherung der PoS-Listen in der DHT, somit wissen nun Freunde der Besitzerknoten, wo ihre Datenobjekte zu finden sind. Des Weiteren folgt in Minute 49 der Start des Datenhaltungsservices. Hier wird ein zweiter PoSs ausgewählt und die Datenreplikation gestartet. Um zu überprüfen, wie viele Daten in diesem Szenario erreichbar sind, startet dann eine periodische *GetOperation* ab Minute 56m. Dieses Szenario läuft dann bis Minute 120, es tritt kein Churn auf, da hier nur die Anfrage der Datenobjekte aus der DHT im Vordergrund stehen.

| Szenario | 1 |
|---|---|
| Aktionen | Join |
| | Datengenerierung |
| | PoS-Listen in DHT speichern |
| | Start Datenhaltung |
| | periodische GetOperation |

Tabelle 5.3: Szenario 1: Überprüfung der Datenerreichbarkeit

### 5.2.3 Szenario 2: Datenerreichbarkeit und PeriodicRandomOperation

Mit Szenario 2 soll überprüft werden, wie denn jetzt die Erreichbarkeit der Daten aussehen, falls Knoten gewollt aus dem Netzwerk austreten und somit den anderen PoSs Bescheid sagen. Zudem besteht die Möglichkeit, dass diese wieder dem Netzwerk beitreten. Hier wird dann geschaut, wie gut Datenobjekte von den PoSs bezogen werden können. Wie zu sehen in Tabelle 5.4 treten von Sekunde 1 bis Minute 30 alle 1000 Knoten dem Netzwerk bei. Gegen Minute 44 generieren dann alle Knoten im Netzwerk Datenobjekte, die später den PoSs geschickt werden können. Ab Minute 46 startet dann die Speicherung der PoS-Listen in der DHT, somit wissen nun Freunde der Besitzerknoten, wo ihre Datenobjekte zu finden sind. Des Weiteren folgt in Minute 49 der Start des Datenhaltungsservices. Hier wird ein zweiter PoSs ausgewählt und die Datenreplikation gestartet. Um zu überprüfen, wie viele Daten erreichbar in diesem Szenario sind startet dann eine periodische *GetOperation* ab Minute 56m. Letztlich startet ab Minute 76 eine *PeriodicRandomOperation*, die zufällig zwischen gewolltem Verlassen, gewolltem erneuten Betreten des Netzwerks und Datenänderung plus Datenaktualisierung auf den PoSs, entscheidet. Diese Szenario läuft dann bis Minute 120.

| Szenario | 2 |
|---|---|
| Aktionen | Join |
| | Datengenerierung |
| | PoS-Listen in DHT speichern |
| | Start Datenhaltung |
| | periodische GetOperation |
| | periodische Zufallsoperationen |

Tabelle 5.4: Szenario 2: Datenerreichbarkeit und PeriodicRandomOperation

#### Szenario 2A: Datenerreichbarkeit mit aktivierter DHT-Eintragsverwaltung

In diesem Szenario wird die DHT-Eintragsverwaltung aktiviert. Hierbei werden die DHT-Einträge beim Verlassen Netzwerks an den Nachfolger-Knoten gesendet. Außerdem bekommen Knoten, die das Netzwerk erneut betreten, von ihren Successor (Nachfolger) nach dem *HandshakeReply* die DHT-Einträge für ihr Verwaltungsintervall zugeschickt. Die Erreichbarkeit der Datenobjekte müssten sich signifikant erhöhen, da um einiges mehr an DHT-Einträgen im Netzwerk gehalten werden müssten. Die DHT-Eintragsverwaltung wird ab Minute 46 aktiviert, also ab dem Zeitpunkt, wo die Besitzerknoten ihre PoS-Listen in der DHT speichern.

### 5.2.4 Szenario 3: Datenerreichbarkeit mit Churn

Im dritten Szenario sind die Operationen bis Minute 56 genau wie im Szenario 2 (s. Abschnitt 5.2.3). Ab Minute 76 tritt dann in diesem Szenario Churn ein, welches ein willkürliches Verlassen und Betreten des Netzwerks bewirkt. Die Knoten verlassen dann, ohne eine Mitteilung, abrupt das Netzwerk. Das Szenario 3 läuft ebenfalls bis zu Minute 120. Die ausgeführten Operationen werden in Tabelle 5.5 dargestellt.

| Szenario | 3 |
|---|---|
| Aktionen | Join |
| | Datengenerierung |
| | PoS-Listen in DHT speichern |
| | Start Datenhaltung |
| | periodische GetOperation |
| | Start Churn |

Tabelle 5.5: Szenario 3: Datenerreichbarkeit mit aktiviertem Churn

### 5.2.5 Gewolltes Verlassen und Betreten des Netzwerks und Churn

Um ein gewolltes Verlassen und erneutes Betreten des Netzwerks zu erreichen, werden in der *PeriodicRandomOperation* mit jeweils 4% Wahrscheinlichkeit die *PoSLeaveOperation* oder die *JoinOperation* ausgeführt. Diese 4% wurden willkürlich gewählt und haben ein ähnliches Verhalten wie das Churn-Modul. Churn nach dem *KadChurnModel* ist hier jedoch deaktiviert und läuft nicht zusammen mit der *PeriodicRandomOperation*.

Beim Szenario 3, in dem Churn aktiv ist, sind dann jedoch das gewollte Verlassen und Betreten des Netzwerks deaktiviert. Hier liegt der Fokus auf dem aktivierten Churn und der Frage, wie sich das *DunbChord* Overlay bezüglich der Datenerreichbarkeit unter diesen Bedingungen verhält.

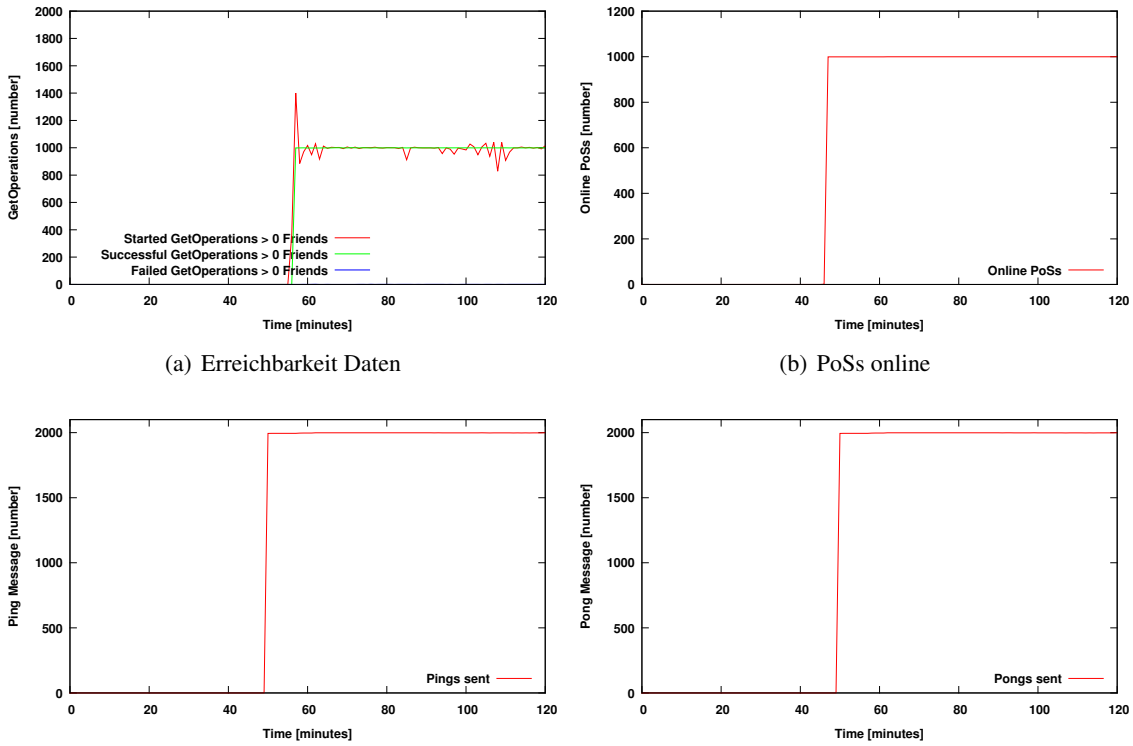
## 5.3 Ergebnisse der Simulation

Im Folgenden werden die erhaltenen Ergebnisse in Abschnitt 5.3.1, 5.3.2 und 5.3.3 besprochen und vorgestellt. Interessant ist es zu wissen, wie gut die Implementierung von DunbChord arbeitet. In [1] kam man auf eine Erreichbarkeit der Daten, die bei ca. 95% lag. Dort wurde gezeigt, dass die Idee, zwei befreundete Point of Storages auszuwählen, ziemlich gut funktioniert. Hier werden ein wenig andere Ergebnisse erwartet, da hier alle 1000 Knoten den Datenhaltungsservice starten und PoSs auswählen. Im Vergleich dazu werden in [1] 1859 Knoten, von denen 28 Besitzerknoten ihre Datenobjekte (Profile) veröffenlichen, verwendet. Es gibt viele Knoten im Netzwerk, die nur mit einem anderen Knoten befreundet sind. Dadurch wird erwartet, dass Knoten mit wenig Freunden Probleme mit der Erreichbarkeit ihrer Daten haben.

### 5.3.1 Ergebnis Szenario 1

Anhand Abbildung 5.1a ist zu erkennen, dass die Gesamterreichbarkeit der Datenobjekte bei 99-100% liegen. Bis zum Ende der Simulation erkennt man um die 1000 gestarteten und erfolgreich beendeten *GetOperations*. Ab Minute 56 starten die Anfragen nach den Daten. Da diese Simulation ohne Churn durchgeführt wurde und nur das Ziel hatte, dass die *GetOperation* die Datenanfragen bearbeitet und Anfragen zu den PoSs schickt, liegen die Ergebnisse nahe der 100%. Somit wissen wir, dass die PoS-Listen in der DHT abgelegt wurden. Des Weiteren wissen wir, dass der Datenhaltungsservice einen zweiten PoS ausgewählt hat und diesem ein Replikat der Datenobjekte des Besitzerknoten gesendet hat. Wie in Abbildung 5.1b zu sehen, sind alle 1000 Knoten im Netzwerk zugleich auch ein PoS für ihre Daten oder für Daten eines Freundes. Die Kosten, um mögliche Ausfälle zu erkennen, sind konstant für Ping und Pong Nachrichten mit jeweils 2000 versendeten Nachrichten bis zum Ende der

Simulation. In Abbildung 5.2a erkennt man, dass sich permanent 1000 Knoten im Netzwerk befinden, da hier, wie erwähnt, keine Ausfälle in der Simulation vorkommen. In Abbildung 5.2c sind Peaks einmal in Minute 46 und 49 dargestellt. Hier startet jeder Knoten einmal die Operation *storePoSList* und dann wenige Minuten später den Datenhaltungsservice, wodurch natürlich dieser starke Peak zu sehen ist. Hier werden in dem Moment die meisten Nachrichten im Netzwerk versendet. Ab Minute 56 beginnen dann die Anfragen der Datenobjekte, um zu überprüfen, ob alle Objekte erreichbar sind. Dementsprechend werden, wie in Abbildung 5.2d zu sehen, ca. 26 MB bis Ende der Simulation pro Minute versendet. Die Anzahl der Hops liegt im Durchschnitt bei knapp 6, wie in Abbildung 5.2b zu sehen ist.



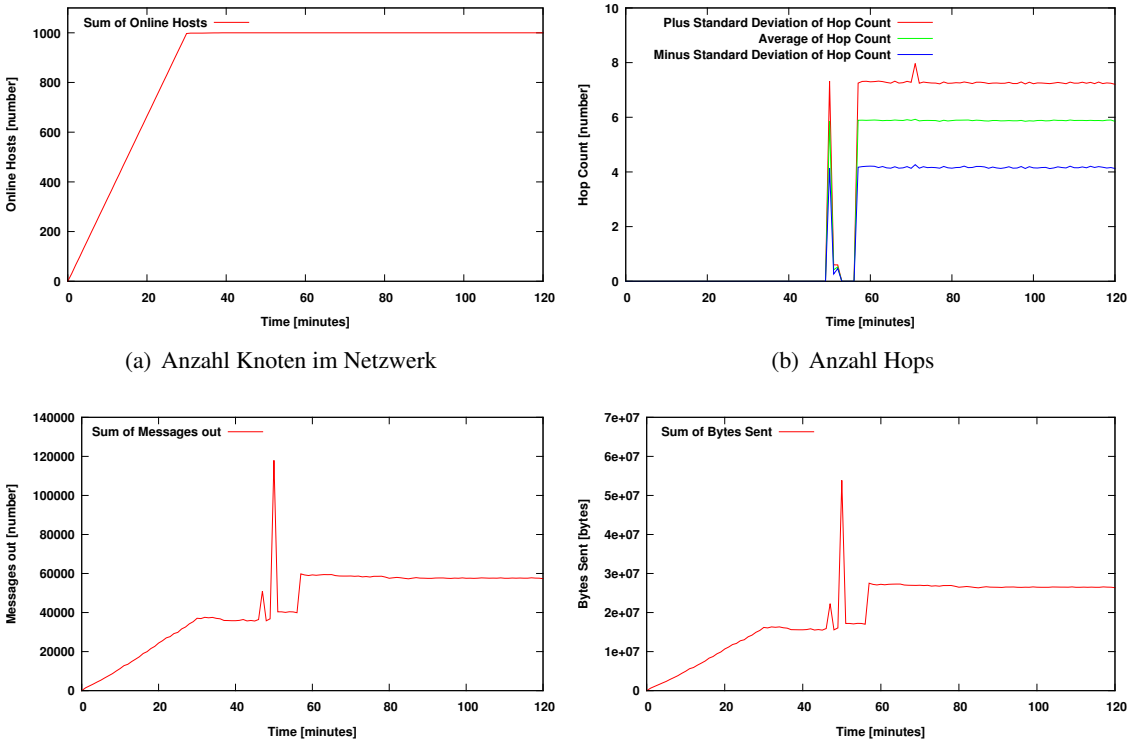

(a) Erreichbarkeit Daten

(b) PoSs online

(c) Kosten Ping Nachrichten

(d) Kosten Pong Nachrichten

Abbildung 5.1: Szenario 1: Erreichbarkeit der Daten, Anzahl der PoSs und Kosten




(a) Anzahl Knoten im Netzwerk

(b) Anzahl Hops

(c) Gesendete Nachrichten

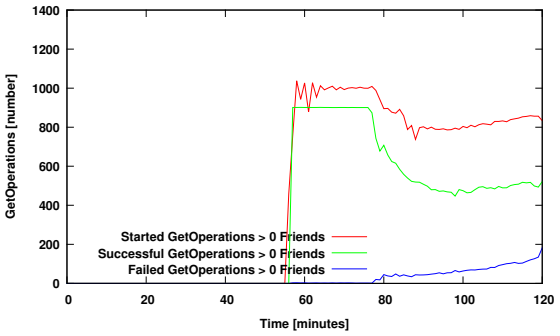
(d) Bytes gesendet

Abbildung 5.2: Szenario 1: Traffic, Anzahl Knoten im Netzwerk und Hops

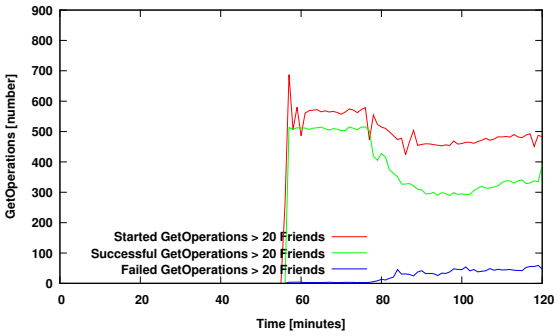
### 5.3.2 Ergebnis Szenario 2

In Abbildung 5.3 ist die Erreichbarkeit der Daten dargestellt. Hier werden die Erreichbarkeiten ermittelt, indem nur Knoten mit Freunden angefragt werden, die jeweils mehr als 0 Freunde , mehr als 20 Freunde und mehr als 40 Freunde haben. Es zeigt sich hier, dass mit Knoten, die einen größeren Freundeskreis besitzen, bessere Werte erzielt werden, als mit Freunden, die zudem einen kleineren Freundeskreis besitzen. Die Erreichbarkeit der Daten scheint also mit der Größe des Freundeskreises zusammenzuhängen. Man erkennt, dass die Gesamterreichbarkeit insgesamt steigt, je höher die Freundesanzahl eines angefragten Knoten ist. Zudem sinken die nicht erfolgreichen *GetOperations* in Richtung der Größe, wo angefragte Knoten mehr als 40 Freunde besitzen. Bei Anfragen an Knoten, die mehr als 0 Freunde haben, erkennt man in Abbildung 5.3a, dass hier die meisten nicht erfolgreichen Operationen vorliegen. Dadurch, dass es hier viele Knoten mit wenig Freunden gibt, die möglicherweise offline gegangen sind, steigen natürlich auch die fehlgeschlagenen Versuche der *GetOperation*. Die Auswahl an möglichen PoSs ist bei diesen geringer. Noch anzumerken ist, zu sehen in Abbildung 5.3a, dass es zu Beginn der *GetOperation*, ab Minute 56 Peaks an gestarteten Operationen gibt. Dies kommt möglicherweise durch die teilweise lange Operationsdauer der ValueLookupOperation und der LookupOperation zustande, die schon in der vorhandenen Chord-Implementierung im Simulator

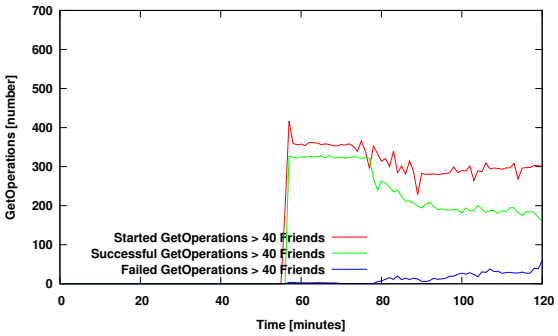
implementiert sind.
Die Kosten der Ping und Pong Nachrichten unterscheiden sich kaum. Diese nehmen ab, wenn ab ca. Minute 76 Knoten gewollt das Netzwerk Verlassen und wieder betreten.
Bei den gesendeten Nachrichten sieht man in Abbildung 5.5c einen Peak bis knapp 500000 Nachrichten. Diese betrifft die *Join-Phase* und tritt höchstwarscheinlich auf, da in der Simulationen von den 10 zufällig gewählten Seeds bei einer oder mehreren Simulationen Probleme auftreten, die den kompletten Aufbau des Chord-Rings betreffen. Ab Minute 46 beginnt dann die Datenhaltung und die Auswahl der PoSs. Des Weiteren werden die Datenobjekte repliziert. Danach folgt der Beginn der *GetOperation*, die periodisch Daten anfordert. Hierdurch liegen die Werte bei mehr als 50000 Nachrichten pro Minute. Ab Minute 76 erkennt man in 5.5d, dass hier der Traffic ein wenig zunimmt, während Knoten das Netzwerk verlassen.
Die durchschnittliche Hopanzahl beim DunbChord Overlay liegt bei ca. 6, zu sehen in 5.5b.
Die gesendeten Bytes in jeder Minute werden in Abbildung 5.5d gezeigt. Diese haben ihren Höhepunkt bei Minute 40 (Chord Join-Phase) mit 250 MB und pendeln sich gegen Ende der Simulation bei 33 MB ein. Dadurch, dass die generierten Datenobjekte mit ca. 1 bis 1024 Byte relativ klein gewählt wurden, kommen diese niedrigen Werte zu Stande.




(a) Anfragen an Knoten mit mehr als 0 Freunden

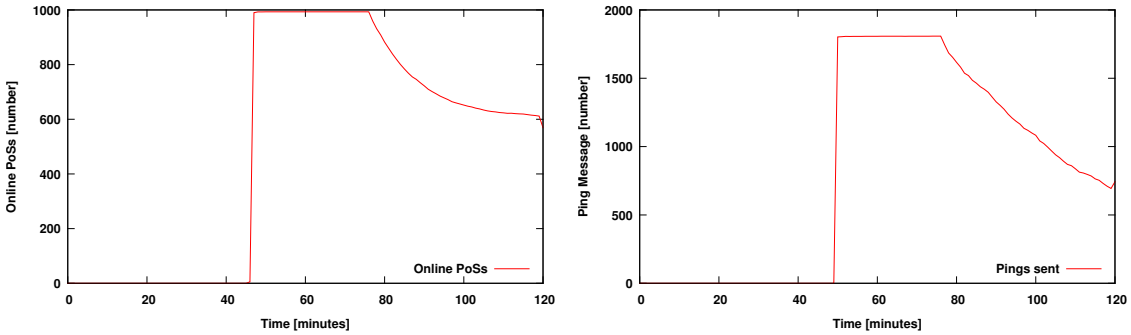



(b) Anfragen an Knoten mit mehr als 20 Freunden



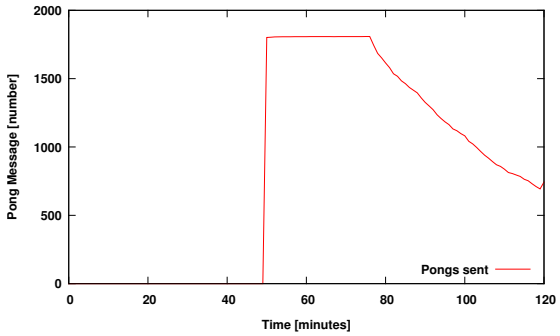

(c) Anfragen an Knoten mit mehr als 40 Freunden

Abbildung 5.3: Szenario 2: Erreichbarkeit der Datenobjekte




(a) PoSs online

(b) Kosten Ping Nachrichten

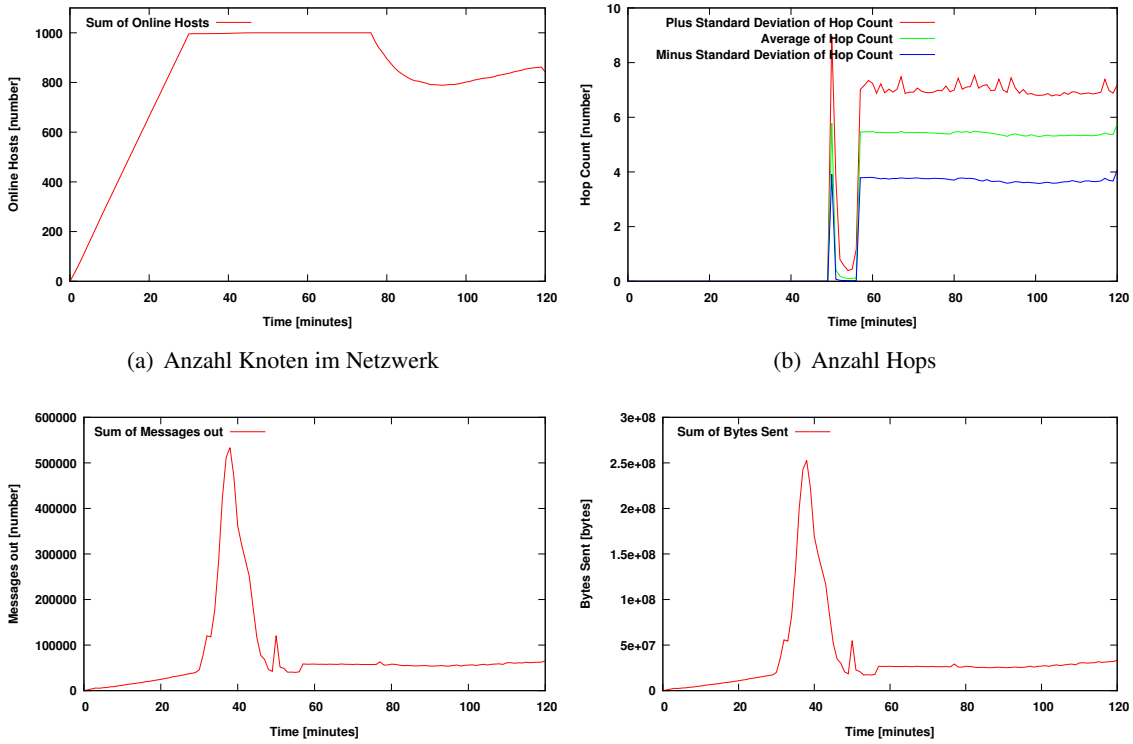



(c) Kosten Pong Nachrichten

Abbildung 5.4: Szenario 2: Online PoSs und Kosten der Ping und Pong Nachrichten

### Ergebnis Szenario 2A

Wie in den Abbildungen 5.6a und 5.6b zu erkennen gibt es einen signifikanten Unterschied in der Erreichbarkeit der Datenobjekte mit aktivierter DHT-Eintragsverwaltung. Diese Simulationen wurden mit *Seed* 286591039 simuliert, um einen genauen Vergleich zu bekommen. Des Weiteren wurden hier Knoten angefragt mit mehr als 20 Freunden. In Abbildung 5.6a erkennt man einen starken Einbruch der erfolgreichen *GetOperationen* zwischen Minute 82 und 110. Hier sind die meisten Knoten gewollt aus dem Netzwerk ausgetreten. Im Vergleich dazu sieht man in 5.6b kaum Einbrüche. Die gestarteten *GetOperationen* liegen knapp auf gleichen Niveau wie die erfolgreichen *GetOperationen*. In Minute 90 werden ca 450 Anfragen gestartet, davon sind fast alle erfolgreich. Bis zum Ende der Simulation wurden damit ziemlich gute Werte erreicht. Zum Vergleich mit Anfragen an Knoten mit mehr als 0 Freunden, sehen wir auch gute Ergebnisse mit der DHT-Eintragsverwaltung (s. Abbildung 5.7).




(a) Anzahl Knoten im Netzwerk

(b) Anzahl Hops

(c) Gesendete Nachrichten

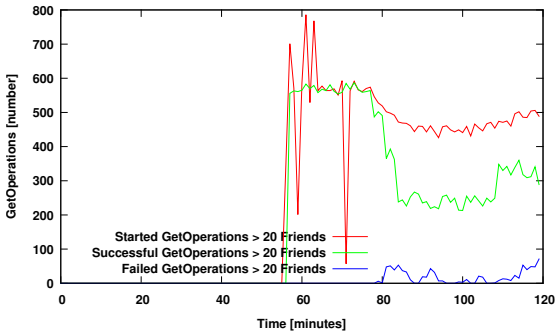
(d) Bytes gesendet

Abbildung 5.5: Szenario 2: Überblick über den Traffic, Hops und Anzahl Knoten im Netzwerk

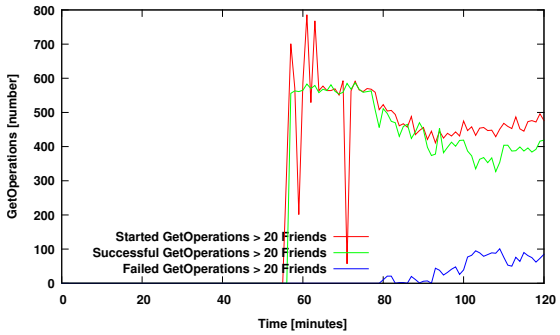
### 5.3.3 Ergebnis Szenario 3

Die Ergebnisse der letzten Simulation zeigen, wie sich *DunbChord* bezüglich des Datenhaltungsservices mit Churn aktiv verhält. Hier geben Knoten nicht bekannt, dass sie das Netzwerk verlassen. Wie in Abbildung 5.8a, 5.8b und 5.8c zu sehen ist, steigt die Erreichbarkeit der Datenobjekte mit der Zunahme des Freundeskreises angefragter Beitzerknoten. Zudem erkennt man in Abbildung 5.10a, dass mit Abnahme der Knoten (bis auf ca. 700) im Netzwerk auch die PoSs (s. Abbildung 5.9a) sinken. Die Kosten pro Minute, um die PoSs zu halten, sind konstant bis zu Minute 76, wo dann Churn aktiv wird. Dadurch, dass Knoten aus dem Nezwerk fallen, senden weniger PoSs ihre Ping und Pong Nachrichten zum anderen PoS.

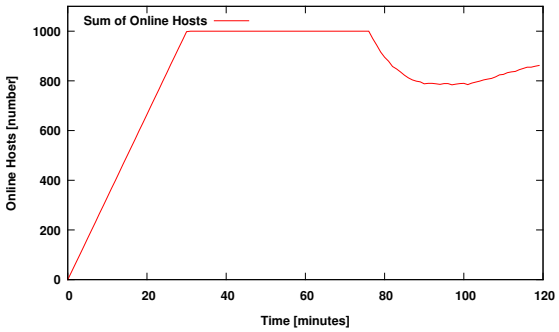
Der Traffic im Netzwerk sieht ähnlich aus wie in den anderen Simulationen. Die beiden Peaks nach Minute 46 und 49 sind in Abbildung 5.10d zu erkennen, wobei hier der Datenhaltungsservice beginnt und vermehrt Nachrichten gesendet werden und befreundete Knoten kontaktiert und als PoSs gewählt und ihnen Kopien der Daten gesendet werden.



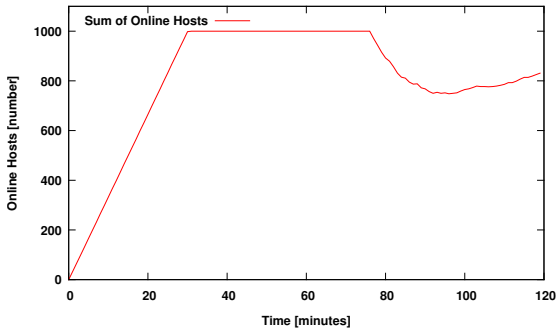

(a) DHT-Eintragsverwaltung deaktiviert




(b) DHT-Eintragsverwaltung aktiviert

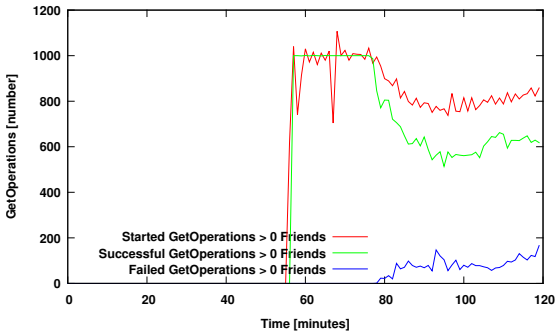



(c) Anzahl Knoten, DHT-Eintragsverwaltung AUS

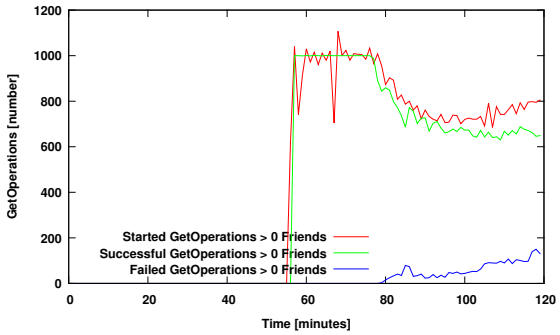



(d) Anzahl Knoten, DHT-Eintragsverwaltung AN

Abbildung 5.6: Szenario 2A: Datenerreichbarkeit mit und ohne DHT-Eintragsverwaltung, Anfragen an Knoten mit mehr als 20 Freunde




(a) DHT-Eintragsverwaltung deaktiviert

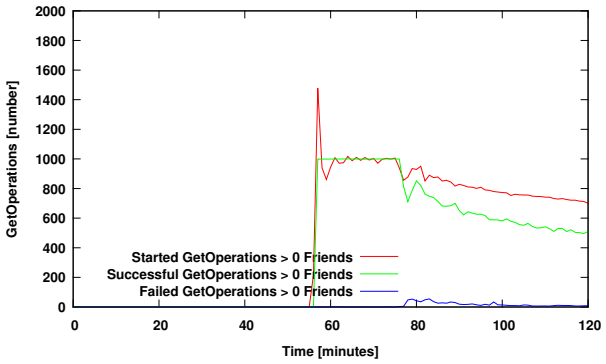



(b) DHT-Eintragsverwaltung aktiviert

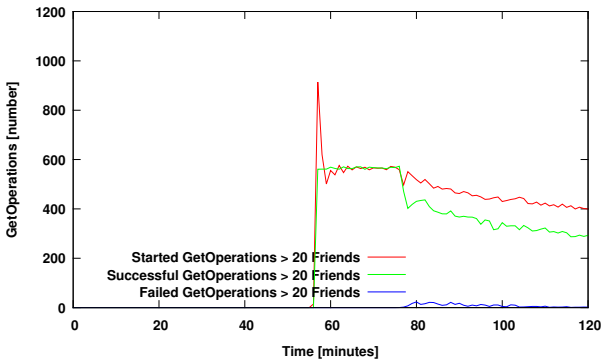
Abbildung 5.7: Szenario 2A: Datenerreichbarkeit mit und ohne DHT-Eintragsverwaltung, Anfragen an Knoten mit mehr als 0 Freunde

### 5.3.4 Zusammenfassung des Kapitels

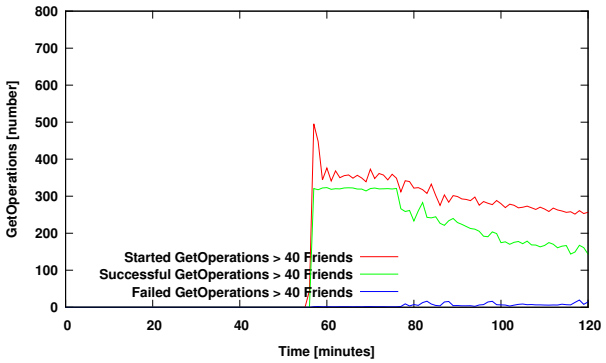
Die Einstellungen des Simulators werden in Abschnitt 5.1 besprochen. In Abschnitt 5.2 werden die Szenarien und verwendeten Parameter für diese Simulation beschrieben. Es werden vier Szenarien




(a) Anfragen an Knoten mit mehr als 0 Freunden




(b) Anfragen an Knoten mit mehr als 20 Freunden

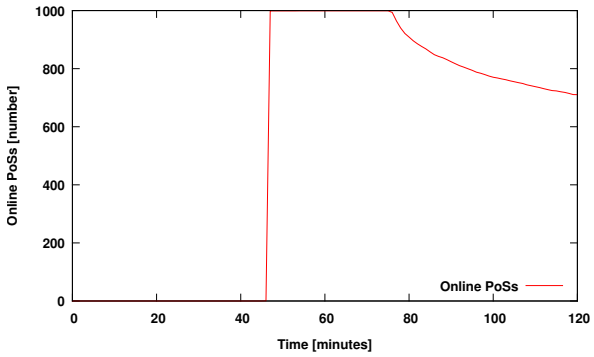



(c) Anfragen an Knoten mit mehr als 40 Freunden

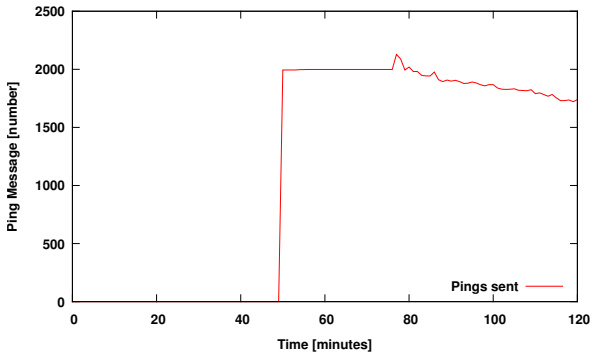
Abbildung 5.8: Szenario 3: Erreichbarkeit der Datenobjekte (Churn aktiv)

betrachtet und das Verhalten in diesen überprüft. Weiter in Abschnitt 5.3.1, 5.3.2, 5.3.3 und 5.3.2 werden die Ergebnisse der jeweiligen Szenarien zusammengefasst.

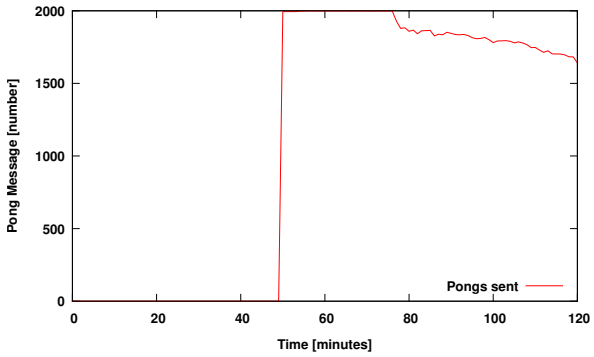
Es wird mit den Ergebnissen der Simulationen gezeigt, dass das modifizierte *DunbChord* Overlay mit dem implementierten Datenhaltungsservice funktioniert. In Kombination mit der DHT-Eintragsverwaltung werden sogar noch signifikant bessere Ergebnisse in Bezug auf Datenerreichbarkeit erzielt. Noch fest-




(a) PoSs online



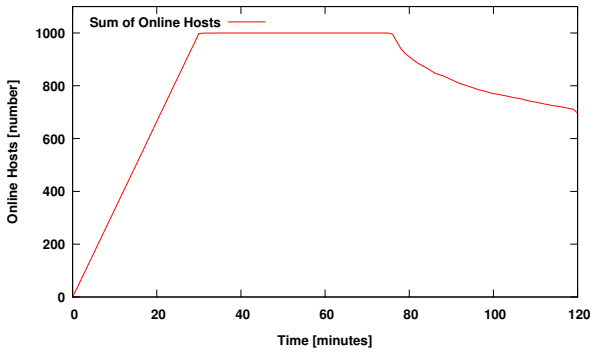

(b) Kosten Ping Nachrichten



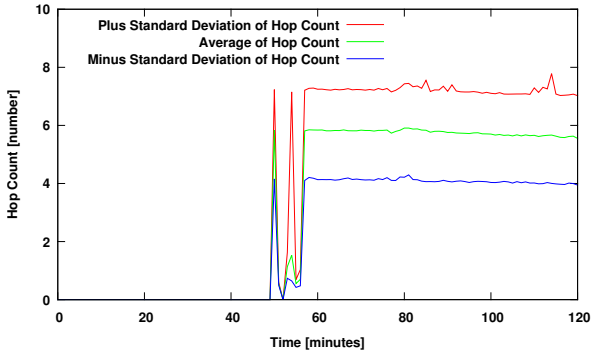

(c) Kosten Pong Nachrichten

Abbildung 5.9: Szenario 3: Online PoSs und Kosten der Ping und Pong Nachrichten (Churn aktiv)

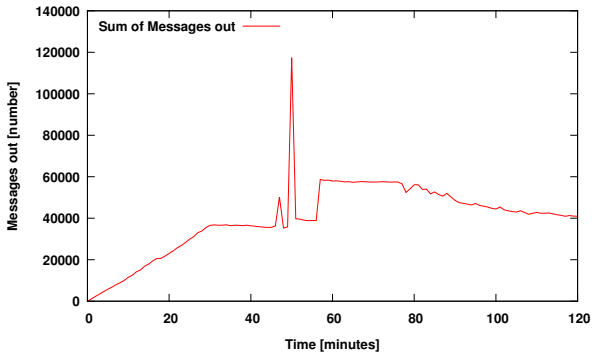
zuhalten ist, das Knoten die einen großen Freundeskreis besitzen, eine höhere Datenerreichbarkeit erzielen, da es einfach eine größere Wahl an möglichen PoSs gibt.



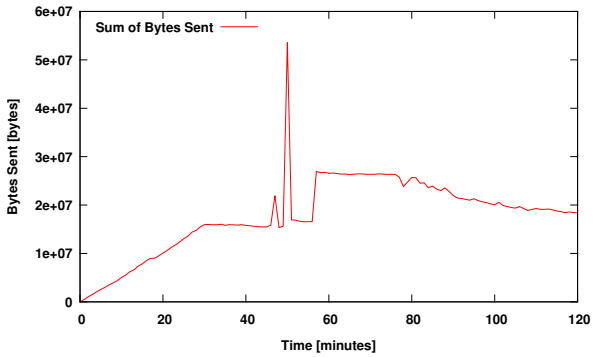

(a) Anzahl Knoten im Netzwerk




(b) Anzahl Hops




(c) Gesendete Nachrichten




(d) Bytes gesendet

Abbildung 5.10: Szenario 3: Überblick über den Traffic, Hops und Anzahl Knoten im Netzwerk (Churn aktiv)

# Kapitel 6

# Zusammenfassung und Ausblick

Diese Arbeit beschäftigte sich damit, eine Dunbar-basierte Technik zu implementieren und zu evaluieren. Als eine solche Technik wurde ein Datenhaltungsservice implementiert und ausgewertet. Hierbei wurde die Basis-Implementierung von Chord aus dem Simulator *PeerfactSIM.KOM* modifiziert und um ein solches System erweitert. Die Idee dafür wurde aus [1] aufgegriffen und daraus ein Entwurf für Chord entwickelt. Hierdurch wurde Chords Problem der nicht vorhandenen Datenhaltung gelöst.

In dieser Arbeit wurde ein System zur Datenhaltung für *Distributed Online Social Networks* untersucht. Hierbei werden Kopien von Daten an Online-Freunde, welche als Point of Storages (PoSs) aggieren, gesendet und für Datenanfragen verwaltet. Hierdurch wird es möglich, dass Daten verfügbar sind, obwohl die Besitzerknoten nicht mehr online sind. Zwei PoSs werden für den Datenhaltungsservice verwendet, wodurch eine gute Datenerreichbarkeit in den Simulationen erreicht wird.

Zudem wurden Mechanismen eingebaut, die dafür sorgen, dass Datenobjekte weitestgehend gut im P2P Overlay erreichbar bleiben, falls Knoten das Netzwerk willkürlich oder gewollt verlassen. Als Freundesbeziehungen wurden reale Facebook-Beziehungen [12] verwendet, wobei hier für die Simulation die ersten 1000 Knoten und deren Beziehungen untereinander extrahiert wurden.

Die Ergebnisse aus Kapitel 5 zeigen, dass die modifizierte Chord-Implementierung *DunbChord* und dessen Datenhaltungsservice funktionieren. Durch das aktivieren der implementierten DHT-Eintragsverwaltung (s. Abschnitt 5.3.2) wird nochmals eine deutlich bessere Datenerreichbarkeit erzielt.

### Ausblick

Eine weitere Möglichkeit besteht darin, das Verhalten mit drei oder mehr PoSs zu untersuchen und auszuwerten. Dies müsste die Erreichbarkeit anheben. Weitere Untersuchungen könnten in dem Bereich der Anonymisierung bzw. Privatsphäre unternommen werden.

In zukünftigen Arbeiten könnte man außerdem weitere Dunbar-basierte Techniken Sozialer Netzwer-

ke angehen. Hier gibt es z.B. die Möglichkeit noch mehr auf den Dunbar Circle [13] einzugehen, wobei hier das *Level* der Beziehungen der einzelnen Knoten aufgezeigt wird. Man könnte ein Routing über die verschiedenen *Level* implementieren und auswerten. Daten würden hierdurch privater verschickt werden, indem die Daten z.B. nur über die jeweils wichtigsten Freunde der Knoten verschickt werden. Eine Verschlechterung der Performance in Bezug auf die Hops, die die Nachrichten benötigen, ist hier zu erwarten, da die Daten nicht auf dem schnellsten, sondern auf einem sichereren, aber langsameren Weg weitergegeben werden. Die Fingertabelle von Chord müsste für das Routing über Freunde ausgeschaltet werden, damit ein Knoten nicht automatisch andere Knoten aus der Finger-Tabelle (keine engen Freunde) als möglichen Routingpfad verwendet, um möglichst schnell eine Nachricht zum Zielknoten zu schicken. Er soll den (möglicherweise langsameren) Weg über die wichtigsten Freunde (z.B. Level 1 Freunde) nehmen. Hierbei könnte man wiederum viel ausprobieren, wie z.B. das Routing über Level-1-Freunde, Level-2-Freunde oder eine Kombination verschiedener Level.

# Literaturverzeichnis


[1] Barbara Guidi, Tobias Amft, Andrea De Salve, Kalman Graffi, and Laura Ricci. Didusonet: A p2p architecture for distributed dunbar-based social networks. *Peer-to-Peer Networking and Applications*, pages 1–18, 2015.

[2] Anwitaman Datta, Sonja Buchegger, Le-Hung Vu, Thorsten Strufe, and Krzysztof Rzadca. Decentralized online social networks. In *Handbook of Social Network Technologies and Applications*, pages 349–378. Springer, 2010.

[3] D. Eastlake and P. Jones. Us secure hash algorithm 1 (sha1). RFC 3174, RFC Editor, September 2001. `http://www.rfc-editor.org/rfc/rfc3174.txt`.

[4] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions of Networking*, 11(1), 2003.

[5] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *ACM SIGCOMM '01: Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 31. ACM, 2001.

[6] Antony I. T. Rowstron and Peter Druschel. 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*, volume 2218 of *LNCS*. Springer, 2001.

[7] Kalman Graffi. 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.

[8] Robin IM Dunbar. The social brain hypothesis. *Evolutionary Anthropology: Issues, News, and Reviews*, 6(5):178–190, 1998.

[9] Sam GB Roberts, Robin IM Dunbar, Thomas V Pollet, and Toon Kuppens. Exploring variation

in active network size: Constraints and ego characteristics. *Social Networks*, 31(2):138–146, 2009.

[10] Alistair Sutcliffe, Robin Dunbar, Jens Binder, and Holly Arrow. Relationships and the social brain: integrating psychological and evolutionary perspectives. *British journal of psychology*, 103(2):149–168, 2012.

[11] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[12] CURRENT Lab. Facebook graph. `http://current.cs.ucsb.edu/facebook/`. Stand September 2015.

[13] Valerio Arnaboldi, Marco Conti, Andrea Passarella, and Fabio Pezzoni. Analysis of Ego Network Structure in Online Social Networks. Technical Report IIT TR-10/2012, Institute of Informatics and Telematics of CNR, 2012.

[14] Warren Matthews and Les Cottrell. The pinger project: active internet performance monitoring for the henp community. *Communications Magazine, IEEE*, 38(5):130–136, 2000.

[15] TS Eugene Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 170–179. IEEE, 2002.

# 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, 10.September 2015 Marjan Basic


**Diese DVD enthält:**

- Eine *pdf*-Version der vorliegenden Bachelorarbeit
- Alle verwendeten LaTeX- und Grafikdateien
- Den Quelltext der im Rahmen der vorliegenden Arbeit erarbeiteten Software
- Die im Rahmen der vorliegenden Arbeit gewonnenen Daten
- Die verwendeten Websites und Publikationen